D1.2 Report on intra-node and multi-node optimizations for HPC codes

Document Information

Contract Number	828947
Project Website	`www.enerxico-project.eu`
Contractual Deadline	Wednesday 30th September, 2020
Dissemination Level	PU
Nature	R
Authors	Michael Bader(TUM), Jose M Domínguez Alonso (VIGO), Albert Farrés (BSC), Okba Hamitou (ATOS), Jaime Klapp (ININ), Soline Laforet (ATOS), Rafael Mayo (CIEMAT), Anne Reinarz (TUM), Sebastian Wolf (TUM)
Contributors	Romain Brossier (UGA), Jean-Matthieu Gallard (TUM), Pablo García Müller (CIEMAT), Stephan Jaure (ATOS)
Keywords	Exascale, intra-node, multi-node, optimization, HPC

Notice:
The research leading to these results has received funding from the European Union's Horizon 2020 Programme under the ENERXICO Project (www.enerxico-project.eu), grant agreement no 828947 and under the Mexican CONACYT-SENER-Hidrocarburos grant agreement B-S-69926.

Contents

1. Introduction

This deliverable presents performance optimisations of all simulation software addressed within the ENERXICO project. It is the second of a series of three deliverables in ENERXICO's work package WP1 on "Exascale Enabling". It thus continues on Deliverable D1.1, in which the focus has been on performance and scalability assessment. Bottlenecks identified in the first deliverable are now being addressed by the ENERXICO researchers and first optimisations have been carried out.

For each code we will first summarise the findings of the performance assessment performed for Deliverable D1.1. Here we distinguish between the audits, which have been done in cooperation with the POP CoE, own performance measurements during the ENERXICO project and other assessments. We also point out further measurements that came up during the work on D1.2. For each exascale optimisation target we explain the strategy and report on work executed to improve performance and scalability, Performance measurements demonstrate respective achievements. Finally, we summarise the status of every code and lay out its progress on the road to exascale. In addition we point out further steps for the remaining project period.

2. Highlights

For each code we summarise the targets, which have been achieved in the ENERXICO project.

ALYA One of the main goals was to improve the execution time in parallel runs by reducing the load imbalance found in some scenarios. By using the Dynamic Load Balancing Library (DLB) when simulating particle-dominated systems (10M of particles) we can achieve speed-ups up to 3.5×, reducing this improvement up to 1.5× for fluid-dominated systems (0.5M of particles).

BSIT The objectives of the work were to optimise BSIT code for Nvidia Volta GPUs. We have implemented and evaluated the effect of some traditional approaches to improve finite difference codes in Volta GPUs, such as blocking, shared memory, register streaming and shuffle instructions. Our fully optimised code shows a speed-up of about 2× when compared with an unoptimised version. Furthermore, we obtain a speed-up of about 3.5× when compared with older accelerators like Intel Xeon Phi, and more than 5× when compared to state of the art HPC processors like Intel Xeon Scalable.

DualSPHysics The POP CoE Performance and Scalability Analysis, which has been delayed for Deliverable 1.1, has now been completed. The extension of the code to multi-GPU is underway, we have important advances but we have encountered several difficulties that are related to architecture differences between old and new GPUs that are expected to be solved within the next few months. An update of the POP audit will then be necessary for the latest GPU architectures.

ExaHyPE The objectives of the work were to port and perform scaling analysis on AMD Rome Zen2 architecture by taking into account the NUMA architecture. The scaling is performed on up to 11000 cores. By targeting AVX2 instruction sets, the optimised XSMM library achieves a speed up of 69% compared to a generic implementation of GEMM.

SeisSol A major goal was to optimise SeisSol, which was originally targeted to Intel architectures, for AMD processors. We identified a more pronounced NUMA architecture as

major difference. By using NUMA aware parallelisation techniques we were able to achieve a speedup of $\approx 26\%$ on AMD Rome architectures. The optimisations were also evaluated on Intel Skylake and achieved a speedup $\approx 18\%$.

SEM46 Following the results of the POP audit, the vectorisation enablings and improvement of the memory management allowed to achieve an 24% gain in the computational time in the isotropic approximation. For the next steps, these optimisation principles shall be carried out in the anisotropic approximation. The outcome shall be tested on a large scale test case to incite the communication optimisations already introduced.

WRF The objective of the work was to perform scaling analysis on the Intel Gold Architecture for identifying bottlenecks of the code. The analysis is made following the methodology defined by the POP CoE. It identified excessive communication calls that should be reduced. Strong and weak scaling tests have been performed on up to 675 cores. By reducing the aforementioned communication calls by introducing local variables in the modules of interest, speed up of 8% compared to a generic implementation of the previous version is achieved.

3. ALYA

Alya is a high-performance computational mechanics code to solve engineering coupled problems. The different physics solved by Alya are incompressible/compressible flow, solid mechanics, chemistry, particle transport, heat transfer, turbulence modelling, electrical propagation, among others.

Multiphysics coupling is achieved in a multi-code manner. MPI is used to communicate between the different instances of Alya, where each instance solves a particular physics, with the potential of performing asynchronous executions of the different physics. Alya is specially designed for massively parallel supercomputers.

Goals in ENERXICO In ENERXICO, we focus our efforts on improving Alya performance in the context of WP4 (biofuels for transportation), which develops and implements models required in the multiphysics code of Alya to investigate combustion processes of selected fuels.

The following table summarises the effort (in person months, PM) spent on WP1 until Month 16 of the project:

Partner	PM spent
BSC	5

3.1. Performance and Scalability Assessment

Alya implements different levels of parallelisation including MPI for distributed memory systems, OpenMP and OmpSs for shared memory [5] and GPU support among others. As far as the parallelisation is concerned, in this work we rely exclusively on MPI as this is the version used in production runs. The MPI parallelisation of Alya relies on Metis to partition the mesh in subdomains. It should be noted that contrary to classical implementations of the Finite Volume and Finite Difference methods, halo cells or halo nodes are not required to assemble the matrices or residuals in the Finite-Element method [9]. Avoiding duplicated work on these cells, provides a certain advantage in the assembly phase for the Finite-Element method, where the scalability only depends upon the control of the load balance.

The Alya benchmarks have been performed on systems with Intel Skylake architectures (JUWELS and MareNostrum4). MareNostrum4 is the Tier-0 system hosted by BSC, Spain. It is based on Intel Xeon Platinum processors from the Skylake generation. It is a Lenovo system composed of SD530 Compute Racks, an Intel Omni-Path high performance network interconnect and running SuSE Linux Enterprise Server as operating system. Its current LINPACK R max Performance is 6.2272 Pflop/s. This general-purpose block consists of 48 racks housing 3456 nodes with a grand total of 165,888 processor cores and 390 TB of main memory. JUWELS, the Jülich Wizard for European Leadership Science, is the Tier-0 system hosted by the Jülich Supercomputing Centre, Germany. JUWELS is an Atos Bull Sequana X1000 system with dual 24-core Intel Xeon Platinum 8168 (Skylake) CPUs @ 2.7 GHz and an EDR-InfiniBand. The peak performance is 9.89 Pflop/s.

Benchmarking We have tested a Alya with two test cases:

Test Case A. A 132 million element mesh representing the flow around a sphere. It is expected to scale up to 1500 MPI tasks.

Test Case B. A 1056 million element mesh representing the flow around a sphere. It is expected to scale up to 12000 MPI tasks.

The elapsed time of only the time-integration phase has been considered, since it is the dominant part in the production runs of Alya. Likewise, the node workload for each system was selected according to the similar configurations used in scientific simulations.

Table 1 and Table 2 present the results for the Skylake systems, JUWELS and MareNostrum4 for Case A and B respectively. We can observe better performance on JUWELS for all the cases because the CPU frequency on JUWELS (2.7 GHz) is higher than on MareNostrum4 (2.1 GHz).

Number of cores	**Time (s)**	**Speedup** JUWELS	**Efficiency**	**Time (s)**	**Speedup** MareNostrum4	**Efficiency**
192	124.24	1.0	100%	129.45	1.0	100%
384	62.56	2.0	99%	67.45	1.9	96%
768	31.24	4.0	99%	33.93	3.8	95%
1536	16.45	7.6	94%	18.28	7.1	89%

Table 1: Alya results for Test Case A running in Skylake architectures.

Number of cores	**Time (s)**	**Speedup** JUWELS	**Efficiency**	**Time (s)**	**Speedup** MareNostrum4	**Efficiency**
1152	372.52	1.0	100%	451.38	1.0	100%
2304	196.48	1.9	95%	262.32	1.7	86%
4608	99.65	3.7	93%	124.98	3.6	90%
9216	61.34	6.1	76%	86.61	5.2	65%

Table 2: Alya results for Test Case B running in Skylake architectures.

Figure 1 and Figure 2 shows the speedup results for the Skylake systems, JUWELS and MareNostrum4 for Case A and B respectively. We can observe that on all the cases is the speedup is better in JUWELS than for MareNostrum4. One of the causes of this difference on the efficiency is the network of each system, Mellanox EDR-InfiniBand on JUWELS and Intel Omni-Path on MareNostrum4.

3.2. Improvements to Performance and Scalability

3.2.1. Dynamic load balance

Lagrangian particle tracking is of great interest in engineering. In the literature, particle tracking is usually solved within the same code, after obtaining the solution of the Navier–Stokes equations. However, in a distributed memory context, it is very likely that particles are concentrated in very few MPI partitions, resulting in a very poor load balance. This work will thus address this issue: the dynamic load balance applied at the MPI level of parallelism.

Synchronous and asynchronous couplings In this work, we only consider a one-way coupling, which means that the particles are transported by the fluid but have negligible effects on the fluid dynamics. Two solutions will be analysed and compared. First, synchronous coupling using the same instance of the code, and thus the same subdomain partitioning for the fluid and the particles. Second, in order to gain asynchronism, the fluid to particle coupling is also achieved via a multi-code strategy using two instances of the Alya code. In this case, the fluid and particle

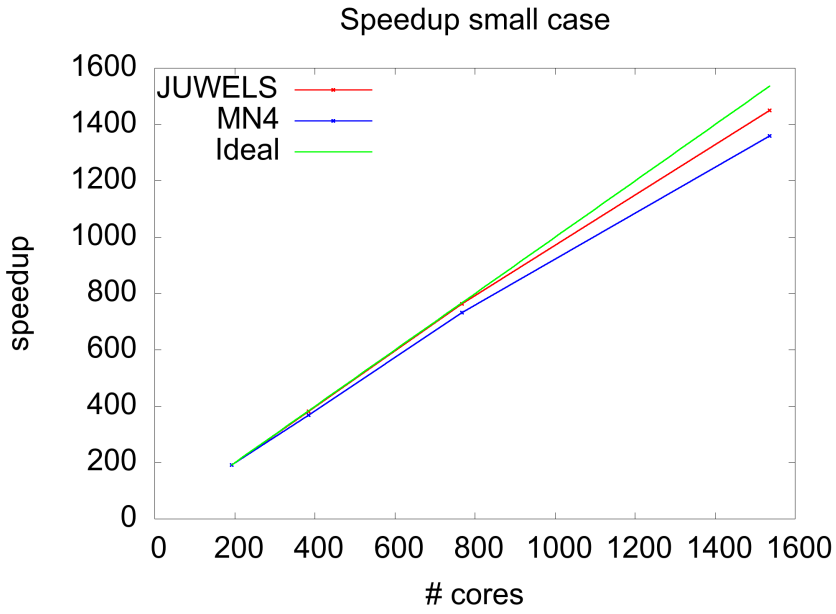


Figure 1: Alya speedup for Test Case A running in Skylake architectures vs Ideal speedup.

domains are partitioned independently, with a different number of subdomains. At the end of a time step, once the velocity and pressure of the fluid are obtained, the velocity field is sent to the particle subdomains, via the `MPI_ISend` function.

In the particle code using the synchronous coupling strategy, the load is dramatically imbalanced, and in the worst case, all the particles may be in a single subdomain. Moreover, even if we were able to distribute the particles in a homogeneous manner, they will be migrating over time and may end up producing a load imbalance. On the other hand, when using the asynchronous coupling strategy, one must select the distribution of MPI processes between the fluid and particle codes. Moreover, as the load of the particles will change during the execution, the optimal distribution of MPI processes between fluid and particle codes can change as well.

To attack these problems, we implement a dynamic solution, which is applied at runtime, the Dynamic Load Balancing Library (DLB). DLB [4] is an independent and interoperable dynamic library that can help parallel applications improve their load balance. This library was developed and is actively maintained by the BSC Computer Science team. DLB is applied at runtime meaning that we do not need to analyse specific inputs or modify the application code. The philosophy of the library is to exploit the computational resources (i.e. CPUs or cores) of the MPI processes blocked in an MPI blocking call by other processes running on the same node, by spawning more threads of the second level of parallelism (i.e. in our case, OpenMP). When running with DLB, whenever an MPI process detects that it is not using its cores (i.e. it is waiting in a blocking MPI call), it will lend its resources to the system. Another process running on the same node can then use these cores and spawn more OpenMP threads to parallelise further the end of its computation.

In Figure 3, we can see different traces of an execution of the Alya code with MPI processes and two OpenMP threads each. The X-axis represents time and each horizontal line is a thread (grouped by MPI process). In these traces, the blue colour represents computation of the fluid

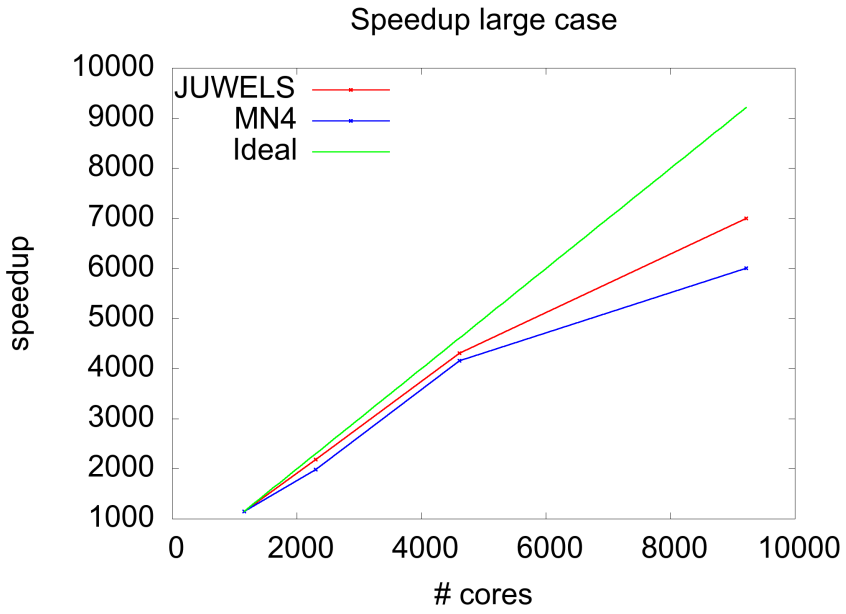


Figure 2: Alya speedup for Test Case B running in Skylake architectures vs Ideal speedup.

code, the green represents computation of the particle code and the orange global communications (in black and white, from darker to lighter). The top image is the original execution, where we can clearly see the imbalance of the particle code, and less spectacular, but significant, we can see the imbalance of the fluid code. In the middle image, we can see the same execution with DLB. In the bottom figure, we show a zoom of one of the nodes (the most loaded one and consequently the bottleneck). We can see that when the particle code is running (green), only one MPI process have computation to perform, and all the other MPI processes are waiting in a global communication (orange). At this point, the process running the particles is able to use the 16 cores of the node, by spawning 16 OpenMP threads. The same observation can be made in the fluid code whenever imbalance is present.

Results In Figure 4, we can see the average execution times for the different configurations. The first conclusion is that the performance, when the computation is dominated by the fluid (0.5M particles, left-hand side charts), differs from the one obtained by the particle-dominated execution (10M particles, right chart of the figure). This is due to the fact that the fluid and the particle codes have different scalability behaviours, i.e. while the fluid code scales quite well the particle code has a poor scalability due to the high load imbalance. Therefore, the global scalability highly depends on which code dominates the computation. We can see how DLB can help to improve the scalability by providing a better resource utilisation. When simulating 10M of particles, DLB can make the execution between a 45% and 72% faster (a speed-up between 1.8 and 3.5 with the same number of resources). When running with 0.5M of particles, DLB can run between 20% and 32% faster (a speed-up between 1.2 and 1.5 of the original code). When analysing the performance of the asynchronous version, we can see that it depends on the distribution of MPI processes among the fluids and the particles codes. Almost in all the cases, we can find a configuration of MPI processes that perform better than the synchronous version. But at the same time, when choosing any of the other configurations, we can see how the

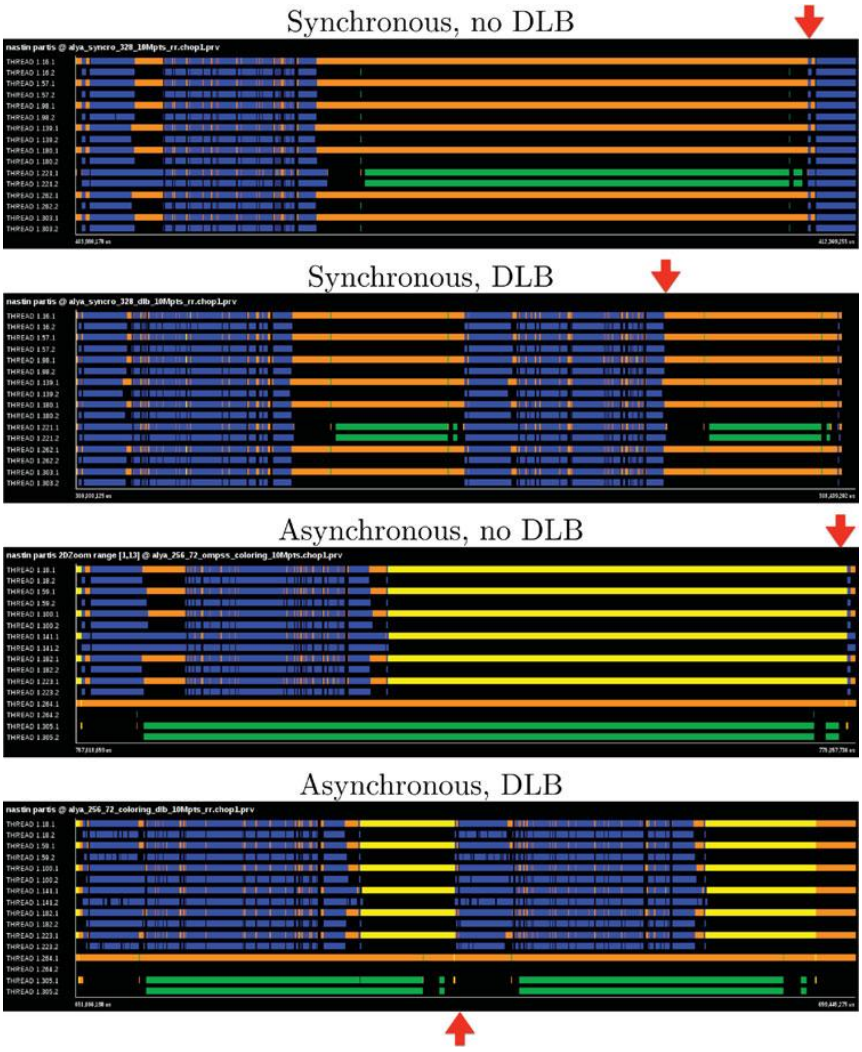


Figure 3: Global Alya traces and zoom.

performance drops; for some of the configurations, the asynchronous code with a bad distribution can be two times slower than the synchronous one. Finally, we can observe that DLB is able to hide the performance problem when using a bad distribution of MPI processes among the codes (fluids and particles). The execution time when using DLB is almost constant independently of which version we are running (synchronous or asynchronous) and how many MPI processes for fluids and particles we are running.

3.3. Summary and Outlook

We have implemented a dynamic load balance strategy to mitigate the uneven distribution of particles across the different MPI partitions. The dynamic load balance strategy equalises the execution times of all possible configurations between fluid and particle MPI processes, thus making the combination of MPI processes a less important decision from the user point of view. Moreover, using DLB when simulating particle-dominated systems (10M of particles) we can achieve speed-ups up to 3.5, reducing this improvement up to 1.5 for fluid-dominated systems (0.5M of particles).

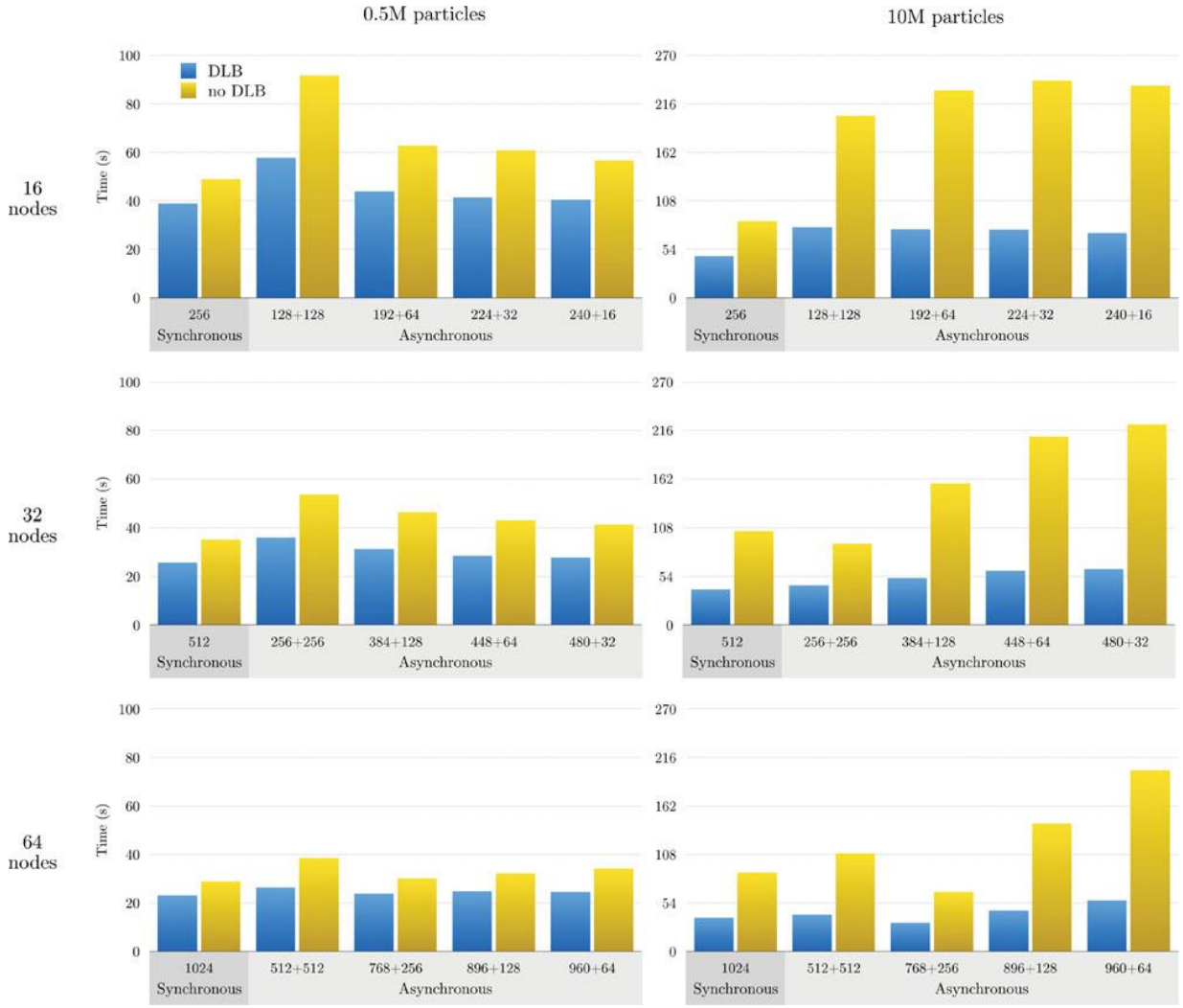


Figure 4: Alya Timings. (From top to bottom: 16, 32, 64 nodes. (Left) 0.5M particles. (Right) 10M particles).

4. BSIT

BSIT is a software platform, designed and developed to fulfil the geophysical exploration needs for HPC applications. Geophysical exploration is a field that needs huge amounts of computational resources. BSIT was developed to cope with such needs, including different types of processing systems running over a wide range or HPC architectures. The main systems included in BSIT are forward modelling, reverse time migration and full waveform inversion. In addition, the software supports different rheologies including acoustic, acoustic with variable density, elastic, viscoelastic and electromagnetic. Moreover, several levels of anisotropy are supported: VTI/HTI, orthorhombic, TTI and arbitrary anisotropy (for elastic and viscoelastic rheologies).

Goals in ENERXICO In ENERXICO, we focus our efforts on improving BSIT performance in the context of WP3 (Oil and Gas), which explores and evaluates methods implemented by BSIT for realistic modelling scenarios proposed by the industrial partners of the project.

The following table summarises the effort (in person months, PM) spent on WP1 until Month 16 of the project:

Partner	PM spent
BSC	5

4.1. Performance and Scalability Assessment

Known performance/scalability bottlenecks Roofline analysis shows a low arithmetic intensity (common in stencil codes), GPU utilisation analysis confirms that performance is bounded by the memory system. The limiting factor is the bandwidth in device memory.

Benchmarking We use roofline analysis [17] to evaluate how well BSIT performs in terms of peak performance relative to the arithmetic intensity.

By using the roofline model, we provide an insight of our application behaviour placing its performance into a graphical representation bounded by both the maximum (attainable) performance FLOPS and the memory bandwidth. The model imposes a limit on the performance based on the operational intensity of an application, showing how much room exists for improvement.

Using a maximum bandwidth of 900 GB/s via HBM (High Bandwidth Memory) and 14000 GFLOPS as maximum Floating Point (FP) performance, we can infer that a minimum operational intensity of ~ 15.55 FP operations/byte would be needed to take advantage of the total FP performance available in a Nvidia Volta GPU. Figure 5 depicts a roofline showing how close we get to the attainable performance for each memory level. Table 3 details the roofline data.

4.2. Improvements to Performance and Scalability

Implemented measures L1 and L2 memory access should be improved to reuse data and increase the arithmetic intensity. If arithmetic intensity is increased there is room for a performance (GFLOPS) improvement. Also, efficiency can be increased by fixing data alignments and improving memory access patterns.

Stencil codes like BSIT are usually bounded by memory bandwidth. Improving performance relies on increasing arithmetic intensity so relative peak performance is increased too. Making a clever use of the different memories on the device as well improving memory access patterns can resolve in an overall improved performance.

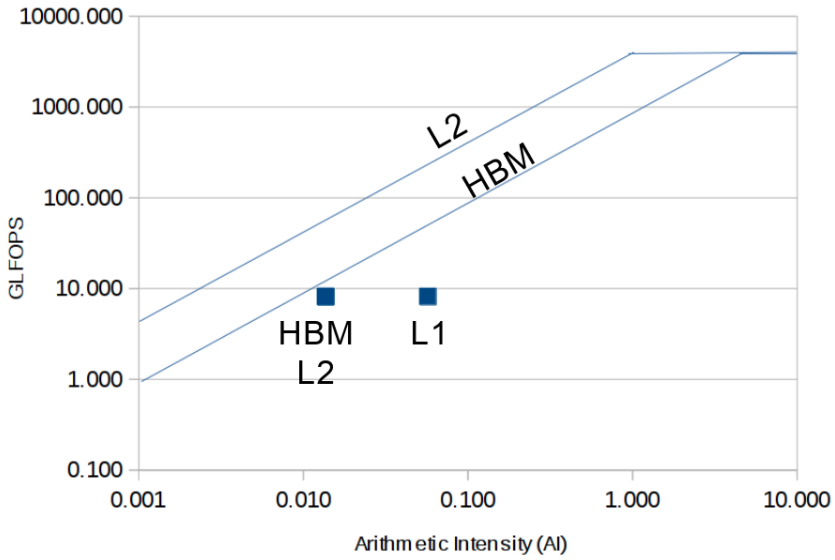


Figure 5: Roofline of attainable performance for each memory level running on Nvidia Volta GPUs vs. the observed operational intensity.

Memory level	Bandwidth (GiB/s)	Arithmetic Intensity (flop/byte)	Achieved performance (GFLOPS)	Maximum Bandwidth (GiB/s)
HBM	598.12	0.014	8.165	900
L2	595.60	0.014	8.165	4200
L1	412.79	0.057	8.165	??

Table 3: Detailed performance data.

In general, achieving huge speed-ups is difficult for stencil memory bounded codes in modern HPC architectures. Consequently, we only expect minor speed-ups relative to the current optimised version of the code.

4.2.1. Optimise BSIT for Nvidia Volta GPU architectures

When porting/re-implementing computational intensive kernels of code, users also face the decision to introduce hardware tailored optimisations that provide a most efficient version of the code compared to a base implementation that does not consider the architecture where the program will be running. We have implemented and evaluated the effect of some traditional approaches to improve finite difference codes in Volta GPUs separately and all together, such as blocking, shared memory, register streaming and shuffle instructions. We observe how each optimisation affects the throughput of the whole compute kernel and each of the costly regions of the algorithm. Additionally, we compare the performance obtained to the resulting throughput on the Power 9 processor to which Voltas are connected and other Intel-based HPC architectures. We

show that with very few optimisations and the computing capacity of the Voltas, the overall obtained performance is higher compared with previous HPC hardware when evaluating finite differences codes.

Our base implementation is the direct result of developing a Finite Differences (FD) method over a Fully Staggered Grid [1] (FSG) grid. This will lead us to a loop in time where velocities are updated based on stresses values in odd iterations and the other way around for even iterations. To update velocities, 12 different 3D stencils plus another 12 3D material interpolations for each point of the grid must be calculated. Materials are stored in a single vertex of the FSG cell for memory saving issues, trading storage per computation. On the other hand, the computation involves 28 3D-stencils calculations plus 84 3D interpolations for the material properties to update stresses. Notice that both velocities and stresses calculations are typically dominated by accesses to global memory to retrieve the data needed to update the corresponding values. Our baseline version of the code shows the two innermost loops in space mapped to a 2D Cuda grid, streaming cells to update to each thread over the slowest dimension (Y), as seen in [14]. Thread block dimensions have not been tuned for Volta architecture. Our work studies the effect of the following optimisations applied to the baseline version, with the aim of improving the performance of FSG in NVIDIA Volta GPU cards:

Y-dim blocking. Traditional approaches for finite differences codes in GPUs usually map 2D thread blocks into ZX planes (i.e., faster dimensions). Then, each thread is in charge to update velocities and stresses for the whole Y (slow) dimension. By adding an extra dimension to thread blocks and mapping it to the grid Y dimension, we increase the number of thread blocks created maximising the GPU utilisation.

Shared Memory. In Volta architecture it is possible to set up the scratchpad memory in each SM as 128 kB of cache (hardware managed) or 96 KB shared (software managed, thread block-addressable) memory. By placing some common data in this memory, it is possible to enhance reuse while maximising the memory throughput per thread.

Register Streaming. The concept of traversing the data volume in the slowest dimension can be used. By doing this, it is possible to keep the slowest dimension values in a set of registers and load only one value between iterations in the traverse direction. In our case, this is useful for the Y dimension. This approach is similar to that used by [13], with some differences to take into account: our algorithm is FSG whereas theirs is Simply Staggered Grid (SSG) (and thus, they consider a smaller number of variables), and they use a spacial order of 4 whereas ours is 8, which raises the number of registers required in the kernel.

Shuffle Instructions. Efficient data exchange between threads within the same warp can be achieved by using shuffle instructions. Also, a shuffle instruction is faster than shared memory since it only requires one instruction versus three for shared memory (write, synchronise, read). Shuffle instructions can be used to exchange data between threads within the same warp when computing stencils in Z (fast) dimension.

Results The proposed optimisations were implemented and evaluated for a computational grid built using FSG cells. Table 4 shows the environment used for the evaluation. Figure 6 depicts the achieved throughput on Volta architecture for all the proposed optimisations and the two different stages of the algorithm, the velocities and the stresses update, as well as the aggregated (denoted as *global*) throughput. The metric used is millions of cells per second (MCell/s). This is a typical throughput metric used for finite-difference kernels. It reports the rate at which the code can compute the required stencil formulas for each cell of the grid. MCell/s is an objective value that measures the actual performance of the problem independently of the architecture.

System	4 × GPU NVIDIA Tesla V100-SXM2 with 16GB HBM2
Compiler	NVIDIA CUDA Compiler 9.1.85
CUDA_FLAGS	-gencode arch=compute_35,code=[sm_35,sm_37] -gencode arch=compute_50, code=sm_52 -gencode arch=compute_70,code=[sm_70]
Grid Size	128 × 128 × 128

Table 4: Evaluation environment specifications

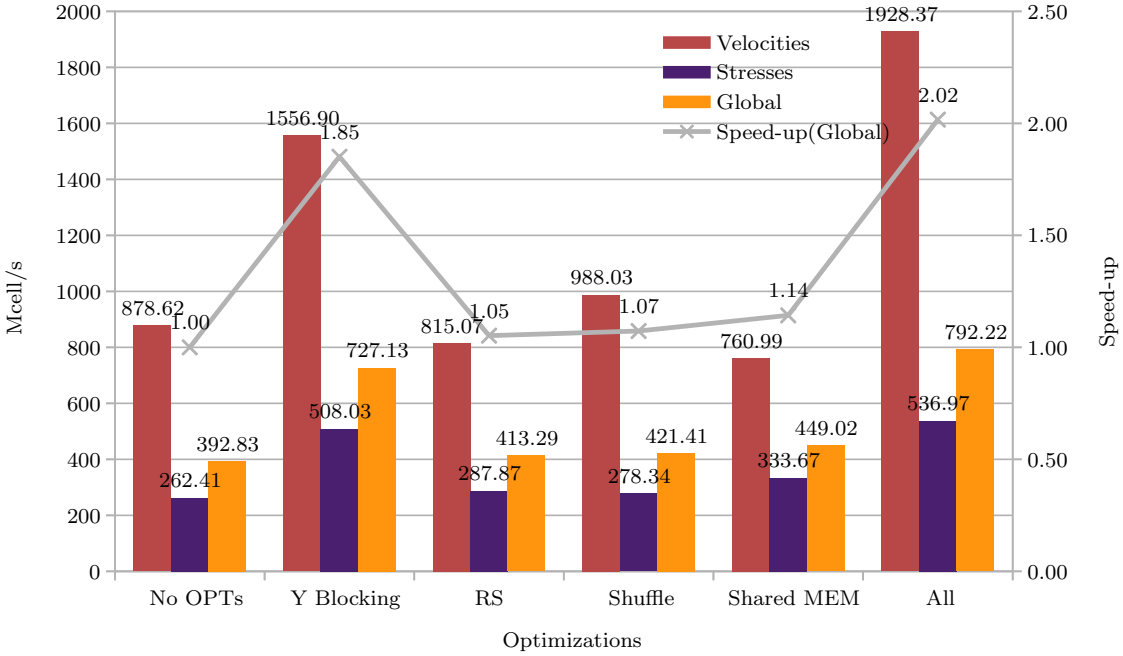


Figure 6: FSG wave propagator throughput on one NVIDIA Volta V100 GPU.

The optimisations were enabled independently; thus the figures show the isolated effect of each optimisation. Besides, the achieved throughput enabling all optimisations at the same time is reported as *All*.

For additional reference, Figure 7 includes the performance of the same algorithm when optimised for the first and second generation Intel®Xeon Phi™ processors (denoted in the Figure *knc* and *knl* respectively), as well as the performance obtained in the third and fourth iteration of the MareNostrum supercomputer (denoted *snb* and *skx* respectively). The performance achieved in the host processor, a dual socket IBM Power9 8335-GTG @ 3.00GHz with 20 cores, where Voltas GPU cards are attached is also reported as *p9*. It is worth mentioning that no effort has been put to optimise the algorithm for this last architecture.

4.3. Summary and Outlook

We have shown a set of optimisations, applied to a Finite Difference Numerical method solving elastic wave propagation equations with support arbitrary anisotropy on NVIDIA Volta GPUs. The evaluated set of optimisations ranges from memory to compute optimisations. Our fully optimised code shows a speed-up of about 2× when compared with an unoptimized version. Furthermore, we obtain a speed-up of about 3.5× when compared with older accelerators like

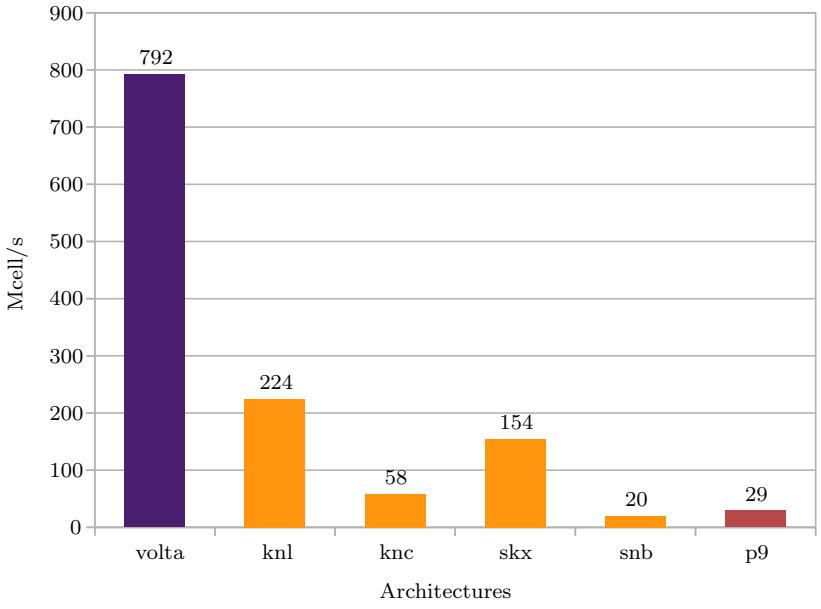


Figure 7: Throughput architecture comparison.

Intel Xeon Phi, and more than 5× when compared to state of the art HPC processors like Intel Xeon Scalable.

5. DualSPHysics

DualSPHysics (`https://dual.sphysics.org/`) is a hardware accelerated Smoothed Particle Hydrodynamics code developed to solve free-surface flow problems. The code is developed to study free-surface flow phenomena where Eulerian methods can be difficult to apply, such as waves or impact of dam-breaks on off-shore structures. DualSPHysics is a set of C++, CUDA and Java codes designed to deal with real-life engineering problems.

DualSPHysics is an open-source code developed and released under the terms of GNU General Public License (GPLv3). Along with the source code, a complete documentation that makes easy the compilation and execution of the source files is also distributed. The code has been shown to be efficient and reliable. The parallel power computing of Graphics Computing Units (GPUs) is used to accelerate DualSPHysics by up to two orders of magnitude compared to the performance of the serial version.

5.1. The Black Hole (BH) Oil Reservoir simulation code.

ENERXICO is developing a computer code called Black Hole (or BH code) for the numerical simulation of oil reservoirs, based on the numerical technique known as Smoothed Particle Hydrodynamics or SPH. This new code is an extension of the DualSPHysics code and is the first SPH based code that has been developed for the numerical simulation of oil reservoirs and has important benefits versus commercial codes based on other numerical techniques.

The BH code is a large-scale massively parallel reservoir simulator capable of performing simulations with billions of "particles" or fluid elements that represents the system under study. It contains improved multi-physics modules that automatically combine the effects of interrelated physical and chemical phenomena to accurately simulate in-situ recovery processes. This leads to the development of a graphical user interface multiple-platform application for code execution and visualisation, and for carrying out simulations with data provided by industrial partners and performing comparisons with available commercial packages. Furthermore, a large effort is being made to simplify the process of setting up the input for reservoir simulations from exploration data by means of a workflow fully integrated in our industrial partners' software environment.

An oil reservoir is composed of a porous medium with a multiphase fluid made of oil, gas, rock and other solids. The aim of the code is to simulate fluid flow in a porous medium, as well as the behaviour of the system at different pressures and temperatures. The tool should allow the reduction of uncertainties in the predictions that are carried out.

An oil reservoir system is very complex and a high resolution realistic simulation requires the use of up to one to ten thousand million particles. In one GPU we can simulate up to 200 millions particles, so a simulation with say one thousand million particles require the use of at least five GPUs or more depending upon the amount of particles that we load per GPU. An important part of this work is to produce a multi-GPU version of both the DualSPHysics and BH code that will allow us to perform high resolution simulations and to be prepared for the exascale technology. In section 5.3 we describe the initial stages towards this goal.

5.2. POP CoE Performance Analysis

The single-GPU version of DualSPHysics was analysed as part of the parallel application performance assessment (Audit) service by the POP Centre of Excellence. Due to a delayed start of the "Mexican half" of the ENERXICO project, the POP Audit for DualSPHysics could only partly be provided in Deliverable 1.1 (as had been planned in the proposal). The audit has been

finished in the meantime and results are reported in the remainder of this section – see also the respective POP report that is added in the appendix.

5.2.1. DualSPHysics Test Cases

The test case is a very typical case in the Smoothed Particle Hydrodynamics (SPH) method that represents a Dam Break hitting a structure and that is frequently applied to compare performance. The test case was analysed for two numbers of particles and physics complexity doing a total of four different experiments. The number of particles selected are 500k and 2M; the physics complexity is denoted as `Sp` for the simplest physics and `Cx` with more complex physics. In this application, the simulation time depends mainly on the number of particles, the physical time to simulate and the physics applied.

Experiments were executed in the CTE-POWER HPC system at the Barcelona Supercomputing Center. The technical description of a compute node is as follows:

- Platform: CTE-POWER
- 2 x IBM Power9 8335-GTH @ 2.4GHz (3.0GHz on turbo, 20 cores and 4 threads/core, total 160 threads per node)
- 512GB of main memory
- 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2
- Software: cuda-9.1+gcc-6.4.0

5.2.2. DualSPHysics Parallel Efficiency

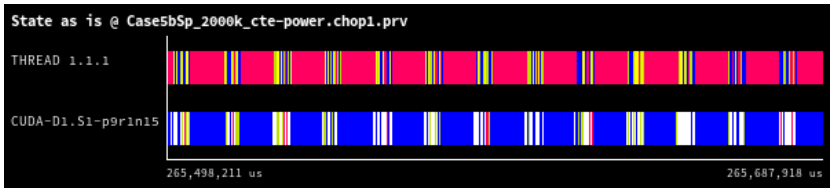
In the POP CoE, the parallel efficiency measures the parallel runtime impact on total execution time and this is composed by two sub-metrics – load balance and communication efficiency. Load balance is focused on the useful computation time per parallel task and the communication in the overhead produced by parallel runtime. In this case, the communication efficiency is related to operations like Scheduling/forkJoin and memory transfer (i.e. memcpy).

Table 5: Single GPU Parallel Efficiency for DualSPHysics

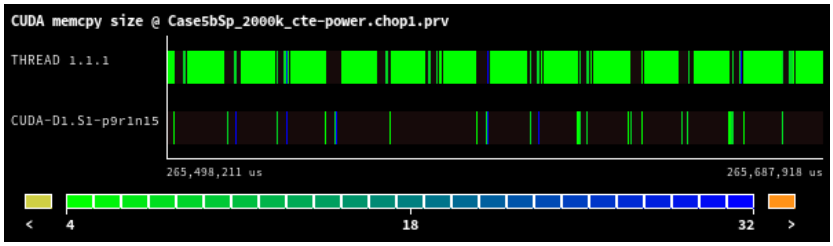
	Sp		Cx	
Efficiencies	500k	2M	500k	2M
Parallel efficiency	35.89%	46.98%	38.15%	47.63%
Load balance	70.88%	59.73%	65.49%	56.82%
Communication efficiency	50.64%	78.66%	58.25%	83.82%

This DualSPHysics version is at tracing level a parallel application with two parallel tasks. One task in the host and another in the GPU device. Therefore the efficiency metrics are measured as a work done by two parallel tasks. The parallel degree at CUDA level is done by using 3,196 grids and 128 blocks for the 500k cases and 13,745 grids and 128 blocks for the 2M cases. The number of GPU threads for 500k is 409,088 and 1,759,360 for 2M case, both for Sp and Cx complexity physic. However, the individual behaviour of GPU threads are not traced by the tool used in this Audit, but the whole behaviour on a GPU device is traced as a timeline for a parallel task.

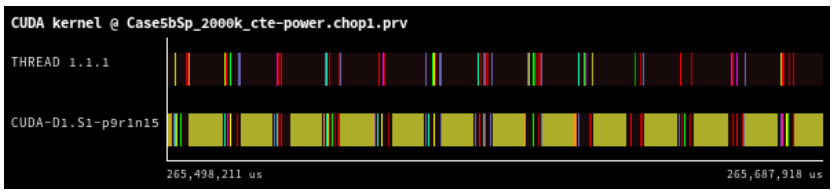
Table 5 shows the efficiencies for the four cases analysed. As can be seen in Table 5, the load balance efficiency decreases from 500k to 2M for the Sp cases, because most computational



(a) Running computation (blue) and memory transfer (magenta) states.



(b) Transfer size for CUDA `memcpy` API. The green-blue scale bar at the bottom depicts the transfer size in bytes.



(c) CUDA Kernels. Brown colour corresponds to the `KerInteractionForcesFluid` kernel.

Figure 8: A `paraver` trace chop for Cx_2M case. First timeline in each plot corresponds to task in the host and second timeline to the task in GPU device.

work is done at GPU level rather than in the host. Opposite behaviour can be observed for communication efficiency, because the time percentage for the memory transfer operations has less weight on 2M cases despite the absolute time increases about the 500k cases. Analysing the whole parallel efficiency, the values reported are low that is mainly related with memory transfer operations.

5.2.3. Analysing CUDA API and Kernels

To identify the source of the poor efficiency reported in Table 5 a deeper analysis is done at CUDA kernels level and CUDA API. To do this, the `paraver` traces are analysed for the four cases. A `paraver` trace chop for 2M particles for Cx case is shown in Figure 8. As can be seen in Figure 8a, most of time the host task is doing memory transfer operations (in magenta colour) in small sizes of 4 bytes (green colour in Figure 8b) for the `KerInteractionForcesFluid` kernel (olive colour in Figure 8c), this behaviour is also observed for the others cases.

Table 6 shows the time percentage that the `KerInteractionForcesFluid` kernel represents in the kernels total time for the four cases. It can be observed that the number of particles has more impact on the percentage than the physic complexity and it is representing more than 70% about the all kernels total time.

Memory transfer time is mainly related with the CUDA API `cudaMemcpy`, in Table 7 the time percentage and count of calls are shown for the four cases. Over this percentage, the `cudaMemcpy` calls done from the host to the device during the running of `KerInteractionForcesFluid` kernel is around the 70% in the analysed cases.

Table 6: Timing and Calls for the `KerInteractionForcesFluid` kernel

Case	Time(%)	Time	Calls	Avg	Min	Max
5bSp_0500k	75.53%	47.40s	21,436	2.21ms	2.13ms	2.53ms
5bSp_2000k	83.14%	339.38s	35,461	9.57ms	9.22ms	10.90ms
5bCx_0500k	79.60%	211.73s	64,188	3.30ms	3.13ms	3.94ms
5bCx_2000k	85.58%	1.6e+03s	111,092	14.30ms	13.87ms	16.67ms

Table 7: Timing for the API `cudaMemcpy`

Case	Time(%)	Time	Calls	Avg	Min	Max
5bSp_0500k	60.13%	53.98s	128,631	419.62us	18.42us	4.56ms
5bSp_2000k	75.92%	361.11s	212,781	1.70ms	17.38us	22.51ms
5bCx_0500k	60.03%	235.00s	449,330	523.00us	17.67us	110.32ms
5bCx_2000k	80.69%	1.7e+03s	777,658	2.17ms	17.22us	64.39ms

5.2.4. Identified Performance Issues

By analysing the obtained results, two main issues can be considered to improve the performance. On the one hand, the most logical step is to increase the parallelism degree by implementing a Multi-GPU version. On the other hand, a more complex analysis is required for the `KerInteractionForcesFluid` kernel that seems require a refactoring to increase the data transfer size between the host and the GPU devices. The impact of this kernel on the performance possibly could be reduced by using a Multi-GPU version without doing a refactoring of such kernel. However, the performance improvement can be greater by optimising the kernel implementation. The problem with this kernel is that each particle has to evaluate which surrounding particles are close enough and then calculate the interaction with each of them. This implies the use of several nested loops and divergence between the threads, because each thread calculates the values for a particle, but each particle interacts with different neighbouring particles and different numbers. This also leads to irregular memory access patterns and this requires a high volume of records that reduces the percentage of occupancy.

5.2.5. Summary of the POP audit

In the final POP report (see apendix), the key findings were summarised as follows:

- The kernel KerInteractionForcesFluid represents more than 70% of the total kernels execution time.
- The kernel KerInteractionForcesFluid has more impact on the execution time in the four cases, due to the small data copied by memcpy API. This behaviour is similar for simple and complex physic cases.
- Computation time is less than 50 % of the total runtime for the four cases.
- CUDA memcpy is representing around the 60% of the execution time that is produced by the the kernel KerInteractionForcesFluid.
- Application's Kernel needs an additional refactoring to increase the degree of parallelism, it could be an Multi-GPU or explicit Streams programming.

5.3. Multi-GPU implementation of DualSPHysics: advances and implementation difficulties

The first multi-GPU approach of DualSPHysics was tested in the BSC (Barcelona Supercomputing Center BSC-CNS) in 2012 and the results were published in Domínguez et al. [3]. This multi-GPU version combines CUDA and MPI, where each MPI process handles one GPU. The division of the domain is done in one direction; therefore, each process usually only has to communicate with one or two neighbouring processes. The communication between processes is carried out through asynchronous sending of messages and synchronous reception to overlap communication and calculation times. In addition, it includes a dynamic load balancing to redistribute the particles after some time steps and minimise the synchronisation times between MPI processes. A more detailed description of the implementation and its results can be found in Domínguez et al. [2]. In 2013 an improved version was developed where CUDA streams, pinned memory and asynchronous transfers between CPU and GPU memory was used to improve the overlapping between communication and calculation times. This version was also tested in the BSC and the results were presented in the 8th International SPHERIC Workshop. The latest version improved the efficiency results of the previous version which were already very good. An efficiency close to 100% was achieved simulating 8 million particles per GPU on 128 GPUs Tesla M2090 of the MinoTauro GPU cluster of the BSC. The new efficiency tests are done in the CTE IBM Power9 cluster of the BSC. This cluster hosts 4 GPUs Tesla V100 per computation node. The software used in the tests carried out in the MinoTauro machine (see original specifications in Domínguez et al., [2]) is not available, so the improved multi-GPU version is compiled with the new software available in CTE-POWER cluster (CUDA 9.2, GCC 6.4.0 and Open MPI v3.0.0). Several problems were found when repeating the efficiency test in this cluster using the current software. Some code used for block size optimisation to run the CUDA kernels is not compatible with the new version of CUDA, so some changes had to be done to remove this code. The most important problem was that many simulations failed for different reasons and at different times. After considerable time and effort, it was found that the problem was in the communication with MPI. One MPI process sent some data, but the destination process received other data. This problem has already been discussed in some forums. There is some incompatibility between the use of pinned memory and MPI when this pinned memory allocated by CUDA is used in asynchronous message sending. The code was updated to detect this specific problem and also to avoid it. Finally, the corrected version was used to test the efficiency in the CTE-POWER cluster without any further failure. The efficiency test (weak scaling) was done up to 64 GPUs (the maximum allowed by the user account) simulating 4, 8 and 16 million particles per GPU. An efficiency of 91 % was achieved by simulating 8 and 16 million particles per GPU and 85% by simulating 4 million particles per GPU on 64 GPUs Tesla V100. The efficiency obtained is not bad, but it is less than the efficiency obtained in the MinoTauro cluster. There are important differences in software and hardware between the two efficiency tests. The most important difference is in the computational power of the GPUs Tesla V100 and Tesla M2090. The new GPU is almost 16 times faster than old GPU simulating the same testcase with 8 million particles, so the calculation time was drastically reduced while the communication time is similar.

6. ExaHyPE

The ExaHyPE engine solves systems of hyperbolic PDEs, as stemming from conservation laws. Models for seismic wave propagation problems, as addressed in ENERXICO's WP3 (Oil and Gas), have been developed within the ExaHyPE project (`www.exahype.eu`) and are being further developed in the ChEESE cluster of excellence (`www.cheese-coe.eu`). ExaHyPE is based on high-order Discontinuous Galerkin (DG) discretisation on tree-structured Cartesian meshes. For non-linear problems it offers an a-posteriori Finite-Volume limiter. Parallelisation of the ExaHyPE engine relies on the underlying Peano framework (`www.peano-framework.org`) for parallel adaptive mesh refinement. MPI is used for distributed-memory parallelism. Shared-memory parallelisation relies on Intel's Threading Building Blocks (TBB). The prime model for seismic wave propagation is based on the elastic wave equations on curvilinear meshes. It allows to simulate problems on suitably complicated geometry (topography, curved faults, e.g.) with a substantially reduced meshing overhead compared to approaches that work on unstructured grids.

Goals in ENERXICO ExaHyPE is a flagship code of the European Centre of Excellence ChEESE, which hosts the majority of activities to advance performance and scalability of ExaHyPE towards exascale (as for SeisSol, cmp. Section 7). In ENERXICO, we therefore focus on prototyping performance improvements for novel use cases of ExaHyPE in the context of WP3 (Oil and Gas), in particular towards inverse problems (Bayesian inversion) and uncertainty quantification. Additional activities towards "exascale enabling" (i.e., WP1) focused on collaboration opportunities within the consortium (TUM, BULL/ATOS, ...).

The following table summarises the effort (in person months, PM) spent on WP1 until Month 16 of the project:

Partner	PM spent
TUM	3
Bull/ATOS	4

6.1. Performance and Scalability Assessment

A POP audit for ExaHyPE has been attempted as part of the ChEESE CoE, before the start of ENERXICO. However, due to limited support of the POP toolset for Intel TBB, it was not possible to obtain audit results. We therefore did not attempt another POP audit in ENERXICO, but relied on performance assessments and findings identified in the ExaHyPE project. In addition, ATOS did a performance and scalability assessment in the first half of the ENERXICO project (cf. Deliverable 1.1).

Issues known since the start of the project:

- **MPI Scaling:** ExaHyPE relies on tree-structured grids and assigns subtrees to MPI ranks. This approach follows the Peano framework's original orientation towards multi-level and multi-grid problems. The subtree approach leads to a very coarse-grain load balancing, such that optimal scaling is only achieved at certain "sweet spots": the number of ranks should be related to powers of 3^d, where d is the spatial dimension.
- **TBB Scaling:** TBB scaling depends strongly on the arithmetic intensity of the underlying models. For the linear elasticity model used in ENERXICO, scaling on Skylake nodes, e .g., drops significantly after 14 cores. These scaling limits result from a mixture of NUMA

effects, overheads of the task-based parallelisation approach and impact of sequential execution of the tree-oriented mesh traversals. The default strategy is therefore to work with multiple ranks per compute node.

Issues identified by ATOS during tests from D1.1

- **Cache Stalls:** A significant proportion of cycles (≈32 %) are spent on data fetches from cache. The strategies adopted in the applications (data alignments ...) do not seem to work optimally. Such issues are related to intra-node parallelisation and therefore, the data sharing should be investigated in depth.
- **MPI Time:** The MPI time is high and represents almost 31% of the elapsed time. It is highly used by the MPI function "Iprobe" which is a non-blocking test for messages. Such problem may be caused by a non-optimal communication schema. On the other hand, it is shown that there is no MPI imbalance issue. Here, a dedicated rank is in charge of performing the load balance. Therefore, a big portion of time is spent in synchronisations.
- **Vectorisation:** A significant fraction of floating-point arithmetic instructions (93 %) are scalar and do not benefit from the advanced vectorisation capabilities of the processor. Only a small portion of the code is vectorised (≈7%) using AVX512 instruction set. This is due to the generic kernels employed in this test. In the optimized kernels a very high vectorisation level is reached.

Of these issues some arise only when using the generic (non-optimised) kernel variants of ExaHyPE. The issues regarding cache-stalls and vectorisation have been resolved in the ChEESE-CoE for the linear applications of ExaHyPE.

Benchmarking ATOS has used Intel's advisor to benchmark ExaHyPE and to ensure comparability of the results we will continue to use Intel Advisor to measure, e.g. vectorisation levels.

For MPI scaling we use internal tools to measure run-times.

6.2. Improvements to Performance and Scalability

Port to Peano4 The issue with MPI scaling only being available at certain "sweet spots" is caused by the underlying AMR framework Peano, which follows a tree-oriented scheduling of subgrids with the tree-structured adaptive Cartesian meshes, which may lead to large granularity of partitions. The new version Peano4 is developed by the group of Tobias Weinzierl at Durham University and should become available during the project period. It is designed to at least partially resolves these issues. We are therefore, in collaboration with Durham, rewriting the necessary interfaces in ExaHyPE to use Peano4. Initially we have focused on the Euler equations and low-order discretisations. As Peano4 develops further, we will enable more of ExaHyPE's features. While we do not see a speedup for the current sweet spots of the code by moving to Peano4, we see a considerable improvement away from the sweet spots.

Multiple runs for UQ ExaHyPE features a multi-solver interface that can be used to combine multiple solves on overlapping or non-overlapping meshes, also including different discretisation orders or even different PDEs. We will implement a multi-solver variant of our Multi-Level Markov-Chain Monte Carlo (MLMCMC) algorithm and intend to use ExaHyPE's multi-solver feature to improve overall runtime. The multi-solver interface will allow us to solve on every level

simultaneously. In very small test setups, we see that ExaHyPE's performance has reached the strong-scaling limit, by grouping multiple runs into one, we can avoid this and gain a speedup. The MLMCMC tests in ExaHyPE are in very early stages and considerable performance improvements are expected.

Prototyping low and mixed precision in ExaHyPE For inverse problems and UQ scenarios, it is often sufficient to execute entire simulations or at least certain performance-critical kernels in lower precision. We therefore examined the potential of exploiting low or mixed precision in ExaHyPE. We prototypically implemented a mixed-precision ADER-DG scheme that computes the element-local space-time predictor in single precision, but keeps double precision for the remaining calculation. We observed speedups of up to 60% for the space-time-predictor kernel, which for the not fully compute-bound elastic wave propagation scheme improved time-to-solution by roughly 30%. However, we also found that extending the prototype to a proper integration of mixed precision in the engine would require extensive changes in the ExaHyPE Toolkit and Kernel Generator, which is not feasible with the resources available in ENERXICO. Massive work would particularly be required in the algorithm layer, which is hard-coded in double precision and implements the ADER-DG scheme on the tree-structured adaptive grids provided by the Peano framework. Similar effort would be required for allowing to switch the general target precision between single and double precision.

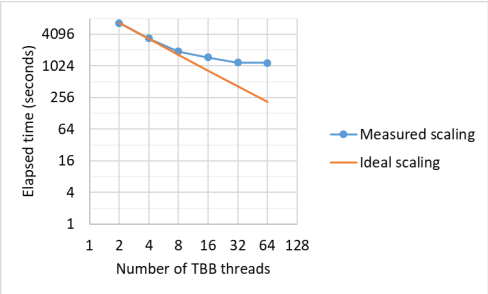
Testing GEMM backends in ExaHyPE Many of ExaHyPE's element-local kernels are tensor operations, which can be computed as sequences of loops over matrix multiplications ("loop over gemm"). For best-possible performance of these small, fixed-size matrix multiplications, we rely on the LIBXSMM library [8] on Intel architectures. We updated the ExaHyPE Kernel Generator to also use the Eigen library (`http://eigen.tuxfamily.org/`) as matrix multiplication backend, which has previously shown promising results with SeisSol on ARM architectures. However, tests on the Intel Haswell architecture revealed that Eigen did not lead to performance advantages compared to a generic loop-based implementations. We will evaluate the issue more closely also on other architectures, but assume that the comparably small matrix sizes for the seismic use cases in ExaHyPE are already efficiently captured by a generic loop-based implementation. Still, the integration of the Eigen library has prepared the Kernel Generator for including other small-GEMM backends, which might be specialised for architecture like ARM, RISC V or similar.

Testing of ExaHyPE on the AMD Zen2 architecture Tests on the AMD zen2 architecture have been performed using the ExaHyPE version which implements several types of GEMM backends. The "generic" GEMM backend uses non-optimised kernels. The"optimised" GEMM backend used the vectorised XSMM library. the latter is compiled targeting the AVX2 instruction set. The "optimised" backend can be combined with two other features: "split_ck" which allows a better utilisation of the cache and "vectorise_terms" which enables aggressive vectorisation.

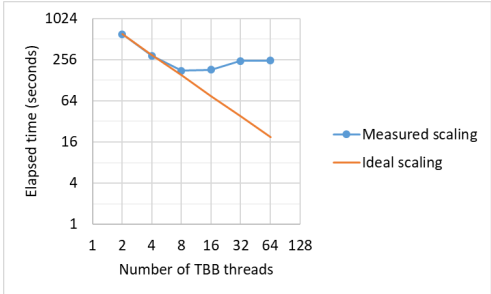
The test case used is a high order discontinuous Galerkin solver for elastic wave equation using a curvilinear mesh as it is described in appendix A.1. We simulate 100 time steps of the benchmark and measure MPI and TBB strong scaling. Following the recommendations of the POP audit, order 7 is used as because of its best performance.

In Figure 9, the strong scaling analysis is performed for both shared memory parallelisation though TBB and distributed memory with MPI. Peano's geometrical domain-decomposition has load balancing sweet-spots at 2, 28 and 731 ranks. The best scaling is obtained via hybrid

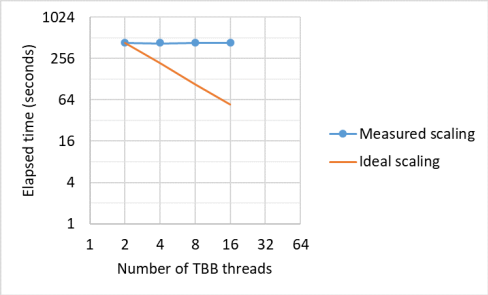
(MPI+TBB) parallelism by exploit MPI sweet spots and increasing the number of TBB threads per MPI rank.



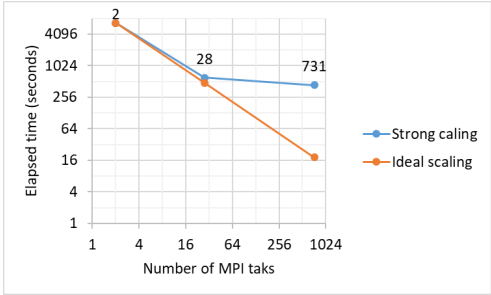
(a) TBB strong scaling with 2 MPI tasks



(b) TBB strong scaling with 28 MPI tasks



(c) TBB strong scaling with 731 MPI tasks



(d) MPI strong scaling using 2 TBB threads

Figure 9: TBB and MPI strong scaling curves for ExaHyPE. The optimised LIBXSMM backend is used. The elapsed time is plotted using a logarithmic scale.

In Figure 9a, Figure 9b and Figure 9c, the number of TBB threads is increased starting from 2 to 64 TBB threads for each MPI sweet sport 2, 28 and 731. In figure Figure 9d, an MPI strong scaling is presented. In the latter, two TBB threads are used.

These results are in agreement with previous audits performed on ExaHyPE on Intel Skylake processors.

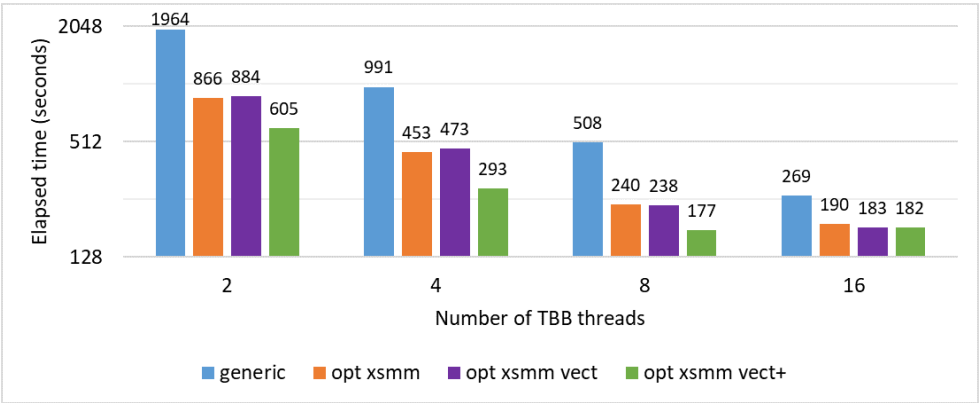


Figure 10: Performance comparison for generic and several level of optimisation in ExaHyPE. The tests are performed using 28 MPI tasks and the number of TBB threads varies from 2 to 16. The elapsed time is plotted using a logarithmic scale.

ExaHyPE shows great performance when using optimised kernels. The shared memory scalability is however limited when the number of threads reaches 16.

6.3. Summary and Outlook

The ExaHyPE application has been successfully ported to the new AMD Rome architecture. The use of different backends, namely generic and several levels of optimised backend using LIBXSMM, has proven to be a good design decision as it is easy to switch to a new architecture. All further activities, especially the port to Peano 4, will be continued in the remaining project period, so we refer to the next Deliverable 1.3.

7. SeisSol

SeisSol is a software package for simulating seismic wave propagation and earthquake dynamics based on the discontinuous Galerkin method with arbitrary high-order derivative time-stepping (ADER-DG). Characteristics of the SeisSol simulation software are:

- use of arbitrarily high approximation order in time and space (ADER-DG with Godunov flux formulation) with the option for cluster-wise local time stepping.
- use of tetrahedral meshes to approximate complex 3D model geometries (faults & topography) for rapid model generation.
- use of elastic, anisotropic, viscoelastic and viscoplastic material to approximate realistic geological subsurface properties.
- parallel geo-information input (ASAGI).

Goals in ENERXICO SeisSol is a flagship code of the European Centre of Excellence ChEESE (`www.cheese-coe.eu`), which hosts the majority of activities to advance performance and scalability of SeisSol towards exascale (including particularly a port to GPU architectures). In ENERXICO, we therefore put a stronger focus on enabling novel use cases of SeisSol in the context of WP3 (Oil and Gas), which contains the extension towards further material models (anisotropic and poroelastic materials) and towards inverse problems. Additional activities towards "exascale enabling" (i.e., WP1) focused on collaboration opportunities within the consortium.

The following table summarizes the effort (in person months, PM) spent on WP1 until Month 16 of the project:

Partner	PM spent
TUM	3
Bull/ATOS	4

7.1. Performance and Scalability Assessment

SeisSol scales up to beyond-petascale supercomputers: extreme-scale simulations have been performed on several of the world's largest supercomputers in the past, for example on Tianhe-2 (8000 nodes accelerated by Intel's Xeon Phi co-processor; achieved 8.6 PetaFlop/s in double precision [7]) or on Cori (6144 nodes Xeon Phi "Knights Landing" CPUs; $\approx$ 6 PetaFlop/s in double precision, follow-up work to [16]).

In terms of single-core performance, the optimisation of SeisSol has been strongly oriented towards Intel architectures (Knights Corner/Landing; recently Skylake). The MPI+OpenMP strategy strived to rely on a single MPI rank per node to reduce the stress on MPI parallelism. A key algorithmic challenge to scalability is SeisSol's cluster-wise local time stepping, for which elements with a similar time step bound[1] are merged into cluster's, which are propagated in a multi-rate fashion. In the strong-scaling limit, small sizes of such clusters impede shared- and distributed-memory scalability. However, local time stepping is crucial for many scenarios that feature strong adaptive mesh refinement and/or highly complicated geometries and typically leads to strongly improved time-to-solution compared to global time stepping, despite losses in parallel efficiency.

[1] from the Courant-Friedrich-Levy condition

As part of the project, we had registered for an "extreme scaling workshop" at Leibniz Supercomputing Centre, for which full-machine runs on the SuperMUC-NG supercomputer (Intel Skylake CPUs, 26.8 PetaFlop/s theoretical peak) were envisaged. This workshop has been postponed to October 2020 due to the COVID-19 outbreak.

7.2. Improvements to Performance and Scalability

7.2.1. Optimise SeisSol for non-Intel architectures

Since AMD architectures are becoming more important (see, e.g., recent installations such as Hawk at HLRS Stuttgart, Mahti at CSC Finland or El Capitan at Lawrence Livermore), we evaluated SeisSol on these architectures.

Hardware description The technical analysis and experiments made for the application SeisSol, were executed by the applicative experts from the Atos Center of Excellence for Performance Programming (CEPP) team. The SeisSol application has been run on ATOS on-premise supercomputers. The specifics of these supercomputers are described in Figure 11.

Atos Supercomputers				
		Genji		Spartan CPU
Processor	Processor SKU	AMD® EPYC® 7702	Intel® Xeon® Gold 6148	AMD® EPYC® 7742
	Processor class	Rome	Skylake-SP	Rome
	Fab Process	7 nm	14 nm	7 nm
	TDP	200 W	150 W	225 W
	Core Frequency (nominal)	2.0	2.4	2.25
	Cores per socket	64	20	64
	Processors per node	2	2	2
	Max. instruction set supported	AVX-2	AVX-512	AVX-2
	DP FLOP/cycle	16	32	16
	DP GFLOPS/node (theoretical)	4096	3072	4608
Memory	Memory channels per socket	8	6	8
	Memory DIMMs	DDR4 3200 MT/s	DDR4 3200 MT/s	DDR4 3200 MT/s
	Memory configuration (per node)	8x 16GB (DR)	187 GB	256 GB
Interconnect	High Speed Interconnect	Mellanox ConnectX-5	Mellanox ConnectX-5	Mellanox ConnectX-5
	Max Speed	4x EDR (100 Gbps)	4x EDR (100 Gbps)	4x EDR (100 Gbps)
	Topology	Fat-Tree (Clos Network)	Fat-Tree (Clos Network)	Fat-Tree (Clos Network)
	Oversubscription ratio (typical)	1:2	1:2	1:2

Figure 11: Description of ATOS supercomputers.

Software environment and test case description The SeisSol version used is the latest development version available in the master branch on the SeisSol Github. We used the community benchmark SCEC TPV13 [6] to compare different setups. The mesh consisted of 52 million elements. Computations have been run with convergence order 6. This test case describes a

spontaneous rupture on a 60-degree dipping normal fault in a homogeneous half-space (see Figure 12). A further detailed description of the test case is given in the reference document of

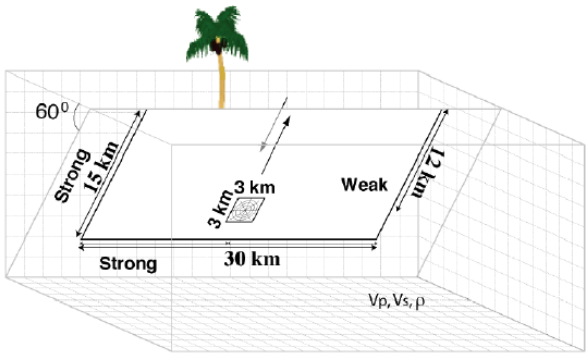


Figure 12: Diagram of the geometry of TPV13. 60-degree dipping normal fault.

SeisSol (`https://seissol.readthedocs.io/en/latest/tpv13.html`). An in-detail description of how we compiled and run SeisSol can be found in appendix A.2.

Testing backends for Matrix Matrix multiplications (GEMM) SeisSol uses the code generator YATeTo [15] for high-performance implementation of element-local operations. YATeTo allows expressing element-local operations on small tensors via a DSL, and maps these tensor operations to backends which perform Generalised Matrix Matrix multiplications (GEMM). For Haswell we use libxsmm (`https://github.com/hfp/libxsmm`) and for Skylake a combination of libxsmm and PSpaMM (`https://github.com/peterwauligmann/PSpaMM`).

We evaluated several code generator libraries for their suitability for AMD architectures.

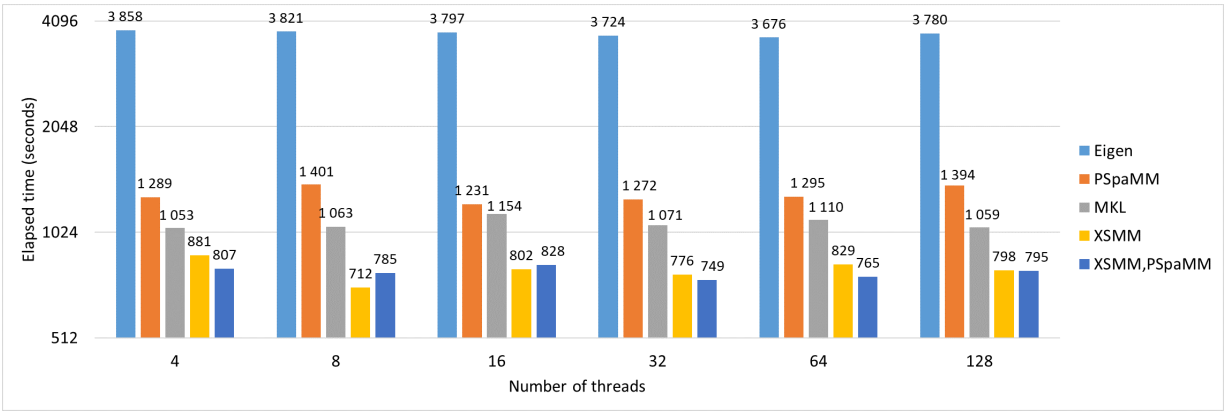
As a target platform we used the cluster Spartan, which is located at ATOS. The AMD section of the cluster consists of 224 nodes each equipped with two AMD Rome EPYC 7742 with 128 cores/node as it is described in Figure 11.

The AMD Rome processors implement the Zen2 microarchitecture, which is similar to Intel's Haswell architecture. Hence the code generator libxsmm is a natural candidate for the Zen2 architecture. In addition we used the linear algebra library Eigen3 (`eigen.tuxfamily.org/`) as backend and Intel's MKL (`https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html`).

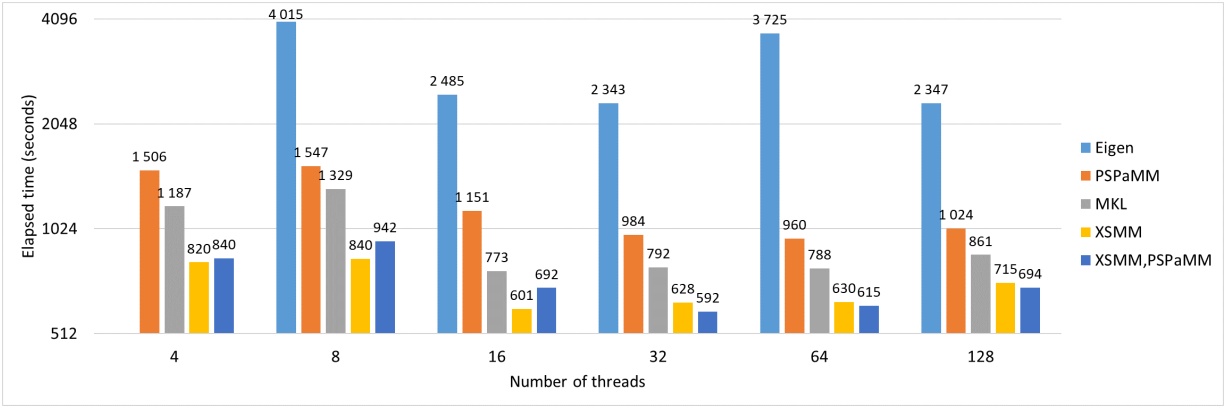
Considering the backends for small matrix matrix multiplications, we identify libxsmm to be superior over the other candidates. Although libxsmm targets Intel architectures, we see that libxsmm configured for Haswell achieves good intra node performance on AMD Zen2 architecture. As PSpaMM makes use of AVX-512 instructions, it is not beneficial to combine it with libxsmm on the AMD Zen2 microarchitecture. Intel's MKL library is targeted at Intel architectures, but can also be compiled for AMD targeting AVX2 instruction set. It is known that Intel MKL performs bad on non Intel architectures. We just included it for the sake of completeness. Although we had considerable success with Eigen3 on ARM architectures, it did not live up to its expectations on AMD.

In Figure 13, for each backend, all 128 cores of one node are used. The number of MPI tasks and OMP threads vary such that all cores are used. For each run, 64 nodes are used.

Remark. ***Time metric:*** *The relevant execution time is mentioned in the log file at the line containing the string "Elapsed time (via clock_gettime):".*



(a) Communication thread enabled



(b) Communication thread disabled

Figure 13: Comparison of the Eigen, MKL, XSMM, PSPaMM and XSMM,PSpaMM backends on 64 nodes runs. Labels on top of the bars denote the elapsed time in seconds.

7.2.2. Evaluation of NUMA-aware MPI+OpenMP parallelisation

The MPI+OpenMP parallelisation has so far strongly focused on a "one MPI rank per node" strategy, as this had clear advantages for the Intel Knights Corner/Landing manycore architectures (which did not show strong NUMA effects). Also the high arithmetic intensity of SeisSol's ADER-DG has not led to considerable NUMA effects on Intel's Haswell architecture (on SuperMUC, e.g.). In ENERXICO, we reexamined NUMA effects on recent architectures (esp. including AMD) and found stronger NUMA effects, which is partly due to architectures with a more pronounced NUMA architecture and due to a general shift of requiring more and more flops per byte to stay compute-bound.

Evaluation of NUMA effects on the AMD Zen2 architecture In Figure 13, we see a comparison of the overall runtime for different setups. With the communication thread enabled, we find an optimal runtime at 16 MPI ranks per node and 8 OMP thread per MPI rank and libxsmm as a backend. This can be explained by the strong NUMA effects we find on the Zen2 microarchitecture. The node topology of the Spartan cluster is given in Table 8. For flux computations, SeisSol needs to access data for a cell and all its neighbours. As cells are not locally ordered this requires memory access through across NUMA domain boundaries.

node	0	1	2	3	4	5	6	7
0	10	12	12	12	32	32	32	32
1	12	10	12	12	32	32	32	32
2	12	12	10	12	32	32	32	32
3	12	12	12	10	32	32	32	32
4	32	32	32	32	10	12	12	12
5	32	32	32	32	12	10	12	12
6	32	32	32	32	12	12	10	12
7	32	32	32	32	12	12	12	10

Table 8: Distances between the different NUMA domains as given by `numactl -H` on the Spartan Cluster.

Although in past studies a communication thread has been proven to achieve better results than a communication pattern where communication and computation are intertwined, we see a different behaviour here. The best setup is found with four MPI tasks per node, libxsmm and PSpaMM combined and the communication thread disabled. This can be explained as follows: When we use one communication thread per node we just dedicate about 0.8% of the computational resources to communication. Due to the NUMA effects, we have to use more MPI tasks per node. But having more MPI tasks per node, each equipped with one communication thread by themselves, means we lose more processors to communication.

Consequently for production runs, where also I/O plays a bigger role, there is now a higher need to tune the number of MPI tasks per node and evaluate whether the communication thread is beneficial or not.

Comparing the baseline (Communication thread enabled, libxsmm as backend and one rank per node) with the new found optimum (Communication thread disabled, libxsmm, PSpaMM as backend and 4 MPI tasks per node) we achieve a speedup of $\approx 26\%$.

Evaluation of NUMA effects on the Intel Skylake architecture After such promising results on AMD, we redid the same experiment on SuperMUC-NG. Here we ran the simulation on 792 compute nodes, which is one eighth of the total of SuperMUC-NG. Each node of SuperMUC-NG is equipped with two Intel Xeon Platinum 8174 processors and 96 GB RAM. The interconnect is an Intel Omnipath at 100 GB/s.

In Figure 14, we see that we also benefit from several MPI ranks per compute node. Here the communication thread still plays out its benefits as with two NUMA domains we do not loose too much computational power if dedicate one thread per MPI rank to communication. Comparing the baseline (commthread enabled, one rank per node) and the optimal configuration (commthread enabled, 4 ranks per node) we find a speedup of $\approx 18\%$.

node	0	1
0	10	21
1	21	10

Table 9: Distances between the different NUMA domains as given by `numactl -H` on SuperMUC-NG.

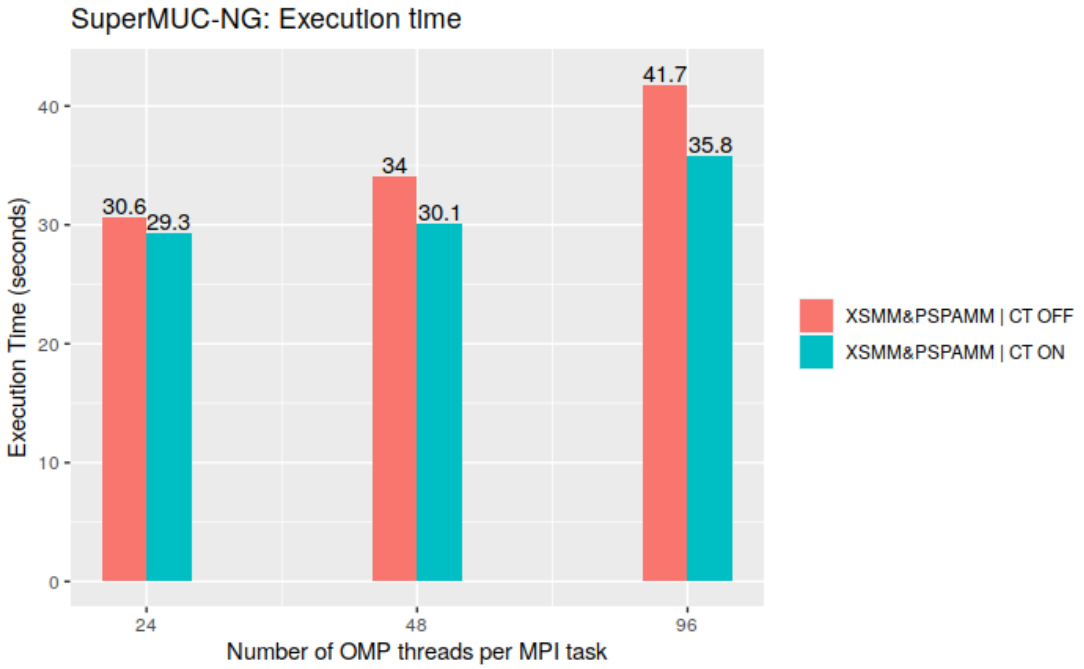


Figure 14: Comparison of different hybrid parallelisation schemes on 792 nodes of SuperMUC-NG.

Connection to WP3 In WP3 a code comparison study is ongoing. The aim of this study is to compare different codes for seismic wave propagation in terms of accuracy and performance. The results from the NUMA optimisation will be used for the production runs of the code comparison.

7.2.3. Evaluation of the computation performance on AMD Zen2 architecture

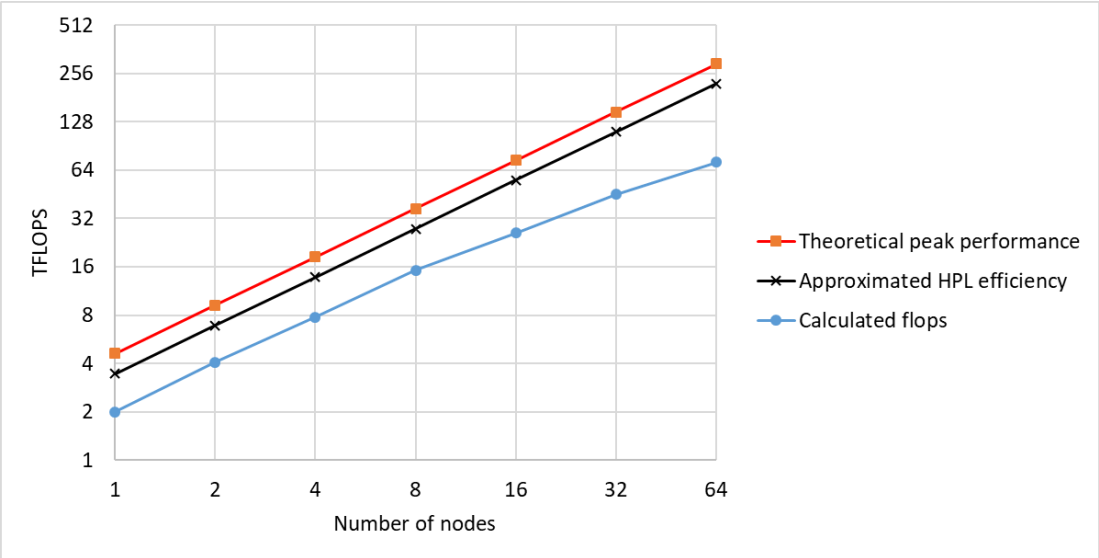


Figure 15: SeisSol calculated flops.

In Figure 15, SeisSol calculated flops are plotted with respect to the theoretical peak performance and an approximated 75% HPL efficiency. The XSMM,PSpaMM backend is used. Runs are performed using 16 OMP threads per MPI task. The number of nodes ranges from 1 to 64. In Figure 16 we see a scaling plot of SeisSol on the Spartan cluster from 1 to 64 nodes. We observe that SeisSol achieves around 50% of the HPL efficiency for low number of nodes. For the 64 nodes we still achieve around 30% of the HPL performance.

Using an in-house MPI profiling tool, an MPI analysis of the application SeisSol shows that there is an average of 12% ratio of the total parallel time spent in communication and MPI I/O. This data partly explains the difference between the calculated flops and the HPL efficiency seen in Figure 15.

Remark. ***Flop metrics*** *SeisSol outputs the hardware flops which are the flops actually calculated by the CPU. So the HW-GFLOP can be seen as machine utilisation. This number is extracted from the output at the line containing the string "Total calculated HW-GFLOP". The theoretical peak performance is calculated using the formulae described in appendix A.3.*

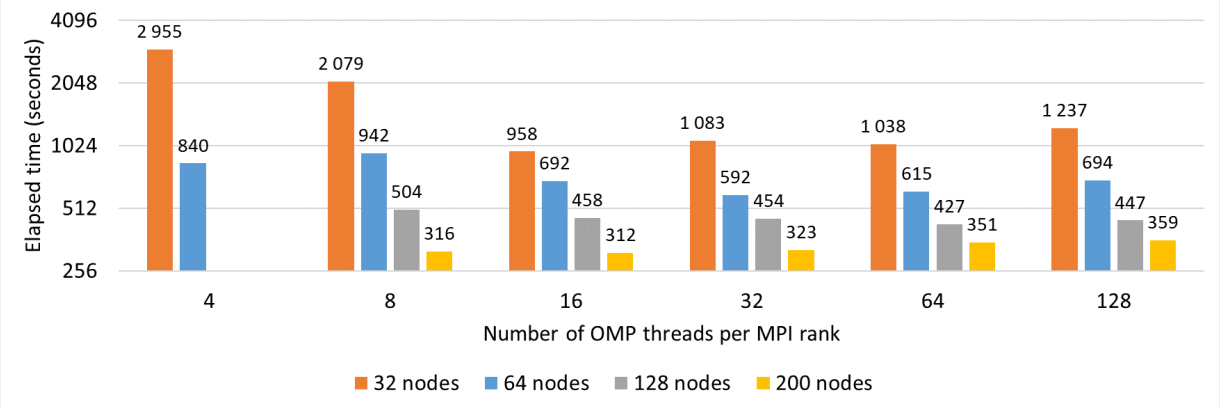


Figure 16: MPI scalability of SeisSol with libxsmm, PSpaMM backend and communication thread disabled. The number of nodes ranges from 32 to 200.

In Figure 16, we see a scaling plot of SeisSol on the Spartan cluster from 32 to 200 nodes. We observe that SeisSol achieves a good speedup up to 16 OMP threads per MPI rank.

7.2.4. Performance engineering for model extension

In the scope of work package 3, SeisSol has been extended to take fully anisotropic materials into account. The results have been published in [18]. As part of the research leading to this publication we conducted performance measurements, which are part of work package 1. The numerical scheme for elastic and anisotropic materials are similar, so we were able to achieve equally good results. With around 1 TeraFlop/s per node in double precision with local time stepping we achieve approximately a quarter of the theoretical per-node peak performance. Using global time stepping we achieve a higher performance at the cost of increased time to solution.

7.3. Summary and Outlook

We have successfully ported SeisSol to the new AMD Rome architecture. The use of different backends to YATeTo has proven to be a good design decision as we are able to easily switch to

a new architecture. As a follow-up to the tests on AMD we redid the same tests on Skylake and also found considerable performance improvements due to a NUMA aware parallelisation strategy.

In the upcoming period of the project we will focus on the model extension to poroelastic materials.

8. SEM46

SEM46 is a 3D seismic modelling and inversion code, developed mainly in the frame of the SEISCOPE project (`https://seiscope2.osug.fr`) for tackling modelling and full waveform inversion topics from the near surface to the deep crustal scale. The modelling kernel is based on spectral elements designed on Cartesian-based hexahedral meshes. The code implements in its current version elastic and visco-elastic equations for both modelling and inversion tasks. The inversion part is coupled with the non-linear Optimisation toolbox of SEISCOPE (`https://seiscope2.osug.fr/SEISCOPE-OPTIMIZATION-TOOLBOX`) in order to implement efficient large-scale non-linear optimisation schemes.

The following table summarises the effort (in person months, PM) spent on WP1 until Month 16 of the project:

Partner	PM spent
Bull/ATOS	2

8.1. Performance and Scalability Assessment

The performance and scalability of SEM46 have been evaluated on different clusters prior the ENERXICO project. Figure 17 shows strong-scaling tests done on different intel-based clusters for cubic-shaped targets.

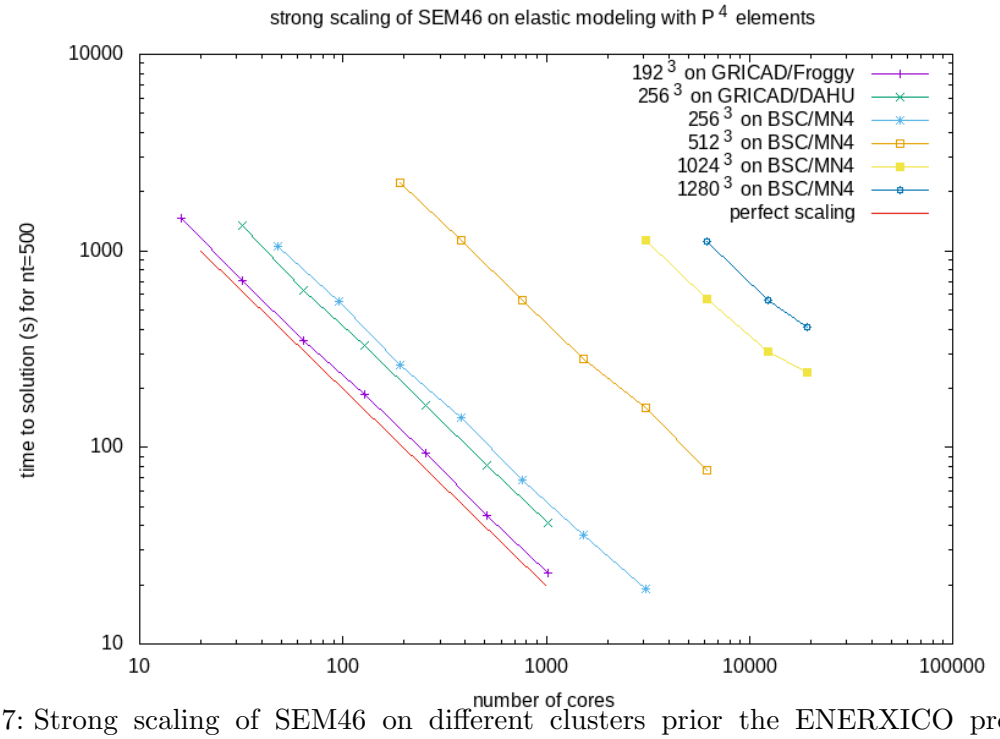


Figure 17: Strong scaling of SEM46 on different clusters prior the ENERXICO project, on cubic reference test-case for the modelling part. Froggy/GRICAD is a Sandy-Bridge Intel-based cluster with Infiniband network hosted at Univ. Grenoble Alpes. Dahu/GRICAD is a Skylake Intel-based cluster with OmniPath network hosted at Univ. Grenoble Alpes. MN4 (MareNostrum4) is a Skylake Intel-based cluster with OmniPath network hosted at BSC

The performance and scalability of SEM46 have also been assessed through a POP audit in November 2019, on the MareNostrum4 cluster, with the design of 4 realistic test-cases: one small geometry target running on 48 cores with one forward and one inversion test-cases; one big geometry target with one forward problem running on 96 cores and one inversion problem running on 192 cores.

This audit highlighted:

- very good parallel efficiency (larger that 94.4% for all test cases), and small unbalanced due to work distribution
- relatively low IPC that should be looked at more carefully
- identification of useless MPI_Barrier for the inversion kernel

8.2. Improvements to Performance and Scalability

The improvements of SEM46 have been focused on the test-cases designed for the POP Audit. These test-cases specifications are detailed below

Test case name	code mode	physics	mesh size
small forward	forward modeling P_4 elements	elastic isotropic	$54 \times 168 \times 52$
small inversion	gradient building P_4 elements	elastic isotropic	$54 \times 168 \times 52$
big forward	forward modeling P_4 elements	elastic isotropic	$108 \times 336 \times 104$
big inversion	gradient building P_4 elements	elastic isotropic	$108 \times 336 \times 104$

Analysis Using Intel VTune Amplifier, one can see in Table 10 that the two main most time consuming regions of the code or "hotspots" are the functions called "stiffness_vector_product_deville_p4" and "update_newmark1sponge".

Function / Call Stack	Effective time	MPI busy wait time	Source file
stiffness_vector_product_deville_p4_cart	66.5%	0 s	stiffness_vector_product_deville_P4.f90
update_newmark1sponge	14.30%	0 s	update_newmark.f90
update_newmark2	8.0%	0 s	update_newmark.f90
apply_mass_matrix	4.8%	0 s	apply_mass_matrix.f90
communicate	0.9%	0 s	communicate.f90
abc_clayton_down	0.7%	0 s	abc_clayton.f90
__intel_avx_rep_memcpy	0.1%	0 s	[Unknown]
pmpi_sendrecv_	0.1%	14.9 s	sendrecvf.c
pmpi_wtime_	0.0%	0 s	wtimef.c
abc_clayton_east	0.0%	0 s	abc_clayton.f90
abc_clayton_weast	0.0%	0 s	abc_clayton.f90
abc_clayton_north	0.0%	0 s	abc_clayton.f90
abc_clayton_south	0.0%	0 s	abc_clayton.f90

Table 10: SEM46 hotspots analysis: most active function in the application

Stiffness Kernel optimisations We have analysed the main kernel, namely stiffness_vector_product_deville_p4, using multiple methods:

- using Intel Advisor, we can see that vectorisation level is rather well vectorised (around 60% efficiency),
- Using a frequency stepping procedure, we can also see the performance dependency to CPU frequency and memory bandwidth. Through this analysis, we obtained an equal split between compute bound and memory bound, though this analysis was done on an Intel Skylake 6130 processor (16 cores with 2.1GHz). On higher-end processors we could think that the code would become more memory bound (the 6130 has a high memory bandwidth with regards to its computational power).

Looking at the compiler reports, we have noticed that although the code is vectorised, the loops are often very small, hence the overhead required to start the vectorised loop is significant compared to the actual vectorised computation. Here the reason is that most vectorised loops take advantage of the fact that each element in the computation has multiple integration points and vectorisation is carried out over those points. One different strategy would be to vectorise over multiple elements at the same time: do the same calculation for each point i on N different elements. This requires some code modification and loop transformations but allows to substantially increase the length of the vectorised loops, hence allowing to obtain a higher vectorisation efficiency.

Another set of modifications concerns an array of tensors called C_{ij}. This tensor is defined for each element, and has 21 components. However, when we look at the code, we can see that only 9 components are really used, but looking further we can see, from the source code generating this tensor in "vpvsrho2Cijkl.f90", that 5 components are simple copies of 4 others. Furthermore, reading the code in "vpvsrho2Cijkl.f90", we can see that all components of the "Cij" tensor can be generated from 2 scalar fields (vp and vs).

We can take advantage of these observations to produce multiple specialised kernels: these kernels will use assumptions based on these observations to reduce the memory accesses within the kernel, hence reducing the memory bound part of the kernel.

UpdateNewMark1Sponge Kernel optimisations The second major kernel "update_newmark1-sponge", is essentially memory bound. We have tried using streaming stores but we have not obtained any significant improvement.

Communication optimisations We have noted that the communication scheme is based on blocking send-receive calls. The problem with this type of communication scheme is that it enforces an order between mpi processes to solve the communications. In practice, it generates "waves" in the communication pattern for example, which is not optimal. We have implemented another version based on non-blocking communication calls (isend/irecv/waitall). However on these small test cases we have not measured any significant improvement and no performance loss neither. To measure improvement, we would require large scale tests.

Preliminary tests and numerical validation To validate our modifications, we have introduced an inline validation process: if a specific environment variable is set, the code will run the original kernel, the optimised kernel and will compute the differences between their outputs. The error is displayed as an L_2 relative residual.

We have reported in Table 11 the some of our initial optimisation results.

Version	Kernel time	Application time	Kernel improvement	Application improvement
Original	11.81	14.35	0%	0%
OPT1 : vectorisation other elements	10.26	12.85	13%	5%
OPT2: specialised kernels using C_{ij}	9.32	11.82	21%	24%

Table 11: SEM46 kernels optimisations results

We have further introduced similar optimisations for the backward step (kernel stiffness_vector_product_deville_P4_inversion_single_field). To enable or disable these optimisations, we have introduced four environment variables:

- Forward kernel (stiffness_vector_deville_p4_cart)
  - BULL_STIFF_FORWARD_OPT=X
    - $X = 0 \rightarrow$ Reference kernel (no optimisation) (default value)
    - $X = 1 \rightarrow$ Optimisation level 1 (gather elements, vectorisation of computations over multiple elements, scatter results, uses most generic C_{ij} tensor)
    - $X = 2 \rightarrow$ Optimisation level 1 + using the fact that only 4 components of the C_{ij} tensor are required,
    - $X = 3 \rightarrow$ Optimisation level 1 + using the fact that the required C_{ij} tensor components can be generated from two fields (V_P and V_S),
  - BULL_STIFF_FORWARD_CHECK=X
    - $X = 0 \rightarrow$ Numerical validation disabled(default value)
    - $X = 1 \rightarrow$ Numerical validation enabled: executes reference kernel and chosen optimised kernel, displays the L_2 relative residual.
- Backward kernel (stiffness_vector_product_deville_P4_inversion_single_field)
  - BULL_KU_DEVILLE_P4_CART_INVERSION_OPT=x
    - $X = 0 \rightarrow$ Reference kernel (no optimisation) (default value)
    - $X = 1 \rightarrow$ Optimisation level 1 (gather elements, vectorisation of computations over multiple elements, scatter results, uses most generic C_{ij} tensor)
    - $X = 2 \rightarrow$ Optimisation level 1 + using fact that only 4 components of the C_{ij} tensor are required,
    - $X = 3 \rightarrow$ Optimisation level 1 + using the fact that the required C_{ij} tensor components can be generated from two fields (V_P and V_S),
  - BULL_KU_DEVILLE_P4_CART_INVERSION_CHECK=x
    - $X = 0 \rightarrow$ Numerical validation disabled (default value)
    - $X = 1 \rightarrow$ Numerical validation enabled: executes reference kernel and chosen optimised kernel, displays the L_2 relative residual.

8.3. Summary and Outlook

SEM46 is a powerfull application which enables to perform seismic modelling and inversion using various approximations, particularly the isotropic and anisotropic approximations. Therefore, the work that has been carried out in WP1 of ENERXICO takes into account this flexibility of SEM46.

Considering the isotropic test case which has been used for the POP audit, basic optimisation issues were identified for this particular setup. The vectorisation enablings and an improved memory management allowed to achieve an 24% gain in the overall walltime.

Enabling and disabling each of these optimisations can be done using an environment variable setup to easily switch from one version of the code to another.

The next steps are to consider first the anisotropic approximation test case. The above optimisation principles shall be applied in this framework. Then, a large scale test case shall be used to measure the gain of the communication optimisations.

9. WRF

WRF is a mesoscale numerical weather prediction system designed for both atmospheric research and operational forecasting applications. It features two dynamical cores, a data assimilation system, and a software architecture supporting parallel computation and system extensibility. For researchers, WRF can produce simulations based on actual atmospheric conditions (i.e., from observations and analyses) or idealized conditions. WRF offers operational forecasting, a flexible and computationally efficient platform, while reflecting recent advances in physics, numeric, and data assimilation. It is continuously contributed by developers from the expansive research community.

Role of WRF in ENERXICO WRF is providing excellent results from the physics point of view to many research groups around the world. However, as its scalability is far from being ideal, it is not generally considered as a target code for exascale machines, even though many researchers perform WRF simulations nowadays on pre-exascale computers. WRF is of vital importance for ENERXICO's WP2, where it has produced remarkable results for the simulation of wind energy fields. This motivates to analyse and improve the scalability and performance of the the WRF version used for WP2 in ENERXICO. See section 9.2 for further details about the WRF modules used.

The following table summarises the effort (in person months, PM) spent on WP1 until Month 16 of the project:

Partner	PM spent
CIEMAT	5.4

9.1. Performance and scalability Assessment

Known performance/scalability bottlenecks Several tests on strong and weak scalability have been performed. This analysis has been carried out internally by the project team, though following the structure of any report issued by the POP CoE. In this sense, it extends this performance methodology in order to be uptaken by the scientific community.

The loss of efficiency observed in this report is due to the serialisation and temporal imbalances of the code that increase with the scale. The analysis identified different regions with temporal imbalances as well as how the imbalance of the largest computing region is absorbed by the point to point communications. The code has around 25000 calls to `MPI_Comm_rank()` per process on each iteration.

The global balance of the code is good and it does not increase with the scale in the range of MPI ranks analysed.

Traditional strong and weak scalability tests have been performed. Deeper analysis of the code performance is carried out with Paraver.

9.2. Improvements to Performance and Scalability

Improving communication patterns Reducing the communication calls per process is expected to increase the performance. Despite the individual overhead is small, it is an issue to overcome for the whole test execution.

Tests have been performed with two specific WRF modules designed and implemented by CIEMAT: PBL (YSU) and SL (Revised MM5 surface layer scheme). There is then good experience with them and how they work from both the computational and modelling point of view.

See e.g. [10–12]. Within ENERXICO, these modules have been used in WP2. Specifically, to identify the most relevant coupling parameters between meso- and micro-scale models for making a comparison with a high resolution climatology obtained with WRF.

Communication efficiency is the worst actor in a global efficiency analysis, presenting values of around 70% out of ideal 100%. We expect to enhance this behaviour around 5-10% by substituting some calls by local variables.

Results The efficiency analysis has allowed identifying the observed problems on the scalability, which are correlated with serialisation, transfer, and computational scalability.

The total number of instructions executed by the computations increases with the scale, which has suggested a code replication or increase of instructions due to the higher number of boundary cells when increasing the scale.

The computational environment description follows:

- 1 to 44 nodes PowerEdge C6420 with 2 Xeon Gold 6148 2,4GHz 40 cores/node
- RAM: 192 GB/node
- Infiniband EDR 100GB/s
- Storage: 28 x disks 12TB 7.2K NLSAS 12GB 3,5”

Looking at one iteration, the first part of the iteration corresponds to a region with a very large number of calls to `MPI_Comm_rank()` per process (in the whole iteration the number is around 25000). Despite the overhead to call the library to get the rank would be small, accessing to a local variable that stores the value is expected to eliminate all these library calls. This has been the strategy followed.

In comparison to our old implementation, we achieved a speedup of $\sim 8\%$ in average in the strong scalability (see Figure 18).

The loss of strong scalability is obviously not due to the I/O time since the size of the problem, and hence I/O time, is constant for all the points in the figure. The loss of efficiency comes from the increase in communication time since the number of points in each cell is less and less as the distribution in the number of cores increases. Communication is a mix of intra and extra node since the number of cores according to the number of cells in the horizontal dimensions have been balanced (as advised in the WRF user forum, this is much more efficient than filling nodes, unless the simulation was covering a perfect square) and does not always correspond to a multiple of the number of cores per node. WRF developers advise using a minimum of 50000 points per core; below that threshold there is a loss of efficiency due to communications. A test on weak scalability has been performed, the results of which are depicted in Figure 19. For carrying it out, ad hoc spaces were created, i.e. 1×10^5 and 2.5×10^5 points per cell. Results show a good agreement with a best ideal fit.

9.3. Summary and Outlook

Within the ENERXICO project we plan to conduct tests on energy consumption per code. It is the final aim to publish computational and energy efficiency results in a joint publication. Future work will be dedicated to carry out these energy tests and prepare the publication.

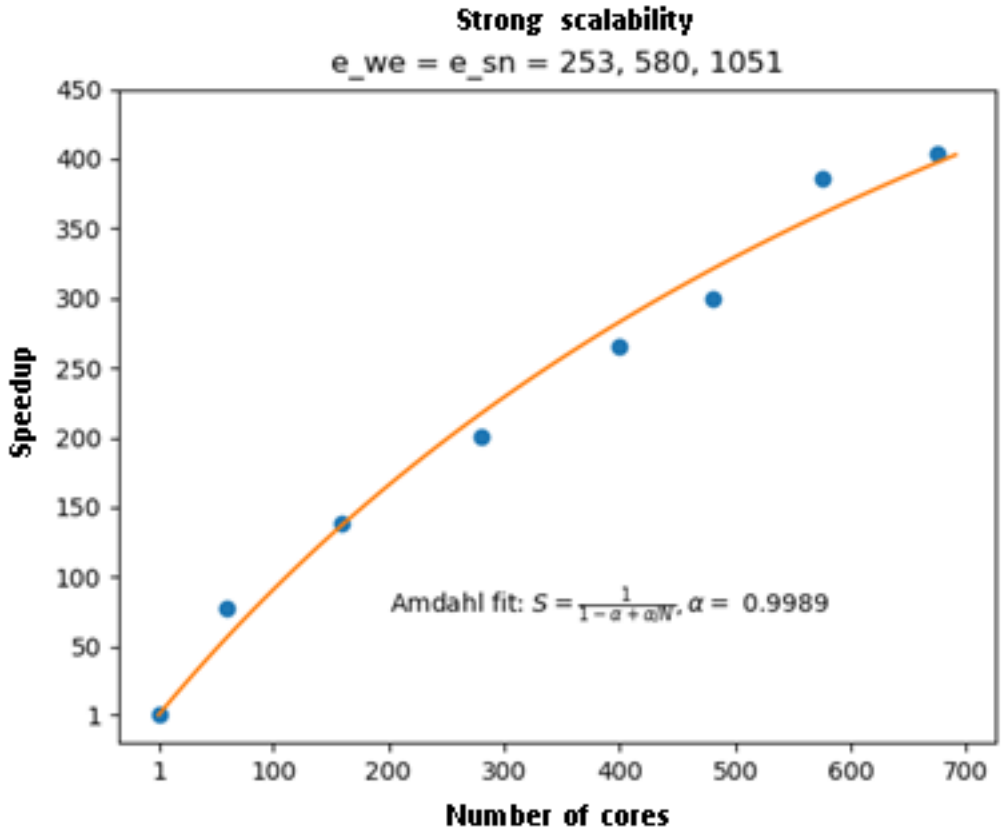


Figure 18: Strong scalability according to the Amdahl Law. An efficiency of 60% is obtained for the maximum number of cores (675). Results are an average of 5 measurements (error bars are below 3% and are not depicted for readability reasons)

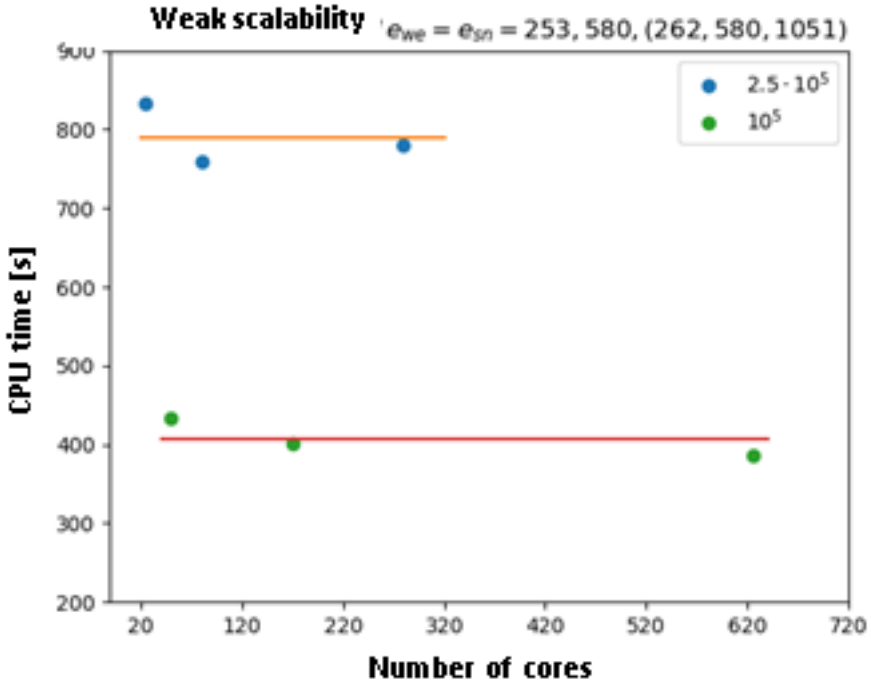


Figure 19: Weak scalability results for ad hoc spaces created; blue stands for $2.5 \cdot 105$ points per cell and green does for 105 points per cell. Best ideal fit is depicted with orange and red lines.

A. Appendix

A.1. ExaHyPE Setup for Zen2

The 3 dimensionnal domain is setup as follow

```
"computational_domain": {
    "dimension": 3,
    "time_steps": 100,
    "offset": [  -20.5, 0.0 , -20.5  ],
    "width" : [   94.0, 94.0,  47.0 ]
}
```

The solver is defined as follow

```
"solvers": [
{
    "type": "ADER-DG",
    "name": "ElasticWaveSolver",
    "order": 7,
    "maximum_mesh_size": 3.0,
    "maximum_mesh_depth": 0,
    "time_stepping": "global",
    "aderdg_kernel": {
        "language": "C",
        "nonlinear": false,
        "terms": ["ncp", "flux", "material_parameters", "point_sources"],
        "space_time_predictor": {"split_ck":true,"vectorise_terms":true},
        "optimised_terms": [],
        "optimised_kernel_debugging": [],
        "implementation": "optimised",
        "adjust_solution": "patchwise",
        "basis": "Lobatto"
    },
    "point_sources": 1,
    "variables": [
        { "name": "v"    , "multiplicity": 3 },
        { "name": "sigma","multiplicity": 6 }
    ]
    "material_parameters": [
        { "name": "rho","multiplicity": 1 },
        { "name": "cp" ,"multiplicity": 1 },
        { "name": "cs" ,"multiplicity": 1 },
        { "name": "jacobian","multiplicity": 1},
        { "name": "metric_derivative", "multiplicity": 9 },
        { "name": "curve_grid", "multiplicity": 3       }
    ],
    "parameters": {
```

```
        "scenario"          : "Loh1",
        "topography"        : "None"
    },
}
]
```

A.2. Compiling and Running SeisSol

To compile SeisSol, the latest version from the Github master branch is downloaded including all submodules (ImpalaJIT and yaml-cpp):

```
$ git clone https://github.com/SeisSol/SeisSol.git
$ git submodule update --init
```

The following software environment is used:

- Intel compiler 2020 update 2,
- Intel MPI 2020 update 2,
- CMake 3.10.0,
- Python 3.7 available in Anaconda 3.0,
- NumPy 1.18 available in Anaconda 3.0,
- HDF5 version 1.10.4,
- NetCDF version 4.6.1,
- ParMETIS version 4.0.3,
- LIBXSMM version release 1.15,
- PSpaMM available in the master branch on Github.

The submodules and dependencies are compiled as follow:

```
export CC=mpiicc
export CXX=mpiicpc
export FC=mpiifort

export PATH=${MYDIR}/bin:$PATH
export LD_LIBRARY_PATH=${MYDIR}/lib:$LD_LIBRARY_PATH
export LIBRARY_PATH=${MYDIR}/lib:$LIBRARY_PATH
export CPATH=${MYDIR}/include:$CPATH
export PKG_CONFIG_PATH=${MYDIR}/lib/pkgconfig:$PKG_CONFIG_PATH

# ImpalaJIT
cd submodules/ImpalaJIT
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=${MYDIR} -- ..
make -j16 install
cd ../../
```

```
# yaml-cpp
cd submodules/yaml-cpp
mkdir build && cd build
cmake -DCMAKE_INSTALL_PREFIX=${MYDIR} -DYAML_CPP_BUILD_TOOLS=OFF
 ↪  -DYAML_CPP_BUILD_TESTS=OFF -- ..
make -j16 install
cd ../../

# HDF5
cd hdf5-1.10.4
CPPFLAGS="-fPIC ${CPPFLAGS}" ./configure --enable-parallel --prefix=${MYDIR}
 ↪  --with-zlib --disable-shared --enable-fortran
make -j16
make install
cd ..

# netCDF
cd netcdf-4.6.1
CFLAGS="-fPIC ${CFLAGS}" CC=h5pcc ./configure --enable-shared=no
 ↪  --prefix=${MYDIR} --disable-dap
make -j16
make install
cd ..

# LIBXSMM
cd libxsmm
make generator
cp bin/libxsmm_gemm_generator ${MYDIR}/bin
cd ..

# PSpaMM
ln -s $(pwd)/PSpaMM/pspamm.py ${MYDIR}/bin

# ParMETIS
cd parmetis-4.0.3
#edit ./metis/include/metis.h IDXTYPEWIDTH to be 64 (default is 32).
make config cc=mpiicc cxx=mpiicpc prefix=${MYDIR}
make -j16 install
cp build/Linux-x86_64/libmetis/libmetis.a ${MYDIR}/lib
cp metis/include/metis.h ${MYDIR}/include
cd ..
```

Finally, SeisSol is compiled as follow:

```
export CC=mpiicc
export CXX=mpiicpc
```

```
export FC=mpiifort

export PATH=${MYDIR}/bin:$PATH
export LD_LIBRARY_PATH=${MYDIR}/lib:$LD_LIBRARY_PATH
export LIBRARY_PATH=${MYDIR}/lib:$LIBRARY_PATH
export CPATH=${MYDIR}/include:$CPATH
export PKG_CONFIG_PATH=${MYDIR}/lib/pkgconfig:$PKG_CONFIG_PATH

mkdir build && cd build
cmake -DNETCDF=ON -DMETIS=ON -DCOMMTHREAD=ON -DASAGI=OFF -DHDF5=ON
 ↪  -DCMAKE_BUILD_TYPE=Release -DTESTING=OFF  -DLOG_LEVEL=warning
 ↪  -DLOG_LEVEL_MASTER=info -DARCH=hsw -DPRECISION=double -DPLASTICITY=ON
 ↪  -DGEMM_TOOLS_LIST="LIBXSMM" ..
make -j16
```

For running the simulations, the parameter file "parameter_tpv13.par" is used after adjusting the length of the simulation "EndTime = 1.0" to have a reasonable execution time.

```
&equations
MaterialFileName = 'tpv12_13_material.yaml'
Plasticity = 1
Tv = 0.03
/

&IniCondition
/

&Boundaries
BC_fs = 1
BC_dr = 1                               ! Fault boundaries
BC_of = 1                               ! Absorbing boundaries
/

&DynamicRupture
FL = 16
ModelFileName = 'tpv12_13_fault.yaml'
inst_healing=0
GPwise = 1                  ! elementwise =0 ; GPwise =1
XRef = 0.0                  ! Reference point
YRef = -3.0e5
ZRef = -7.0e9
RF_output_on = 1            ! RF on
OutputPointType = 5         ! Type (0: no output, 1: take GP s 2: 4 points per
 ↪  surface triangle, 3: output at certain pickpoints)
t_0 = 0.0
/
```

```
&Elementwise
printIntervalCriterion = 2  ! 1=iteration, 2=time
printtimeinterval_sec = 0.5 ! Time interval at which output will be written
OutputMask = 1 1 1 0 1 1 1 1 1 0 0   ! output 1/ yes, 0/ no - position: 1/
 ↪  slip rate 2/ stress 3/ normal velocity 4/ in case of rate and state output
 ↪  friction and state variable
                              ! 5/ background values
refinement_strategy = 2
refinement = 1
/

&Pickpoint
printtimeinterval = 1       ! Index of printed info at timesteps
OutputMask = 1 1 1 0        ! output 1/ yes, 0/ no - position: 1/ slip rate 2/
 ↪  stress 3/ normal velocity 4/ in case of rate and state output friction and
 ↪  state variable      ! 5/ background values
nOutpoints = 10
PPFileName = 'tpv13_faultreceivers.dat'
/

&SourceType
/

&SpongeLayer
/

&MeshNml
MeshFile = './mesh/tpv12_13_200m'    ! Name of mesh file
meshgenerator = 'PUML'               ! Name of meshgenerator (format)
/

&Discretization
Material = 1                         ! Material order
CFL = 0.5                            ! CFL number (<=1.0)
FixTimeStep = 5                      ! Manualy chosen minimum time
ClusteredLTS=2                       ! This enables local time stepping
/

&Output
OutputFile ='./output13/tpv13'
iOutputMask = 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1      ! Variables ouptut
iOutputMaskMaterial = 1 1 1          ! Material output
Format = 6
Refinement =1                        ! Format (0=IDL, 1=TECPLOT, 2=IBM DX,
 ↪  4=GiD))
TimeInterval = 0.5                   ! Index of printed info at time
printIntervalCriterion = 2           ! Criterion for index of printed info:
 ↪  1=timesteps,2=time,3=timesteps+time
```

```
SurfaceOutput = 0
SurfaceOutputRefinement = 1
SurfaceOutputInterval = 0.05
pickdt = 0.005                          ! Pickpoint Sampling
pickDtType = 1                          ! Pickpoint Type
FaultOutputFlag = 1                     ! DR output (add this line only if DR is
 ↪  active)
nRecordPoints = 12                      ! number of Record points which are read
 ↪  from file
RFileName = 'tpv13_receivers.dat'     ! Record Points in extra file
/

&AbortCriteria
EndTime = 1.0                          ! original value 8.0
/

&Analysis
/

&Debugging
/
```

Finally, launching SeisSol is done as follow using SLURM:

```
#!/bin/bash

##############################################################################
# SLURM Header
##############################################################################

#SBATCH -J SeisSol
#SBATCH -N 1
#SBATCH -p ROME-7742_hdr100_256gb_3200
#SBATCH --time=0:15:00
#SBATCH --exclusive


##############################################################################
# Source application environment
source /opt/intel/compilers_and_libraries_2020.2.254/linux/bin/compilervars.sh
 ↪  intel64

export SEISDIR=${HOME}/software/intelimpi2020u2
export PATH=${SEISDIR}/bin:$PATH
export LIBRARY_PATH=${SEISDIR}/lib:$LIBRARY_PATH
export LD_LIBRARY_PATH=${SEISDIR}/lib:$LD_LIBRARY_PATH
export PKG_CONFIG_PATH=${SEISDIR}/lib/pkgconfig:$PKG_CONFIG_PATH
```

```
export CPATH=${SEISDIR}/include:$CPATH

###############################################################################
# Setup OpenMP
export OMP_NUM_THREADS=16
export KMP_AFFINITY=granularity=core,compact

###############################################################################
# Setup MPI
export I_MPI_PIN=1
export I_MPI_PIN_DOMAIN=${OMP_NUM_THREADS}:platform
export I_MPI_PIN_ORDER=compact

###############################################################################
# Hostfile
nodeset -e $SLURM_NODELIST | tr ' ' '\n' > ./hostfile

###############################################################################
# Run application
mpiexec.hydra -np 16 -ppn 16 -hostfile hostfile
 ↪  ./SeisSol_Release_dhsw_elastic_6 parameters_tpv12_13.par 2>&1 | tee
 ↪  output.log

###############################################################################
# Post processing
elapsedtime=$(grep 'Elapsed time (via clock_gettime):' output.log | awk
 ↪  '{print $10}')
hwgflop=$(grep 'Total calculated HW-GFLOP'         output.log | awk '{print
 ↪  $9}' )
echo ${elapsedtime} ${hwgflop}
```

A.3. Theoretical peak performance

To achieve good performance results on HPC hardware it is a key component to use the provided hardware as efficient as possible. Each architecture has its specific maximum number of FLOP/s, which can be achieved. FLOP/s are the measure of performance used for comparing the theoretical peak performance of a core, processor, node or a system by using floating point operations. The following formula is used. It can be used by any researcher to compute the maximum possible FLOP/s on their system.

$$FLOP/s_{core} = \frac{instructions}{cycle} * \frac{operations}{instruction} * \frac{FLOP/s}{operation} * \frac{cycles}{second},$$

and as processors are composed of many cores, hence

$$FLOP/s_{processor} = \frac{cores}{socket} * FLOP/s_{core},$$

and as modern nodes are often composed of several processors

$$FLOP/s_{node} = \frac{sockets}{node} * FLOP/s_{processor},$$

and finally, for full systems with many nodes, the above formula is extended to

$$FLOP/s_{system} = \frac{nodes}{system} * FLOP/s_{node}.$$

Remark. *Usually, the theoretical peak performance of a processor is compared to the measured one. The most common benchmark used is the HPL (High Performance Computing LINPACK Benchmark). It is a software package that solves a random dense linear system in double precision arithmetic on distributed memory computers. Therefore, HPL allows to measure the effective peak performance as opposed to the theoretical peak performance. The ratio between these two figures corresponds to the HPL efficiency. As a rule of thumb, the HPL efficiency of the nodes presented in the above table is between 70% and 90% depending on the number of nodes, the tuning of software environment and many other parameters.*

B. POP Audit for DualSPHysics

POP CoE
Dualphysics (POP2_AR050_)

Sandra Mendez (sandra.mendez@bsc.es) , BSC - March 04, 2020

EU H2020 Centre of Excellence (CoE)

1 December 2018 – 30 November 2021

Grant Agreement No 824080

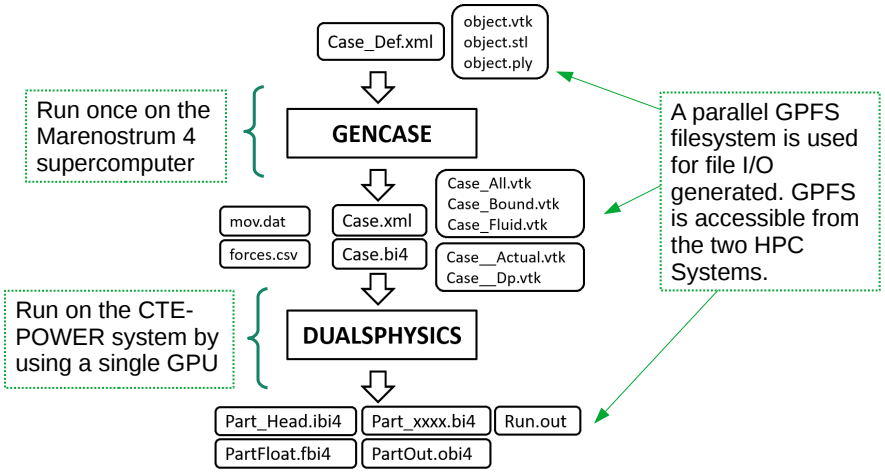
Background

Applicant: José Domínguez (Core developer) - EPhysLab (Universidade de Vigo)

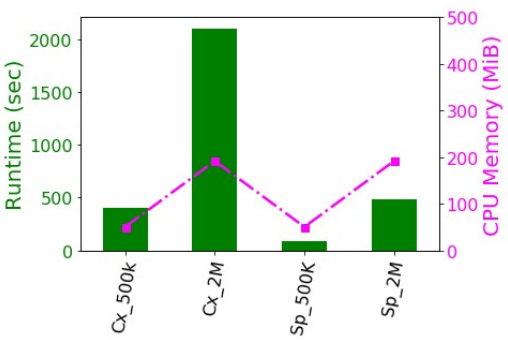
- Name of the code: DualSPHysics
- Scientific/technical area: Physic
- Programming: C++, CUDA
- Input case: 4 variants of the Dam Break case.
- Platform: CTE-POWER. 2 x IBM Power9 8335-GTH @ 2.4GHz (3.0GHz on turbo, 20 cores and 4 threads/core, total 160 threads per node). 512GB of main memory. 4 x GPU NVIDIA V100 (Volta) with 16GB HBM2. Software: cuda-9.1+gcc-6.4.0
- Scale: 500k and 2M particles
- Initial set-up: 500k (409088 GPU threads) and 2M (1759360 GPU threads) particles for simple and complex physic in Single GPU.
- BSC collected the performance data.

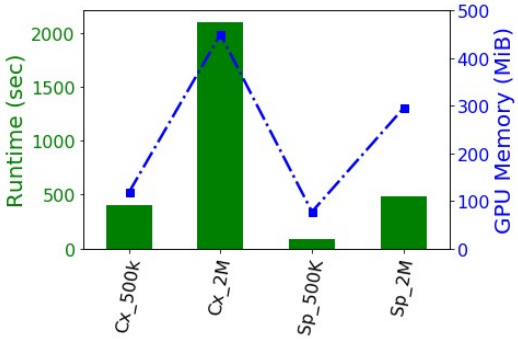
2

Application structure (1)



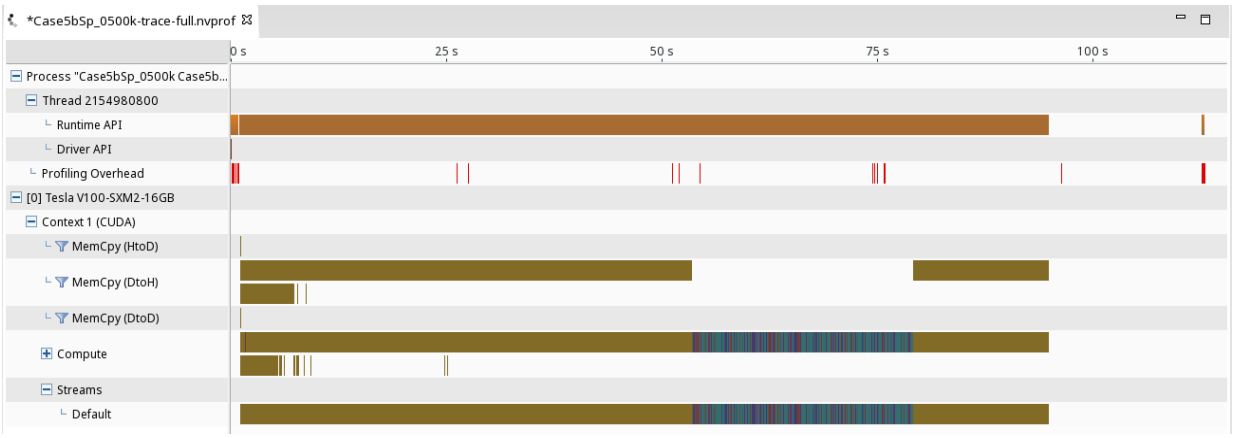
Initial Performance Evaluation





Runtime and utilized memory correspond to the output reported by the Dualsphysics part.

Initial profiling with nvprof



Default stream depicts the whole behavior for the parallel application. It can observed the kernels in blue scale and in brown scale the data copy between the host/device. Computation is representing less than 50% of the total runtime. The theoretical occupancy is near the 50%.

Profiling – kernel and CUDA API

Kernel with more timing in the total runtime: **KerInteractionForcesFluid**

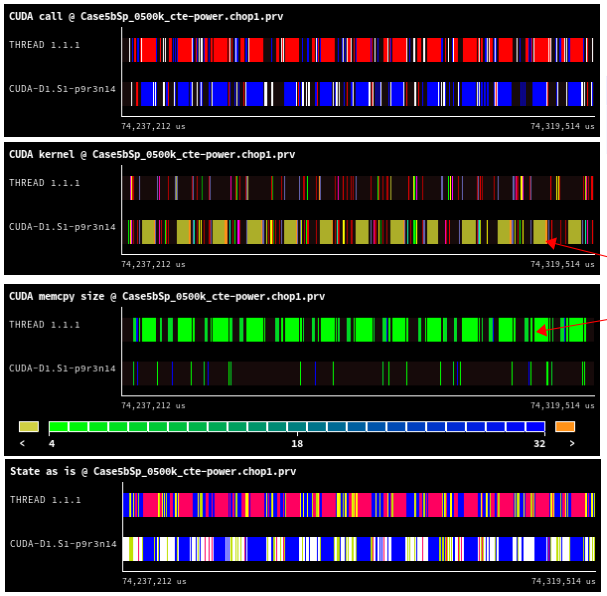
Case	Time(%)	Time	Calls	Avg	Min	Max
5bSp_0500k	75.53%	47.3957s	21436	2.2110ms	2.1255ms	2.5316ms
5bSp_2000k	83.14%	339.376s	35461	9.5704ms	9.2193ms	10.899ms
5bCx_0500k	79.60%	211.733s	64188	3.2986ms	3.1331ms	3.9406ms
5bCx_2000k	85.58%	1.6e+03s	111092	14.303ms	13.869ms	16.669ms

Simple physic timing represents a high percentage of runtime for the kernels that increase for 2M of particles. A similar behavior can be observed for the complex physics. The parallel degree at CUDA level is done by using 3196 grids and 128 blocks for 5k cases and 13745 grids and 128 blocks for 2M cases.

CUDA API with more timing in the total runtime: **memcpy**

Case	Time(%)	Time	Calls	Avg	Min	Max
5bSp_0500k	60.13%	53.9767s	128631	419.62us	18.424us	4.5602ms
5bSp_2000k	75.92%	361.109s	212781	1.6971ms	17.384us	22.510ms
5bCx_0500k	60.03%	235.000s	449330	523.00us	17.674us	110.32ms
5bCx_2000k	80.69%	1.7e+03s	777658	2.1681ms	17.222us	64.387ms

FOA: InteractionForcesFluid (Sp_500k)



GPU Utilization

kernel	Time(%)	Time	Calls	Avg	Min	Max
Interaction ForceFluid	75.53%	47.40s	21436	2.21ms	2.12ms	2.53ms

Grid Size	Block Size	Register per CUDA thread
(3196 1 1)	(128 1 1)	62

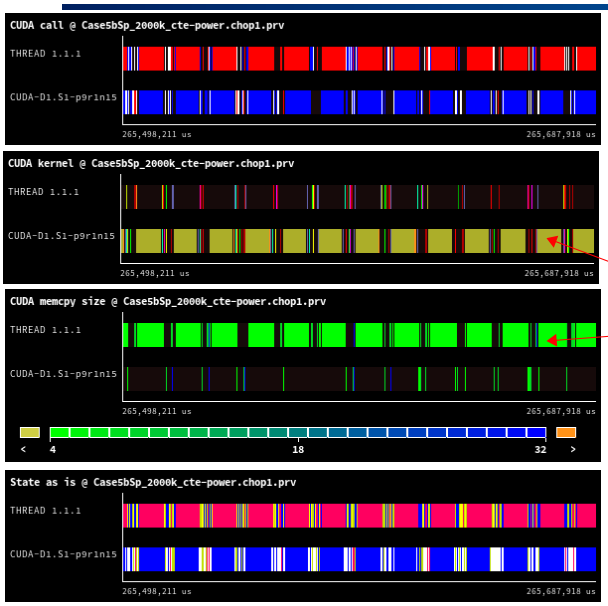
The kernel InteractionForcesFluid, its memcpy size is very small 4B.

API Calls

API	Time(%)	Time	Calls	Avg	Min	Max
cudaMemcpy	60.13%	53.98s	128631	419.62us	18.42us	4.56ms

Size memcpy (bytes)	Count	Total (bytes)
32B	21437	685984
4B	64308	257232
8B	42875	343000

FOA: InteractionForcesFluid (Sp_2M)



GPU Utilization

kernel	Time(%)	Time	Calls	Avg	Min	Max
Interaction ForceFluid	83.14%	339.4s	35461	9.57ms	9.22ms	10.9ms

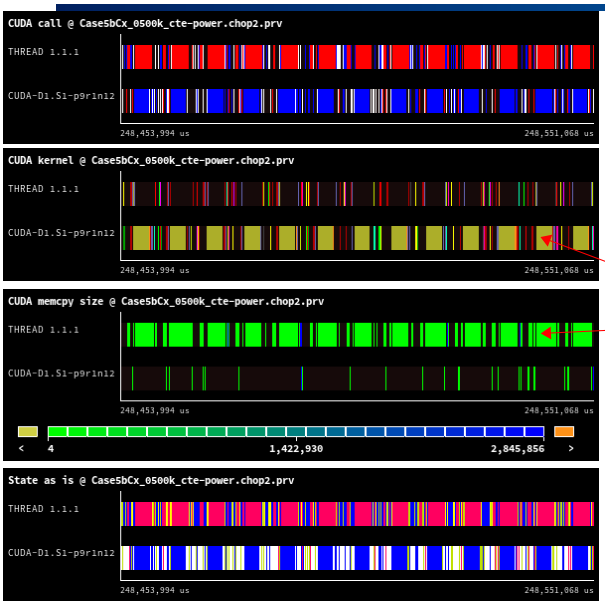
Grid Size	Block Size	Register per CUDA thread
(13745 1 1)	(128 1 1)	62

The kernel InteractionForcesFluid, its memcpy size is very small 4B.

API Calls

API	Time(%)	Time	Calls	Avg	Min	Max
cudaMemcpy	75.92%	361.1s	212781	1.7ms	17.4us	22.5ms

FOA: InteractionForcesFluid (Cx_500k)



GPU Utilization

kernel	Time(%)	Time	Calls	Avg	Min	Max
Interaction ForceFluid	79.6%	211.7s	64188	3.3ms	3.1ms	3.9ms

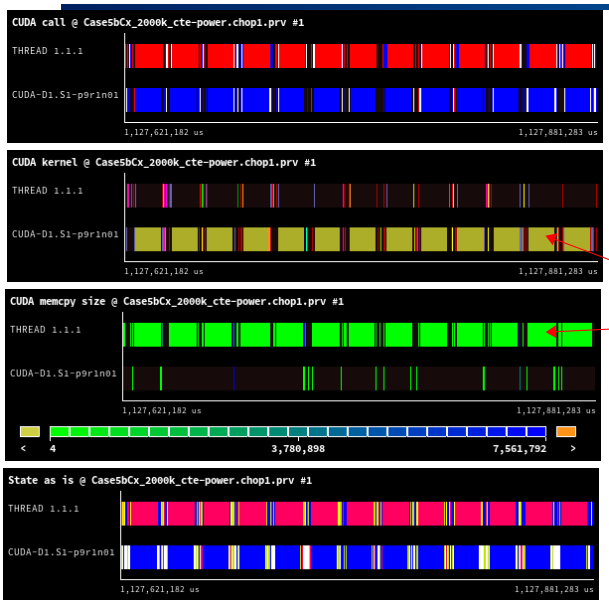
Grid Size	Block Size	Register per CUDA thread
(3196 1 1)	(128 1 1)	68

The kernel InteractionForcesFluid, its memcpy size is very small 4B.

API Calls

API	Time(%)	Time	Calls	Avg	Min	Max
cudaMemcpy	60.03%	235.0s	449330	523us	17.7us	110ms

FOA: InteractionForcesFluid (Cx_2M)



GPU Utilization

kernel	Time(%)	Time	Calls	Avg	Min	Max
Interaction ForceFluid	85.58%	1.6e+03s	111092	14.3ms	13.9ms	16.7ms

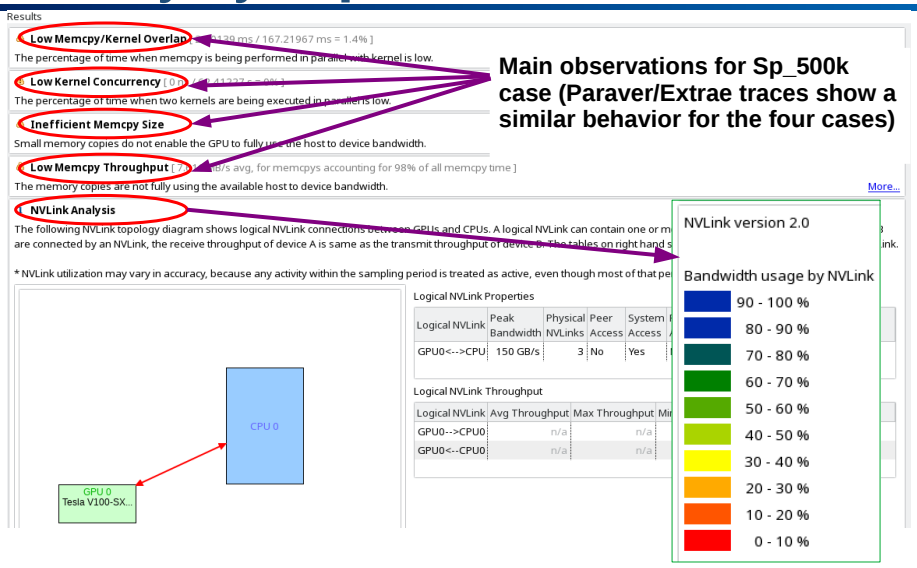
Grid Size	Block Size	Register per CUDA thread
(13745 1 1)	(128 1 1)	62

The kernel InteractionForcesFluid, its memcpy size is very small 4B.

API Calls

API	Time(%)	Time	Calls	Avg	Min	Max
cudaMemcpy	80.69%	1.7e+03s	777658	2.2ms	17.2us	64.4ms

Summary by nvprof



Summary of observations

- The kernel KerInteractionForcesFluid represents more than 70% of the total kernels execution time.
- The kernel KerInteractionForcesFluid has more impact on the execution time in the four cases, due to the small data copied by memcpy API. This behavior is similar for simple and complex physic cases.
- Computation time is less than 50% of the total runtime for the four cases.
- CUDA memcpy is representing around the 60% of the execution time that is produced by the the kernel KerInteractionForcesFluid.
- Application's Kernel needs an additional refactoring to increase the degree of parallelism, it could be an Multi-GPU or explicit Streams programming.

Performance Optimisation and Productivity

A Centre of Excellence in HPC

Contact:

https://www.pop-coe.eu

mailto:pop@bsc.es

@POP_HPC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 676553 and 824080.

References

[1] Sofia Davydycheva, Vladimir Druskin, and Tarek Habashy. An efficient finite-difference scheme for electromagnetic logging in 3D anisotropic inhomogeneous media. *Geophysics*, 68(5):1525–1536, 2003.

[2] J.M. Domínguez, A.J.C. Crespo, Gomez-Gesteira M., and B.D. Rogers. Simulating more than 1 billion sph particles using gpu hardware acceleration. *Proceedings of the 8th International SPHERIC Workshop*, pages 249–254, 2013.

[3] J.M. Domínguez, A.J.C. Crespo, D. Valdez-Balderas, B.D. Rogers, and Gomez-Gesteira M. New multi-gpu implementation for smoothed particle hydrodynamics on heterogeneous clusters. *Computer Physics Communications*, 184(8):1848–1860, 2013.

[4] Marta Garcia, Jesus Labarta, and Julita Corbalan. Hints to improve automatic load balancing with lewi for hybrid applications. *J. Parallel Distrib. Comput.*, 74(9):2781–2794, September 2014.

[5] Marta Garcia-Gasulla, Guillaume Houzeaux, Roger Ferrer, Antoni Artigues, Víctor López, Jesús Labarta, and Mariano Vázquez. MPI+X: task-based parallelization and dynamic load balance of finite element assembly. *CoRR*, abs/1805.03949, 2018.

[6] R. A. Harris, M. Barall, R. Archuleta, E. Dunham, B. Aagaard, J. P. Ampuero, H. Bhat, V. Cruz-Atienza, L. Dalguer, P. Dawson, S. Day, B. Duan, G. Ely, Y. Kaneko, Y. Kase, N. Lapusta, Y. Liu, S. Ma, D. Oglesby, K. Olsen, A. Pitarka, S. Song, and E. Templeton. The SCEC/USGS Dynamic Earthquake Rupture Code Verification Exercise. *Seismological Research Letters*, 80(1):119–126, January 2009.

[7] Alexander Heinecke, Alexander Breuer, Sebastian Rettenberger, Michael Bader, Alice-Agnes Gabriel, Christian Pelties, Arndt Bode, William Barth, Xiang-Ke Liao, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, and Pradeep Dubey. Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers. In *SC '14: Proc. Int. Conf. HPC, Networking, Storage and Analysis*, pages 3–14. IEEE, 2014.

[8] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. Libxsmm: Accelerating small matrix multiplications by runtime code generation. pages 981–991, 11 2016.

[9] Guillaume Houzeaux, Ricard Borrell, Yvan Fournier, Marta Garcia-Gasulla, Jens Henrik Göbbert, Elie Hachem, Vishal Mehta, Youssef Mesri, Herbert Owen, and Mariano Vázquez. High-performance computing: Dos and don'ts. In Adela Ionescu, editor, *Computational Fluid Dynamics*, chapter 1. IntechOpen, Rijeka, 2018.

[10] Etor Lucio-Eceiza, J. Fidel González Rouco, Elena Garcia Bustamante, J. Navarro, and Hugo Beltrami. Multidecadal to centennial surface wintertime wind variability over northeastern north america via statistical downscaling. *Climate Dynamics*, 12 2018.

[11] Etor E. Lucio-Eceiza, J. Fidel González-Rouco, Jorge Navarro, and Hugo Beltrami. Quality control of surface wind observations in northeastern north america. part i: Data management issues. *Journal of Atmospheric and Oceanic Technology*, 35(1):163–182, 2018.

[12] Etor E. Lucio-Eceiza, J. Fidel González-Rouco, Jorge Navarro, Hugo Beltrami, and Jorge Conte. Quality control of surface wind observations in northeastern north america. part ii: Measurement errors. *Journal of Atmospheric and Oceanic Technology*, 35(1):183–205, 2018.

[13] David Michéa and Dimitri Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using gpu graphics cards. *Geophysical Journal International*, 182(1):389–402, 2010.

[14] Paulius Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM.

[15] Carsten Uphoff and Michael Bader. Yet Another Tensor Toolbox for discontinuous Galerkin methods and other applications. *ACM Trans. Math. Softw.*, accepted, 2020. Preprint on `http://arxiv.org/abs/1903.11521`.

[16] Carsten Uphoff, Sebastian Rettenberger, Michael Bader, Elizabeth H. Madden, Thomas Ulrich, Stephanie Wollherr, and Alice-Agnes Gabriel. Extreme scale multi-physics simulations of the tsunamigenic 2004 Sumatra megathrust earthquake. In *SC '17: Proc. Int. Conf. HPC, Networking, Storage and Analysis*. ACM, 2017.

[17] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[18] Sebastian Wolf, Alice-Agnes Gabriel, and Michael Bader. Optimization and Local Time Stepping of an ADER-DG Scheme for Fully Anisotropic Wave Propagation in Complex Geometries. In Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira, editors, *Computational Science – ICCS 2020*, Lecture Notes in Computer Science, pages 32–45, Cham, 2020. Springer International Publishing.