H2020-INFRAEDI-2018-2020

The European Centre of Excellence for Engineering Applications

Project Number: 823691

D3.1
Report on Exa-enabling enhancements and benchmarks

The EXCELLERAT project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 823691

Workpackage:	WP3	Driving Exa-HPC Methodologies and Technologies	
Author(s):	Daniel Mira		BSC
	Ricard Borrell		BSC
	Ivan Spisso		CINECA
	Gabriel Staffelbach		CERFACS
	Thomas Gerhold		DLR
	Niclas Jansson		KTH
	Janik Schüssler		SSC
	Gavin Pringle		UEDIN
	Nicholas Brown		UEDIN
Approved by	Executive Centre Management		30/11/2019
Reviewer	Claudio Arlandini		CINECA
Reviewer	Tomislav Subic		ARCTUR
Dissemination Level	Public		

Date	Author	Comments	Version	Status
2019-10-15	D. Mira	Initial draft	V0.1	Draft
2019-10-28	ALL	First round of contributions	V0.2	Draft
2019-11-12	ALL	Additional contributions	V0.3	Draft
2019-11-26	D. Mira	Document submitted to review	V1.0	Draft
2019-11-28	ALL	Updates after internal review, Additional review round	V1.1	Draft
2019-11-29	D. Mira	Document accepted following review	V2.0	Final

List of abbreviations

CFD	*Computational Fluid Dynamics*
CLI	*Command Line Interface*
CPU	*Central Processing Unit*
CoE	*Center of Excellence*
CUDA	*Compute Unified Device Architecture*
DMA	*Direct Memory Access*
FPGA	*Field-Programmable Gate Array*
GASPI	*Global Address Space Programming Interface*
GPU	*Graphics Processing Unit*
HPC	*High-Performance Computing*
HLS	*High Level Synthesis*
MPI	*Message Passing Interface*
PETSc	*Portable, Extensible Toolkit for Scientific Computation*
RCB	*Recursive Coordinate Bisection*
SFC	*Space Filling Curve*
Spliss	*Sparse Linear Systems Solver*
TDP	*Termal Design Power*
UDF	*User Defined Function*
UPC	*Unified Parallel C*
VHDL	*VHSIC Hardware Desciption Language*
VHSIC	*Very High Speed Integrated Circuit*
VPN	*Virtual Private Network*
WP	*Work Package*

Executive Summary

The progress on the development of Exascale enabling technologies on the EXCELLERAT core codes is presented for the first year of the project. The developments have been guided by the definition of an individual code development roadmap in collaboration with Work Package 2 (WP2) and WP4, so the demonstration of Exascale simulations with the use cases can be achieved. From this roadmap, several requirements were identified (see D2.1 "Reference Applications: Roadmap and Challenges" [1]) and a summary of the activities conducted to address these requirements is presented here. Two fundamental activities are associated to these developments, Task 3.1 focused on node-level performance and Task 3.2 on system-level performance engineering. Note that main changes in the evolution of HPC systems are occurring at node level. This is a major reason to have a specific task focused on this topic.

In this first year, the activities carried out by the partners on these tasks have been focused on auditing the performance at node level (DLR, BSC, KTH, CERFACS), enabling the utilization of accelerators through the directives based language OpenACC (BSC, KTH), developing new data structures to better exploit new architectures (CERFACS, BSC) and developing techniques for the introduction of FPGAs on the CoE's codes (UEDIN).

The second major activity is focused on identifying and overcoming bottlenecks at system level that will arise on the road to Exascale. In this first year, the activities carried out by the partners have been focused on auditing the performance and system level and identify bottlenecks (DLR, KTH, CERFACS), improving the strong scaling of the codes (CERFACS, KTH) and designing and implementing new distributed memory load balancing strategies (BSC). The activities on this WP also include the development of a benchmark suite for each code to be able to test and monitor the evolution of the codes, and the development of an efficient data transfer and dispatching strategy to operate the codes in an HPC cluster. Meshing activities have recently started and a compilation of information from the partners involved in these tasks (KTH, CERFACS and BSC) has been conducted.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

The present document is a summary of all activities and achieved results within WP3 in its work on Exa-enabling during the first year of the EXCELLERAT project. It provides information on the optimizations of the reference applications' performance at system and node levels. In addition, the advances in meshing techniques, the implementation of the data layer and the benchmarks developed in the project are presented. The report is divided into different Sections that are referred to the different tasks of the EXCELLERAT WP3. This deliverable is made from the different contributions of the partners, which have been compiled and linked to the requirements of the use cases defined in WP2.

2 Task 3.1. Node-level performance optimization

The main changes in the evolution of HPC systems are occurring at node level. Consequently, the complexity associated with unlocking the intra-node performance of computing systems has increased substantially. This task addresses all the aspect related with performance at node level, including code porting and algorithms refactoring on various architectures. In this task, the level of readiness of each core code is analysed and the required developments supported. Here after we present the activities carried out in T3.1 for the first twelve months of the EXCELLERAT project.

2.1 Acceleration of Nek5000 kernels based on OpenACC

During the first year, KTH has focused on improving Nek5000's GPU performance, which has mainly been implemented using OpenACC directives. As a starting point the work was based on the proxy-app Nekbone [2], focusing on optimizing the small matrix-matrix multiplication kernels (*mxm*) which constitutes most of the work in Nek5000. In Nekbone, these kernels had already been implemented using both OpenACC and CUDA. Starting with the OpenACC version, since the implementation is fairly old, the performance could be improved up to 40% by reordering the loop directives.

The CUDA kernel in Nekbone was written in a general fashion without any optimizations for a particular polynomial order. But to achieve good performance for the CUDA kernels the *mxm* operations needs to fit into the GPU's shared memory, which is not possible for polynomial order ten and higher. However, production runs are seldom performed at these high orders, therefore those could be limited to ninth order kernels and rewrite them to take advantage of shared memory, making them up to 60% faster (Figure 1).

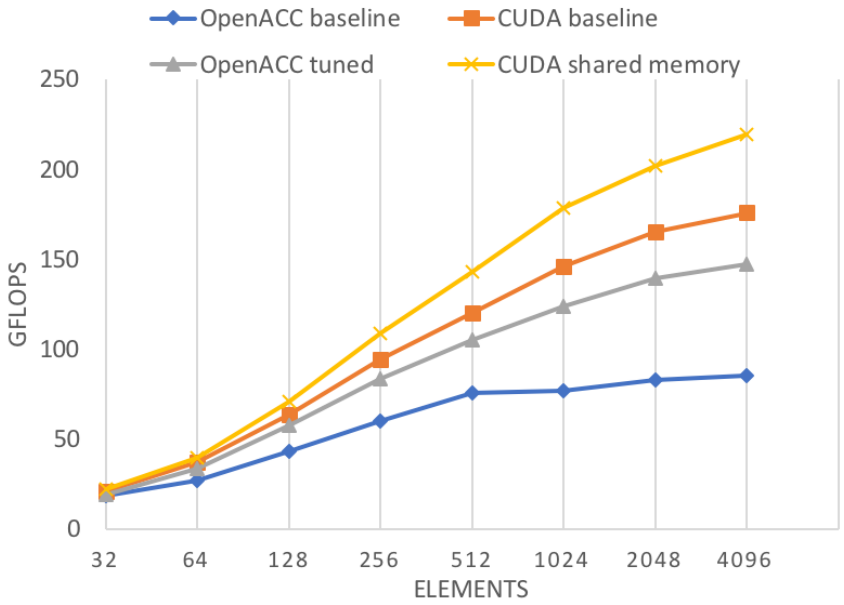

Figure 1: Performance results for Nekbone on a single GPU using 9th order polynomials

Based on the knowledge gain from tuning Nekbone, similar kernels were identified in Nek5000 namely *axhelm* and *multd*. Both kernels are quite similar to the ones in Nekbone, thus refactoring the OpenACC directives could be performed directly in Nek5000 reducing the actual kernel time by 34% and 24% respectively. Given the good experience with CUDA kernels in Nekbone the kernels *axhelm* and *multd* were also rewritten in Nek5000 using CUDA, restricting to polynomials of ninth order. These kernels reduced the runtime even

further compared to the reordered OpenACC versions, with 18% faster *multd* and 31% faster *axhelm* (Figure 2).

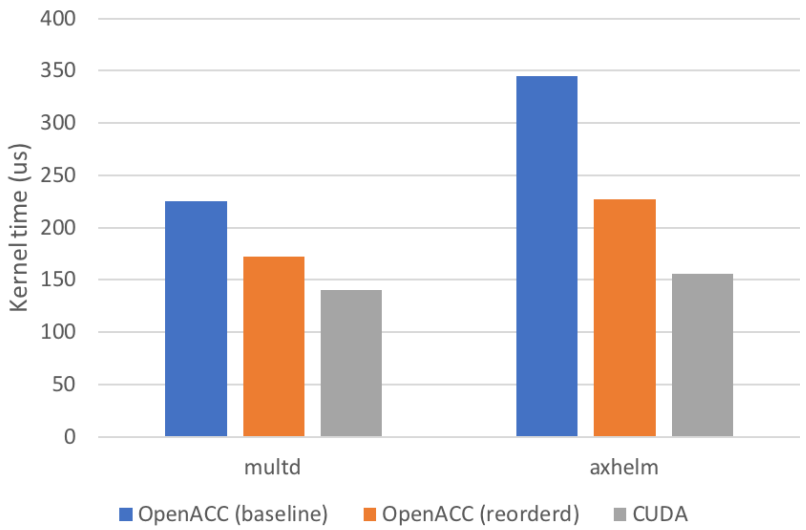

Figure 2: Optimized kernels in Nek5000

In the coming period the kernels of Nek5000 will continue being optimized, moving most of them to CUDA. However, most of the focus will be on obtaining good system-level performance across nodes.

2.2 Introduction of dynamic data structures in AVBP

A development required for the use case based on combustion instabilities and emission prediction, was to introduce new dynamic data structures in AVBP. This new packaging requires extensive validations but also performance evaluation and analysis. Within this task, the new releases of AVBP have been tested and optimized on available architectures. Furthermore, CERFACS was granted access to the new Tier 1 machine JEANZAY (2x20 Skylake) from GENCI-IDRIS [3] and an ARM prototype with ThunderX2 processors. The performance of AVBP 7.5 released in September 2019 was analysed accounting for requirements AVBP R1 (dynamic mesh structures) and R3 (automatic remeshing).

Porting to Intel processors, has identified a major bottleneck in the previous releases: the -fPIC option used to generate user define functions (UDF) in the code via dynamic libraries disables high level vectorization and reduces performance of the code by up to 30%. The issue is being investigated with a bug report set to Intel. UDF are not required for the uses cases and can be disabled for now.

2.3 Unified CPU/GPU vectorization strategy developed in Alya

The last level of parallelism within Alya consists of a SIMD vectorization on the CPU and SIMT in the GPU. A data restructuring has been carried out in EXCELLERAT to optimize the performance at this last level for both devices. In the GPU execution model, two additional parameters are needed: number of threads and blocks. The workload is divided in thousands of threads that are grouped into blocks. Each block containing the same amount of threads. The threads within the blocks are essentially executed in SIMD mode, so at this level the CPU and GPU optimal data structures follow the same pattern. The GPU manages the number of blocks that can keep active depending on the memory requirements of the threads (i.e. number of registers, shared memory and number of threads per block).

The work carried out at node level in Alya has been focused on generating a new data structure to enhance the efficiency of the assembly on both the CPU and GPU devices. Firstly, we carry out a data reordering to store contiguously in memory the elements of the same kind (tetrahedron, hexahedron, prism, and pyramid), that follow the same integration rule and that can be computed simultaneously without race conditions. To meet this last condition, classical colouring strategies have been used. Then, we group such elements into packs of size PACK_SIZE. Note that zeros are padded in the data structure when elements of the same category are not enough to fill a pack. Finally, the assembly runs on each pack of elements instead of on every single element. This approach has a two-fold benefit. On the one hand, it improves data locality, because it stores elements in dense packs. On the other hand, the code exposes the SIMD/SIMT potential and the compiler can leverage more instructions for the vectorial unit. We use the common approach for CPUs and GPUs being the PACK_SIZE then tuned for each specific device.

2.3.1 Optimization of PACK_SIZE for the CPU

The *PACK_SIZE* can have a significant impact on the CPU performance. Here we show this impact for the particular case of an Airplane LES simulation for a mesh of 31.5M elements [4]. We have performed the experiments on 10 nodes of the MareNostrum CTE POWER9 cluster, launching 40 MPI processes per node (i.e., one MPI process per CPU-core).

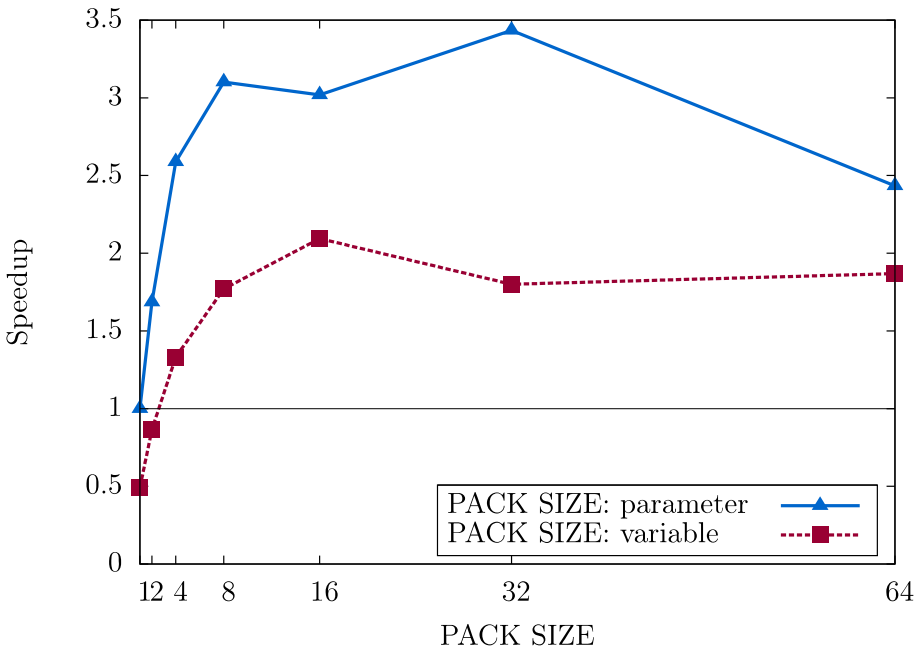

Figure 3: Speed up for different pack sizes, for PACK_SIZE defined as a Fortran parameter or a variable

Figure 3 shows the speedup obtained when using different values for the pack size in the CPU, considering both its definition as a compilation parameter or as a variable of the code. The speedup is computed according to the execution time using a pack size of 1 as a compilation parameter. The red line with square dots evaluates the improvement in performance due to the locality of the data, and it will be related to the length of the cache line for the last level cache. The optimum pack size, when used as a variable, is 16 in this architecture. If we look at the pack size defined as a compilation parameter, we are evaluating the combined benefit of the better data locality and the better use of the vector units. For this reason, the performance of the pack size defined as a compilation parameter is always better

than the same size defined as a variable. Note as a summary that the achieved speedup reaches 3.5x.

2.3.2 Optimization of PACK_SIZE for the GPU

The same analysis is shown in Figure 4 for the GPU execution. In this case the optimal PACK_SIZE is much larger than the one required for the CPU, something that is expected since a GPU needs a critical occupancy to achieve good performance. Finally, in the right part of Figure 4 we show the speedup of the GPU vs the CPU execution. In this case we are comparing the two POWER9 CPUs composing each node (40 CPU-cores in total) versus the performance using 4 GPUs. The Figure also shows different optimizations that we carried on the OpenACC based implementation developed to use the accelerators; the final speedup achieved is close to 4x.

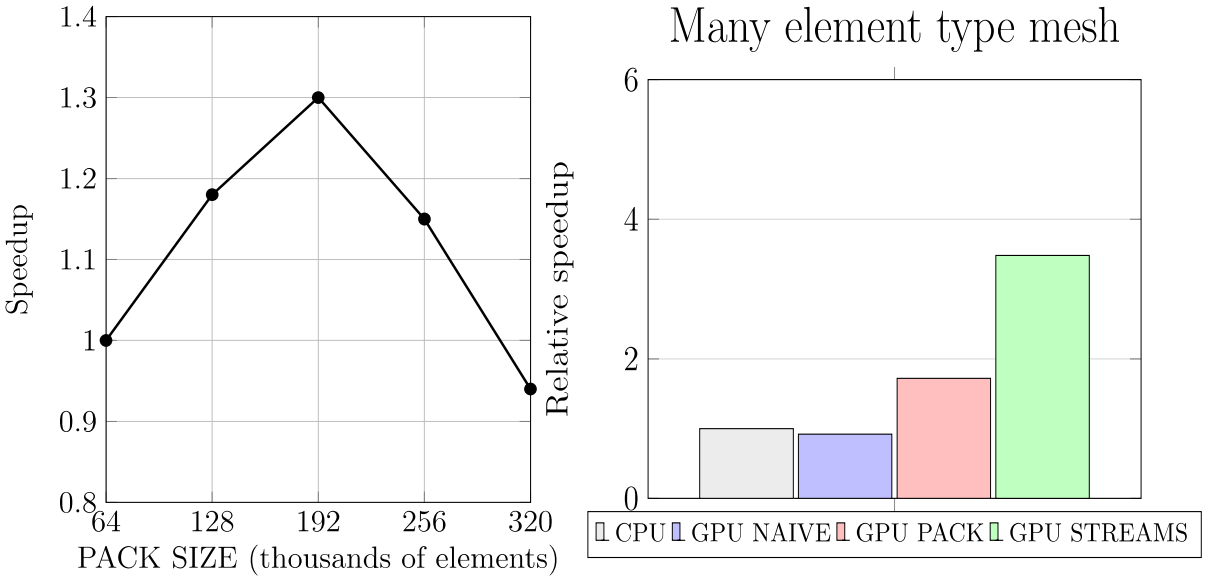

Figure 4: Left: Speed up for different pack sizes for PACK SIZE for the GPU execution. Right: speedup of the GPU execution vs the CPU execution

Further details of the developments and tests carried out in this task will be available in [4].

2.4 FPGA acceleration of the CoEs applications

In this period, UEDIN has been working on the FPGA acceleration of the CoEs applications. Field Programmable Gate Arrays (FPGAs) are configurable chips that can be programmed to execute specific functionality in hardware. This is potentially very beneficial for HPC codes because, in contrast to running on a CPU, executing directly in hardware can provide significantly increased performance at a fraction of the energy usage. Traditionally, FPGAs were very difficult to program, requiring the mastery of hardware description languages. However, in the past couple of years vendors have made very significant advances in the software development eco-system and it is now commonplace to program FPGAs using C or C++.

With the predicted slowdown in Moore's law, any alternative option to accelerating the CoE's codes is worth exploring. FPGAs are interesting because, not only do they avoid the overhead of a generalised microarchitecture, where the programmer can specialise the processing and related items such as the cache directly for their application, but also FPGAs can be configured to work at arbitrary precision. The latter is important because the HPC community

is currently very interested in reduced precision. Whilst the use of FPGAs in HPC is still fairly early, there are a number of HPC machines (e.g. the Cray CS500 at Paderborn University), which contain this technology, along with all the major cloud vendors.

As such, a key question for the EXCELLERAT CoE is, moving to Exascale, what role could FPGAs play in exploiting our applications for next generation science on future supercomputers? We are focussed very much at the node-level here, with one or more PCIe FPGA cards plugged into a single node. There are three questions that we have been focusing around in order to answer this overarching question:

1. Can FPGAs provide performance benefits for accelerating HPC codes?
2. What algorithm level modifications are required to fully take advantage of this technology?
3. What is the state of current software development tooling for FPGAs, and how might this be improved to suit the needs of HPC codes?

Up until this point we have been focussed on accelerating a single stencil-based code, with the idea being that the lessons learnt will then easily apply to other HPC codes in the CoE. The kernel we have focussed on contains 53 double precision operations per grid cell and accounts for around 50% of the runtime of the entire code. From the programming perspective we are using High Level Synthesis (HLS), where kernels written in C, C++ or System C, are translated into the underlying hardware description level by the tooling. Driven in code by pragma style hints, using a high-level language such as C substantially speeds up development time in comparison to traditional approaches such as VHDL. This also enables application developers to take advantage of the knowledge and experience of the FPGA vendor at the hardware level, for instance in the concrete implementation of floating-point operations. It should be noted that FPGAs come in all shapes and sizes. For this work we are using an ADM-PCIe-8k5 card which combines 16GB on-card DRAM with an Ultrascale Kintex FPGA. The big benefit of using a PCI-e based FPGA is, compared to embedded FPGAs such as the Zynq family, these can be combined with any x86 CPU and typically provide more resources.

2.4.1 Development of the kernel

A detailed description of the work done implementing the kernel in C using HLS is provided by [5] and [6], which have resulted directly from this work. Table 1 provides a general overview of the performance of our HLS kernel (running at 250 Mhz) at various stages of optimisation, against the original code running on 1 CPU core (Sandybridge) for a standard test-case with 67 million grid cells. The CPU code takes 676.4 ms runtime, and it can be seen that the initial port to FPGAs, with the kernel code unchanged from the CPU is over 70 times slower.

Description	Runtime (ms)
Reference CPU code	676.4
Initial port	51498
Pipelining loops	14130
Use of BRAM for caching	1513.2
Reordering memory access	621.3
Concurrent load and store to DRAM	189.64
Match data width to DRAM controller	63.49

Table 1: Runtime of kernel based on algorithmic changes

Table 1 can be thought of as illustrating the performance impact in adopting different strategies to optimise the code. These are critically important, because the final version of the HLS kernel ends up running over 10 times faster than on the CPU and over 800 times faster than the CPU code directly ported to the FPGA initially! To achieve this speed up the code has changed very substantially, requiring a significant rethink of the underlying algorithm, converting to from a *von-neumann* to *dataflow* style of computing.

The overarching steps we adopted in this optimisation can be explained in a fairly general manner, and at this point represent a set of best practice rules that we believe can be applied to numerous algorithms. This is important to highlight, as these rules have not been published or formalised previously, and many come from in-depth discussions with FPGA vendors. Whilst inevitably some specialisation is required on a kernel by kernel basis, building up as a community an overarching understanding of the steps required to optimise codes for FPGAs is of great benefit and furthermore mirrors efforts of the community a decade ago for GPUs. It is the reason why we have focussed on one initial application so far, and it is our strong belief that these lessons will now apply to many, if not all, of the CoE applications.

2.4.2 Performance comparison

Table 1 illustrates the performance of a single HLS kernel against a single CPU core. However, to understand the performance properties of the kernel on FPGAs against CPUs, a more in-depth study is required. CPUs contain multiple cores and an FPGA can contain multiple HLS kernels, so a multi-core and multi-HLS kernel comparison is more interesting. Furthermore, the measurements in Table 1 ignore the cost of data transfer to and from the PCIe FPGA card, which could represent a significant fraction of the overall runtime.

Figure 5 illustrates a performance comparison using a standard test-case of the code with 67 million grid points. The performance of our FPGA approach is compared against a C version of the same algorithm, threaded via OpenMP across the cores of the CPU (Sandybridge, Ivybridge, and Broadwell). For all runs the host code was compiled with GCC version 4.8 at optimisation level 3 and the results reported are averaged across fifty timesteps. For each technology there are two runtime numbers reported in milliseconds. The first, *optimal performance*, illustrates the best performance by threading over all the physical CPU cores (4 in the case of Sandybridge, 12 in the case of Ivybridge, 18 in the case of Broadwell) or the advection kernels (8, as this is the maximum that can fit on the FPGA chip.) We also report a four core number, which includes only running over four physical cores, or kernels in the case of the FPGA designs, as this is the limit of the Sandybridge CPU and allows a more direct comparison.

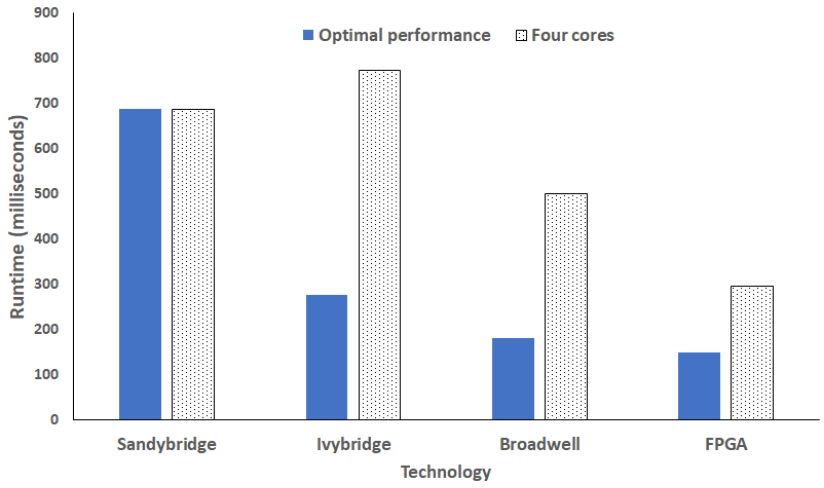

Figure 5: Speed up for different pack sizes, for PACK_SIZE defined as a Fortran parameter or a variable

With the optimal performance experiment, our HLS kernels are outperforming 18 cores of Broadwell (148 ms against 180 ms), and the other two CPU technologies. Eight HLS kernels are outperforming eighteen cores, and whilst it might seem that if we could fit more kernels onto the FPGA then performance would be even higher, it should be noted that the overhead of DMA transfer accounts for 42% of FPGA runtime at this problem size.

Figure 6 illustrates how the time, in milliseconds, changes one scales the number of grid cells. For our FPGA approach (8 kernels) we report three numbers, the total FPGA runtime, the execution time of the kernel alone (*FPGA kernel only runtime*) and the Direct Memory Access (DMA) transfer overhead time (*FPGA DMA overhead*). We compare against 18 cores of Broadwell, and for smaller grid sizes of 1 and 4 million grid cells our approach is 2.59 and 1.52 times faster than the CPU respectively. The FPGA and CPU are comparable at 16 million grid points, and the FPGA again outperforms the Broadwell by 1.22 times at 67 million grid points. However, Broadwell out performs the FPGA approach by 1.23 times at 268 million grid points.

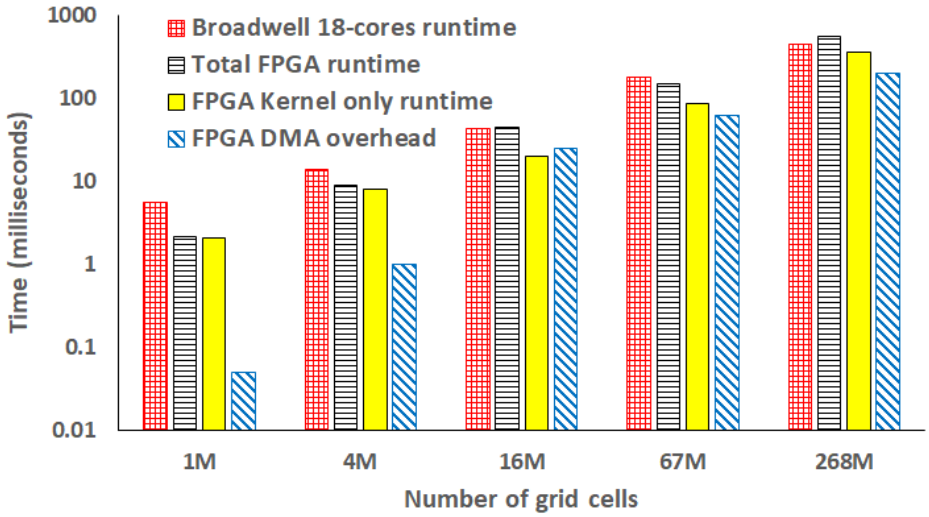

Figure 6: Runtime of FPGA code (8 kernels) vs 18 cores of Broadwell against grid size with a standard test-case. For our FPGA approach we report three numbers: the total FPGA runtime, the execution time of the kernel alone (FPGA kernel) and the FPGA DMA overhead

It should be noted that, at all grid sizes, the FPGA kernel execution time alone is significantly smaller than the execution time of 18 Broadwell cores. However, as the problem size increases, the waiting for data to be transferred from the host to the device (which is itself optimised, see [6] for details) is a source of over 40% overhead at 268 million grid points, whereas at a grid size of 1 million points it only accounts for 2% of the total runtime. Based upon on-board sensors, the configured but idle total power draw of the ADM-PCIe-8k5 board is 28.9 Watts and this increases to 35.7 Watts under full load with the largest problem size when our advection kernels are running. The TDP of the Broadwell is 120 Watts, so is drawing significantly more power to complete the computation.

2.4.3 Software development tooling

Whilst the tooling for programming FPGAs has improved considerably in the past few years, it is still not yet fully mature when compared against the environment HPC developers commonly enjoy. An example of this is the lack of profiling, where the software development tooling estimated that early versions of our HLS kernel were only spending around 20% of

runtime in computation, but without profiling this could not be validated during execution or insight gathered around where the rest of the time was being spent. As such, we developed a simple but effective technique which is illustrated in Figure 7. This connects our HLS kernel to a specialised profiling block that we also developed, and this profiler connects to a timer. Our HLS kernel communicates to the profiler to inform it when blocks of code are entered and exited, with the profiler collecting this information and sending it back to the host on termination. This approach was required due to limits in HLS which mean that collecting accurate timing data and computation cannot be mixed together in a single block. From the data gathered, we in-fact deduced that early versions of the HLS kernel were only computing for around 5% of the time, and were able to pin-point exactly where in the code the overhead lay.

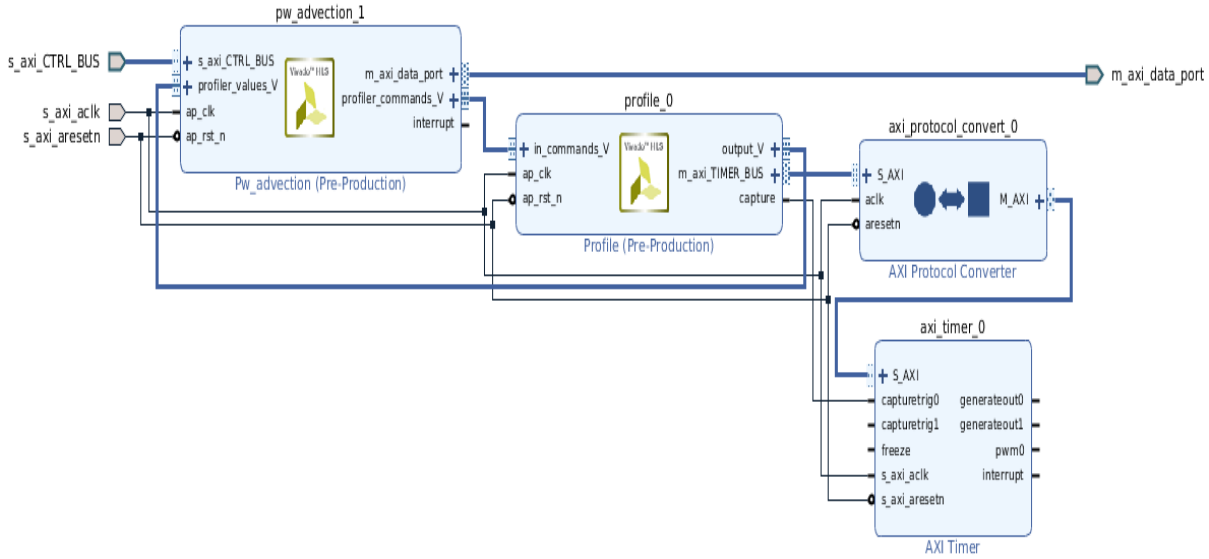

Figure 7: Profiling connection to HLS kernel and timer

2.4.4 Summary and next steps

The focus of this ongoing work is to leverage the knowledge and technology developed so far and apply this to a wider range of the EXCERLLERAT CoE applications. The optimisation methodology developed is applicable to a wide range of codes, and as such we are also planning on writing research papers about this, using the CoE codes as benchmarks and test-cases. Additionally, we have only explored kernels which are double precision floating point, and think it will be very interesting to consider alternative precisions and fixed point. This will be trivial to accomplish and we believe will significantly aid in accelerating CoE codes.

There is interest in this work from Xilinx and Alpha Data, both international companies, and already both hardware and software development licences have been donated from them. Going forwards it is very likely that they will provide us with further FPGA hardware, for instance next generation FPGAs that combine the chip with High Bandwidth Memory (HBM).

3 Task 3.2. System-level performance optimization

This task is focused on identifying and overcoming bottlenecks at system level. Load balancing and communication/synchronization reductions are key aspects to achieve parallel performance. Advanced features of MPI such as non-blocking collectives, fault tolerance and remote memory access will be considered throughout the project. The developments carried out in this task include both implementation optimizations and algorithms refactoring. Here after we present the activities carried out in T3.2 for the first twelve months of the EXCELLERAT project.

3.1 Improving strong scalability of FEniCS

In this reporting period, KTH has mainly focused on improving the strong scalability of matrix assembly in FEniCS. For time-dependent problems this has to be done in each time-step, thus it can quickly become a major bottleneck in a simulation. A key issue during matrix assembly is communication latency, in particular for low-order finite elements, at scale with few elements per core.

As a first step, the hybrid MPI+PGAS parallelization of FEniCS has been further developed and evaluated (Figure 8). In this branch of FEniCS, the linear algebra backend is changed from the MPI based PETSc [7] to a KTH-developed backend written in Unified Parallel C (UPC). This new backend stores the sparse matrix in the partitioned global address space, accessible by all ranks. With this abstraction, each rank can use low latency one-sided communication to fetch remote dependencies during matrix assembly. This greatly improves strong scalability of the assembly process, in particular for the very latency sensitive situations at scale with low-order elements.

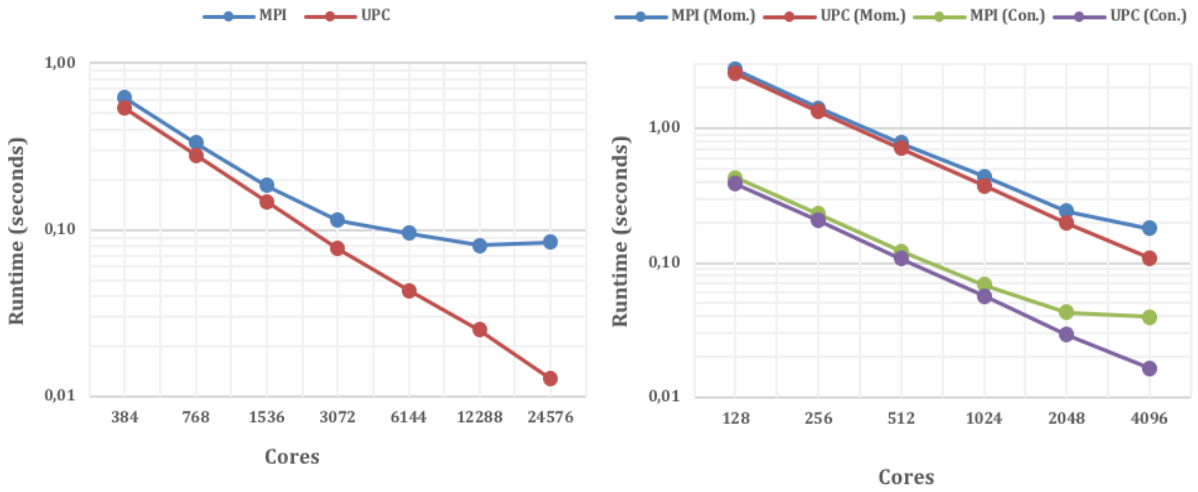

Figure 8: Matrix reassembly time for Laplace's equation in 3D on a mesh with 317M elements (left). Reassembly times for the momentum and continuity equations in an implicit LES solver on a mesh with 60M elements (right)

3.2 Improve strong scalability of AVBP

Strong scaling of AVBP has been tested on the JeanZay system up to 12k cores with excellent performance as demonstrated in Figure 9. Load balancing above 4k cores required the switch from ParMetis [8] to Treepart partitioning, based on recursive coordinates bisection, to avoid crashes in MPI collective calls. Treepart is a new CERFACS partitioning library to uses the system hierarchical structure to reduce communications and map the mapping to the node/socket/core distribution.

The tests were performed using the Intel 2019.0.4 compiler and MPI suite and HDF5 1.8.21 using use case C3U1 [1] without mesh adaptation. Additionally, the code was ported on an experimental cluster equipped with thunderx2 processors. Scaling has been tested up to 1024 MPI tasks so far with adequate results (80% strong scaling). Tests for larger systems are expected in Q1 2020 (access to UEDIN and JSC systems have been requested). An early user access to the IRENE Joliot Curie AMD extension TGCC-GENCI Tier 0 system has been granted for Dec-April.

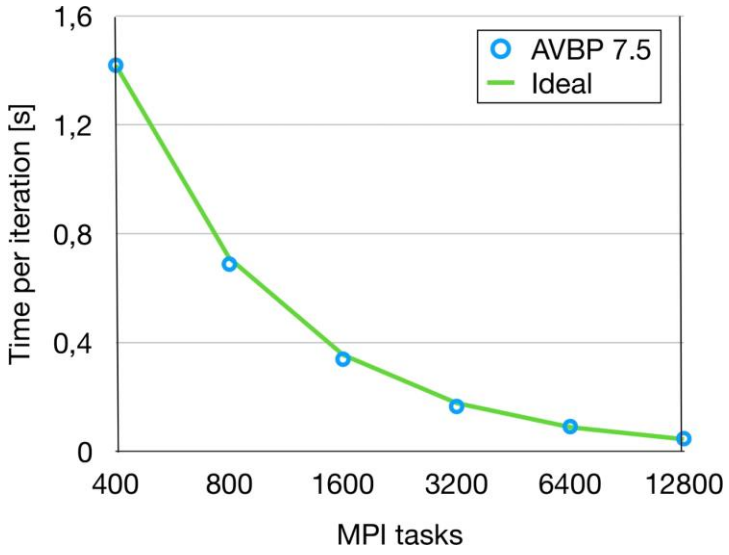

Figure 9: Strong scaling for AVBP in the JeanZay system (C3U1: static mesh)

3.3 System level dynamic load balancing enabled in Alya

In this first year of the EXCELLERAT project, a dynamic load balancing strategy has been implemented in Alya. This is a runtime mechanism that is executed during the simulation. In particular, these developments accomplish the requirement Alya-R2 ("Dynamic load balancing") and is a basic building block for Alya-R3 ("Mesh adaptation") that requires dynamic load balancing to be efficient in parallel, see [1] for details about the requirements. The dynamic load balancing strategy implemented in Alya is based on an efficient in-house SFC-based mesh practitioner. The partition is repeated with some correction coefficients to correct the measured imbalances. Therefore, it is mandatory that the partition process is fast to minimize the overhead of the balancing process. Some optimizations have been implemented on the partitioning algorithm, which were recently presented in the SC19 conference in Denver [9]. Below, in Table 2 we show the partition costs for a mesh of 250M elements for Airplane simulations (C2U2). In particular, the performance of the in-house partition is compared with the Zoltan library [10] from Sandia National Laboratories. We observe that the speedup of our implementation reaches up to 10x.

Partitions	Nodes used	LB in-house	LB Zoltan	Time (s) in-house	Time (s) Zoltan	Speedup
384	8	0.99	1.0	0.25	0.87	**3.5x**
768	16	0.99	1.0	0.15	0.54	**3.6x**
1536	32	0.99	1.0	0.10	0.48	**4.8x**
3072	64	0.99	1.0	0.07	0.50	**7.1x**
6144	128	0.99	1.0	0.08	0.79	**9.9x**

Table 2: Comparison of the in-hose SFC partitioner with the Soltan library. The case used is a mesh around an airplane of 250M elements.

The efficiency of the partitioning algorithm enables its utilization for dynamic load balancing. We have carried out all the developments required to restart Alya online, this means basically reallocation of the arrays as well as redistributing data among the MPI processes. An illustration of this feature is shown in Figure 10 for an airplane simulation using 24 POWER9 AC922 CPUs, each one with 20 Cores. In the configuration employed, one MPI-process is assigned to each pair of cores, where 2 OmpSs [11] threads are launched. We can observe the elapsed time per node with the initial imbalanced distribution (red line) and after the balancing process is carried out (blue line).

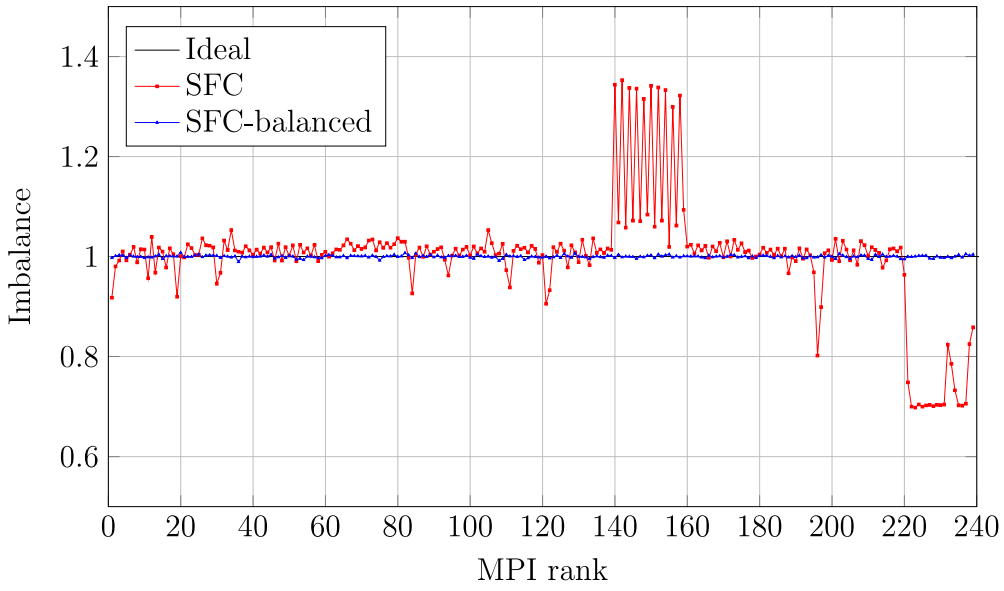

Figure 10: Normalized elapsed time per MPI rank. Assembly phase of the airplane simulation (176M elements mesh)

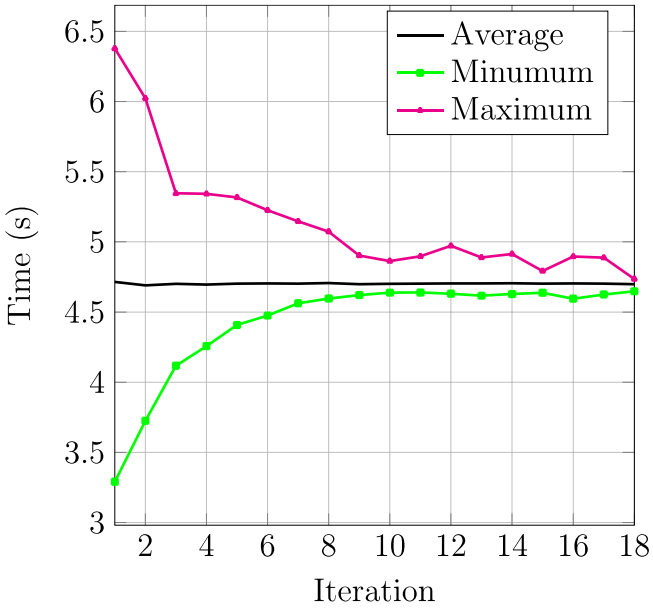

Figure 11: Convergence of the balancing process (176M mesh). Evolution of the maximum, minimum and average time for the assembly phase

Finally, in Figure 11, we show how the maximum and the minimum time tend to the average time through the online balancing process. Further details of the developments and tests carried out in this task will be available in [4].

3.4 System level performance analysis in CODA

In 2013, DLR started the implementation of the next generation CFD solver FLUCS. Since 2018 FLUCS is the basis of a strong partnership between Airbus, ONERA and DLR focusing on the development of a common next generation CFD code for aircraft flow predictions. In January 2019, the consortium agreed on the name CODA for the common CFD code.

CODA is still under active development, i.e. it currently includes a subset of the planned functionality and its scalability is in the order of thousands of cores. Due to the ongoing development and frequently changing functionalities of CODA, one of the main tasks of the performance analysis and optimization process, is the continuous re-analysis of the code. For instance, in the recent period the internal linear algebra solver was replaced by the newly developed Sparse Linear Systems Solver (Spliss).

The work in CODA focused on five main activities. First, we performed an initial performance measurement, analysis and evaluation of CODAs current state (FLUCS-R3). This resulted in an internal performance report that includes a detailed analysis of CODA's performance (node-level and system-level), the identification of potential performance issues and recommendations for code optimization (FLUCS-T1) [1]. After that, a second performance analysis was performed on an improved version of the test case and the results internally discussed. For both analyses we used a very small version of the Use Case C6U1 to allow a strong scalability analysis at relatively small core counts. Figure 12 highlights some analysis results and the speedup for the small test case.

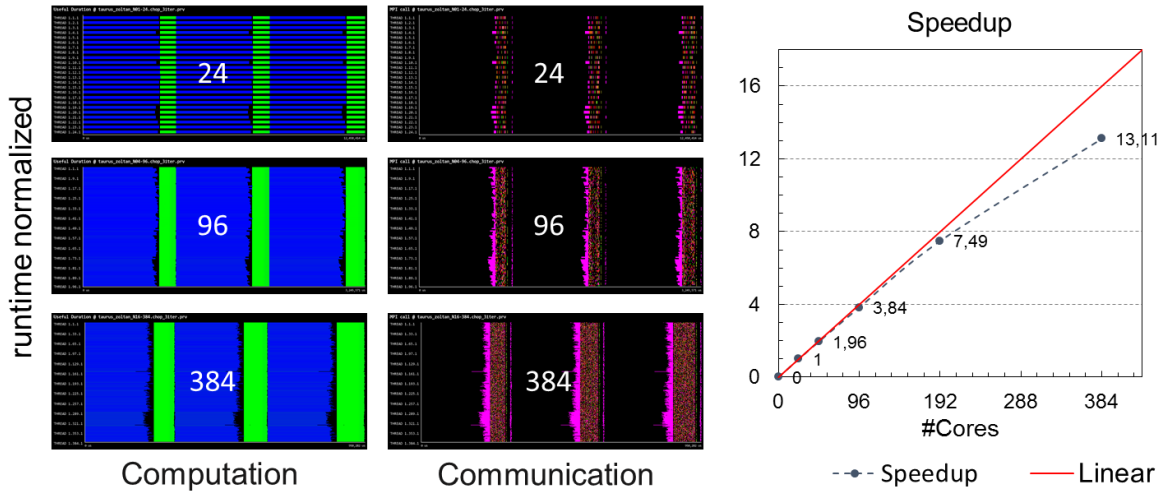

Figure 12: Performance analysis results and speedup for a very small test case designed to test strong scalability at low core counts

Second, we compiled a list of priorities for improving the performance of CODA, which are internally documented (FLUCS-T2). We started with their realization and implementation (FLUCS-T3).

Third, we started the integration of basic performance metrics in the user interface of CODA and their inclusion in the continuous software integration and review process. This allows setting up a common performance baseline and quickly identifying software changes that introduce performance degradation.

Fourth, we performed a study to analyse two different methods for the partitioning of mesh data to the processes: the fast-recursive coordinate bisection (RCB) method and the graph partitioning method Zoltan [10]. We analysed the impact of both partitioners to identify the causes for the different resulting runtime behaviour.

Fifth, we cooperate with two performance analysis tool providers to extend their tools' functionality to support complex engineering codes like CODA. Since CODA is implemented in Python and C++ with a multi-level parallelization via MPI or GASPI and OpenMP, it is a challenging application for current performance analysis tools and currently no existing tool allows an analysis of all CODA features and parallel programming models.

4 Task 3.3. Implementation of advanced meshing techniques

This section describes the activities related to the meshing techniques that have been developed during the first year of the project. There are two partners (BSC and CERFACS) involved in the development of the adaptive mesh refinement (AMR) with two codes, Alya and AVBP. The third partner involved in this task is KTH, as Nek5000 has already this capability and the effort is only directed to optimizing this tool for certain conditions, so this effort is not reported here, but on WP2 in particular for the deliverable D2.2.

Despite this task has not started for Alya and only partially started for AVBP, an overview of activities planned for the two codes is now described. In the case of Alya, the mesh adaptation corresponds to the requirement Alya-R3 in Year 2 that will focus on the implementation of adaptivity to the current non-adaptive meshes (parallel dynamic mesh adaptivity, accurate mesh adaptivity according to physics). This activity required the achievements of Alya-R1 (load balance strategy) and Alya-T2 (parallel pre-processing) before starting. As already described in the previous sections, these capabilities are available in the code and the mesh adaptivity work will start in M13. For the case of AVBP, the effort on mesh adaptivity has recently started. The work is based on the AVBP-R1: dynamic mesh structure at runtime, AVBP-R2: accurate interpolation methods, AVBP-R4: Incorporate automatic mesh refinement and AVBP-R5: remeshing. The activities for AVBP-R4 and AVBP-R5 have recently started in M11.

5 Task 3.4. Test lab for emerging technologies

The activities on this task were focused on testing the use cases or kernels representing the use cases on emerging technologies. During the first year of the project, PRACE resources were allocated for the partners involved in this activity, and despite most of the activities are going to start in year 2, some progress was accomplished for testing in ARM-based systems.

The use of ARM in HPC is of great interest, as it promises to be an important future HPC technology, but there are key questions about how best to leverage this for HPC. For instance, how do our different applications perform and scale on ARM based systems, what modifications are required to these codes to fully take advantage of ARM systems, and how do these codes running on ARM compare against x86 based HPC systems.

Fundamental to all of this is the correct choice of underlying communications library, and we have investigated the relative performance differences of these and presented this during an invited talk at the MVAPICH User Group in Ohio. We are using Fulhame for this work, providing 64 nodes each containing two 32-core Marvel ThunderX2 ARM CPUs (4096 cores total), comparing and contrasting the different MPI implementations on AR. These implementations were MVAPICH, OpenMPI, and HPE's MPT, and driving this exploration was the investigation of performance and scaling for a number of popular HPC codes, including CoE applications. For context, we also compared against a couple of x86-based systems, MVAPICH and MPT running on Cirrus, which is an x86 Broadwell system with InfiniBand, and ARCHER, a Cray XC30 system with Aries interconnect and Cray's tuned implementation of MPICH.

There were some really interesting patterns highlighted and, generally speaking, MVAPICH was very competitive against the other implementations, for instance on the x86 Cirrus system it outperformed MPT, in some cases very significantly. On Fulhame, the performance patterns were more nuanced, where in some cases MVAPICH demonstrated some really important performance benefits, for instance with 2D pencil decomposed FFT codes as their *AlltoAl*l collective significantly out performs what OpenMPI or MPT provide. In other situations, OpenMPI or MPT performed slightly better, but it is very important to note that OpenMPI had to be configured to select the correct communication protocol and instead MVAPICH gave good performance *out of the box*. The quantitative values are still being worked out.

The MVAPICH team have not yet fully tuned their technology for ARM, and this is very important because MVAPICH contains many advanced algorithms, which suit different situations and as ARM systems are so new, then it is likely that rules selecting which algorithm to use when, need to be tweaked. A video about the aforementioned invited talk at the MVAPICH User Group in Ohio was published by insideHPC [15].

6 Task 3.5. Validation and benchmarking suites

In order to quantify and evaluate the progress and evolution of the codes after the technical developments made in WP3, some benchmarks were defined for each code. This will permit to monitor the progress of the codes throughout the life of the project and evaluate the performance of the codes respect to the starting day. These benchmark cases or micro-benchmarks are not expected to be as the use cases of WP2, but they are defined in order to expose the bottlenecks of the codes when running the reference applications in WP2. The different cases and activities involved in the execution of these benchmarks are provided below in Tables 3-7.

Partner	BSC		
Code	Alya		
Test case	Technically premixed swirling combustor		
Linked use case	C2U1 - Emission predictions in engines		
Requirements (WP2)	Alya-R1: Fully parallel workflow		
Objective	Analysis and optimization of the pre-processing stage		
Short Description	6.1.1.1 The hybrid mesh is based on a combination of tetrahedrons, prisms and pyramids with different levels of refinement within the domain. Two meshes are considered of with 110 million (M1) and 856 million (M2) elements respectively. The analysis includes the operations going from the mesh reading to the start of the time marching.		
Activities	**Description**	**Start date**	**End date**
A1	6.1.1.2 Performance analysis and identification of bottlenecks	M1	M4
A2	Low level optimizations	M5	M8
A3	6.1.1.3 Algorithms refactoring to overcome parallelization bottlenecks	M9	M12

Table 3: Benchmark suite Alya

Partner	CERFACS
Code	AVBP
Test case	Explosion simulation
Linked use case	C3U1 – Explosion simulation
Requirements (WP2)	AVBP-R1: Dynamic mesh structure at runtime AVBP-R4: Efficient remeshing
Objective	6.1.1.4 Profiling and measurement of simulation time requirements pre- mesh adaptation.
Short Description	6.1.1.5 Static meshes need to be accurate for all stages of a simulation. In this case the mesh is uniformly refined to be able to discretize the flame everywhere even though the scales to resolve are much large for 90% of the domain at a given time step. The analysis will be used as a benchmark to assess the gains that will be gained by having a dynamic mesh refinement method.

Activities	Description	Start date	End date
A1	6.1.1.6 Introduce dynamic mesh structure in the code	01/01/2019	30/09/2019
A2	6.1.1.7 Validation of the code using in house non-regression tests	15/06/2019	15/09/2019
A3	6.1.1.8 Perform a Large Eddy Simulation with the new code and measure each computing phase time per time-step and time to solution	15/09/2019	31/12/2019

Table 4: Benchmark suite AVBP

Partner	DLR		
Code	FLUCS/CODA		
Test case	6.1.1.9 CFD-solver for aircraft aerodynamics		
Linked use case	C6U1		
Requirements (WP2)	None		
Objective	6.1.1.10 Analysis and optimization of FLUCS/CODA		
Short Description	6.1.1.11 The use case will demonstrate the CFD solver performance and scalability based on an aircraft geometry.		
Activities	**Description**	**Start date**	**End date**
A1	Performance analysis	M1	M6
A2	6.1.1.12 Concepts and implementation for potential performance enhancements	M7	M12

Table 5: Benchmark suite CODA

Partner	KTH		
Code	Nek5000		
Test case	6.1.1.13 AMR simulation of flow over NACA0012 airfoil with 3D wing tip		
Linked use case	C1U1 – Wing with 3D wing tip		
Requirements (WP2)	Nek5000-R3 : Efficient strategies for hex-based meshing of complex geometries Nek5000-R4 : Proper scheme for element's geometry description and projection of grid points on defined surface		
Objective	6.1.1.14 Pre-processing stage: building hex-based coarse mesh for moderately complex geometries 6.1.1.15 Code initialisation: testing initial AMR pipeline focusing on geometrical mesh consistency		
Short Description	6.1.1.16 Performing AMR simulation starts with creating very coarse mesh, that would be later refined in the region with significant computational error. For hex-based meshes with complex geometries this is a challenging task. During a run the mesh is dynamically modified by adding/removing computational subdomains (elements) keeping external domain surfaces unchanged. This requires additional geometry correction step based on 3D projection.		
Activities	**Description**	**Start date**	**End date**

A1	6.1.1.17 3D projection routines for NACA0012 profile with rounded wing tip	Mar 2019	Apr 2019
A2	6.1.1.18 Coarse mesh of NACA0012 profile with rounded wing tip	Apr 2019	May 2019
A3	6.1.1.19 Initial refinement on wing surface (without use of error indicator)		

Table 6: Benchmark suite Nek5000 – (1)

Partner	KTH
Code	Nek5000
Test case	6.1.1.20 AMR simulation of flow over 3D periodic hill
Linked use case	C1U1 – Wing with 3D wing tip
Requirements (WP2)	Nek5000-R5 : High quality mesh partitioner based on graph bisection Nek5000-R6 : Efficient pressure preconditioner for non-conforming, deformed elements
Objective	6.1.1.21 Code initialisation: testing mesh partitioning using graph bisection; testing initialisation of the coarse-grid solver for deformed elements 6.1.1.22 Code executions: monitoring pressure iteration count for different element aspect ratio.
Short Description	6.1.1.23 A key aspect of the performance of the incompressible flow solver is efficient solution of pressure problem, as divergence-free constraint is a man source of stiffness in the set of equations. In this test we focus on the main performance issues e.g. work balance and efficient pressure preconditioner.

Activities	**Description**	**Start date**	**End date**
A1	6.1.1.24 Merging/adapting existing AMR branch with official Nek5000 repository	May 2019	
A2	6.1.1.25 Testing different partitioning tools (ParMETIS, PARRSB)		
A3	6.1.1.26 Improved pressure preconditioners for non-conformal meshes using AMG		
A4	6.1.1.27 Improved pressure preconditioners for non-conformal meshes with deformed elements		

Table 7: Benchmark suite Nek5000 – (2)

7 Task 3.6. Data dispatching through data transfer

The general goal is to combine data transfer and data management. The vision is to provide a new software solution, on which the data, that needs to be calculated, is uploaded, sent to the cluster, compiled and executed. Further services could be:

- Possibility to interact with cluster through a Command Line Interface (CLI).
- Visualization of result data.
- Data transmission in encrypted form.
- Fast data transfer due to a data reduction technique.
- Visual feedback on cluster allocation in form of a dashboard.

The platform will be connected to all HPC systems in the project. At any time, there should be traceability of what happens to the data or where the data is located.

In the first few months, the goal is to develop a prototypical application in which a first real HPC use case can be mapped. The first use case will be the use case C2U1 from BSC in collaboration with RWTH Aachen, which will be implemented into the system. That means, the source code of the solver will be compiled and integrated into the software. When all configuration files are available, the actual solver execution will take place and the result data is going to be transferred back. For the future, the following feature could be implemented:

- Compression of the returned data
- Data encryption of the transferred content
- Addition of more solvers
- Possibility to connect all HPCs

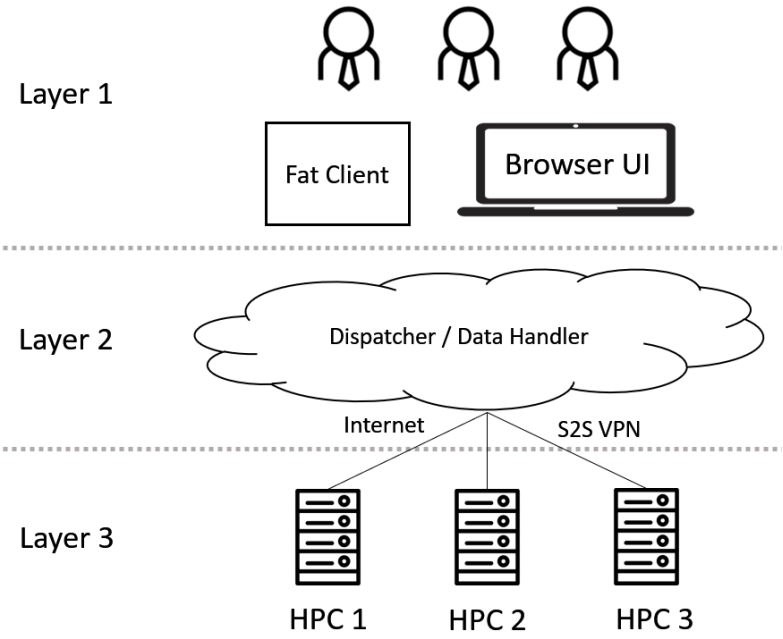

Figure 13: Conceptual model of the data transfer system

Figure 13 gives an initial overview of how the new system could look like. The complete service should be distributed over three layers. The first layer are the users and the available interfaces they can use. For example, there should be a web interface and a small client. The web interface could be used to handle smaller amounts of data and the client should perform more complex actions like delta building or data compression. With both interfaces it should be possible to upload data, configure jobs and download or view the result data.

In the second layer, a central distribution mechanism would be installed. This would be a data dispatcher which would also be responsible for data management. This layer can either run in one of the HPCs or in the cloud, where a very high data throughput is possible - for example Google or Amazon cloud services. Of course, the data would not be stored there and only passed to the appropriate HPC. The transition between the second and the third layer could be controlled either directly via the Internet or, for example, with a Site-2-Site VPN.

The lowest layer is formed by the individual HPCs on which the code is executed and each of these HPCs needs to run a small application, in order to communicate with the second layer.

Furthermore, the creation of data deltas could contribute to a good data management system. Each file will have a unique content identifier which is built for example by 1 MB blocks and each of these files has its one hash. In the end there is one big tree with all the hashes. That means if a file changes, only the changed blocks and not the whole file has to be transferred.

Figure 14 represents a first draft of a general HPC workflow, which should be transformed onto the new software.

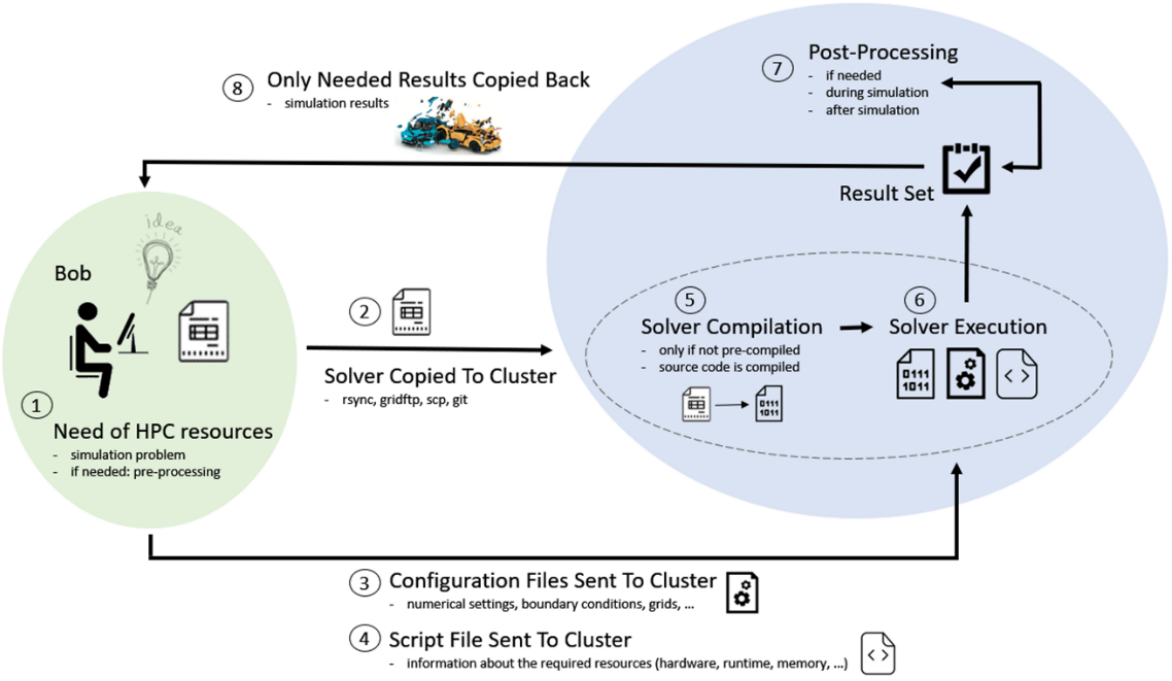

Figure 14: HPC Workflow

The following description will give a technical overview of how the all-in-one platform is built and which technology is used.

The platform is based on a container-based infrastructure and microservices. Microservices are small, autonomous services that have a single job and work together. In order to run the networked services in a secure and connected way, Istio [12] is used as service mesh.

The container architecture in this case is Docker, which is managed by Kubernetes. Kubernetes is an open-source system for automating deployment, scaling and management of containerized applications. To operate and scale the Kubernetes cluster on an infrastructural level, the cloud service provider from Google Cloud is used at the moment. Therefore, the package manager Helm is used to provide applications into Kubernetes.

The programming languages used in the repositories are mainly Java and Typescript. The following markup, style and script languages are also partly used: HTML, CSS and JavaScript.

The whole source code of the platform is managed by a self-hosted GitLab. GitLab offers a location for online code storage and collaborative development of software projects. Figure 15 gives an overview of the project structure.

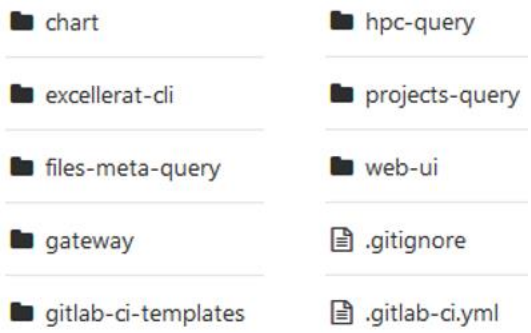

Figure 15: Project structure

Each project folder is responsible for single functionality in the platform. For example, the "*web-ui*" provides the web server Nginx, that stores web site files and broadcasts them over the internet. The "*gateway*" and "*projects-query*" folders contain various applications like MongoDB [13] or Micronaut [14].

Micronaut is used as JVM-based, full-stack framework for building modular microservice applications. Gradle is used as build tool behind that assembles the individual components into finished JAR files, which are then transformed into docker images using the Java Jib plugin. In order for all components to be built successfully, a separate bash script is used to build finished docker images from the gateway, project, and Web UI components.

For storing all the data, the two databases MongoDB and Neo4j are implemented. The document database MongoDB stores data in JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time. On the other side, Neo4j is an open source graph database management system. A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. It is composed of two elements - nodes and relationships.

In order to publish and subscribe to streams of records or to store and process streams and events, the platform Apache Kafka is used. It also provides multiple interfaces for writing data to Kafka clusters, reading data, importing and exporting data to and from third-party systems and it acts as a messaging system between the sender and the receiver. The Kafka client sends any event into the project queue, which can then be consumed by anyone.

To enable messaging, in order to connect and scale the all-in-one platform, the message broker RabbitMQ is used. It is a message-queueing software to which all HPCs of the all-in-one platform are connected to.

8 Conclusion

As a conclusion, the progress on the development of Exascale enabling technologies on the EXCELLERAT core codes for the first year of the project has been presented. Most of the work has been dedicated to node-level performance and system-level performance engineering. The activities carried out by the partners on these tasks have been focused on auditing the performance at node and system level, enabling the utilization of accelerators, developing new data structures and developing techniques for the introduction of FPGAs on the CoE's codes. Additional focus has been given to improving the strong scaling of the codes and designing and implementing new distributed memory load balancing strategies. A benchmark suite for each to test and monitor the evolution of the codes has been put in place, and the development of an efficient data transfer and dispatching strategy to operate the codes in HPC cluster has been accomplished during this first year. Meshing activities have recently started and reports and progress will be presented on the progress report of year 2 (D3.2).

9 References

[1] EXCELLERAT project, D2.1 “Reference Applications: Roadmap and Challenges”

[2] https://github.com/Nek5000/Nekbone

[3] http://www.idris.fr/annonces/annonce-jean-zay-eng.html

[4] R. Borrell, D. Dosimont et al. Airplane Simulation using Heterogeneous CPU/GPU co-Execution targeting the POWER9 Architecture. Future Generation of Computer Systems, under review.

[5] Brown, N 2019, Exploring the acceleration of the Met Office NERC Cloud model using FPGAs. in ISC High Performance 2019 International Workshops. ISC19 IXPUG Workshop: Using FPGAs to Accelerate HPC & Data Analytics on Intel-Based Systems, Frankfurt, Germany, 20/06/19.

[6] Brown, N, Doleman D 2019, It’s all about data movement: Optimising FPGA data access to boost performance. To appear in Fifth International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'19), Denver, USA, 17/11/19

[7] Abhyankar, Shrirang and Brown, Jed and Constantinescu, Emil M and Ghosh, Debojyoti and Smith, Barry F and Zhang, Hong. PETSc/TS: A Modern Scalable ODE/DAE Solver Library. arXiv preprint arXiv:1806.01437,2018.

[8] A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. George Karypis and Vipin Kumar. 10th Intl. Parallel Processing Symposium, pp. 314 - 319, 1996.

[9] R. Borrell, G. Oyarzún, D. Dosimont and G. Houzeaux, Parallel SFC-based mesh partitioning and load balancing. Proceedings of ScalA2019: 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, SC19 Denver.

[10] E. G. Boman and U. V. Catalyurek and C. Chevalier and K. D. Devin. The Zoltan and Isorropia Parallel Toolkits for Combinatorial Scientific Computing: Partitioning, Ordering, and Coloring, Scientific Programming, (20): 2012.

[11] https://pm.bsc.es/ompss

[12] https://istio.io

[13] https://www.mongodb.com

[14] https://micronaut.io

[15] https://insidehpc.com/2019/08/a-performance-comparison-of-different-mpi-implementations-on-an-arm-hpc-system/