H2020-INFRAEDI-2018-2020

The European Centre of Excellence for Engineering Applications

Project Number: 823691

D3.3
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
	Gabriel Staffelbach	CERFACS
	Thomas Gerhold	DLR
	Niclas Jansson	KTH
	Adam Peplinski	KTH
	Jens Gerle	SSC
	Gavin Pringle	EPCC
	Nicholas Brown	EPCC
Approved by	Executive Centre Management	
Reviewer	Andreas Ruopp	USTUTT
Reviewer	Claudio Arlandini	CINECA
Dissemination Level	Public	

Date	Author	Comments	Version	Status
17/05/2022	Daniel Mira	First full version	V0.0	
19/05/2022	Claudio Arlandini	Review	V0.1	
23/05/2022	Andreas Ruopp	Review	V0.2	
24/05/2022	Daniel Mira	Final updates	V1.0	

List of abbreviations

AMR	*Adaptive Mesh Refinement*
CFD	*Computational Fluid Dynamics*
CLI	*Command Line Interface*
CPU	*Central Processing Unit*
CRM	*NASA Common Research Model*
CoE	*Center of Excellence*
CUDA	*Compute Unified Device Architecture*
DLB	*Dynamic Load Balancing*
DMA	*Direct Memory Access*
DSP	*Digital Signal Processors*
FPGA	*Field-Programmable Gate Array*
GASPI	*Global Address Space Programming Interface*
GMRES	*Generalized Minimal Residual*
GPU	*Graphics Processing Unit*
HPC	*High-Performance Computing*
HLS	*High Level Synthesis*
HDL	*Hardware Description Language*
HBM	*High Bandwidth Memory*
MPI	*Message Passing Interface*
PARRSB	*Parallel graph partitioning using recursive spectral bisection (RSB)*
PE	*Parallel efficiency*
PETSc	*Portable, Extensible Toolkit for Scientific Computation*
PRACE	*Partnership for Advanced Computing in Europe*
PVC	*Precessing Vortex Core*
RCB	*Recursive Coordinate Bisection*
RCT	*Reduced computation time*
SFC	*Space Filling Curve*
SEM	*Spectral Element Method*
Spliss	*Sparse Linear Systems Solver*
SLR	*Super Logic Region*
TDP	*Thermal Design Power*
UDF	*User Defined Function*
UPC	*Unified Parallel C*
VE	*Vector Engines*
VPN	*Virtual Private Network*
ISA	*Instruction Set Architecture*

Executive Summary

The document summarizes the main progress and achievements on the development of Exascale enabling technologies on the EXCELLERAT core codes during the life of the project, but with focus on the last year. Previous reports, D3.1 and D3.2, contain detailed information about the activities conducted in year 1 and 2 respectively. The developments have been driven by the definition of individual code development roadmaps in collaboration with WP2 and WP4 to demonstrate Exascale simulations for the use-cases.

From this roadmap, several requirements were identified (see D2.1, D2.2 and D2.4 about ”Reference_Applications_Roadmap and Challenges”) and a summary of the activities conducted to address these requirements is presented here. Two fundamental activities are associated with these developments: i) Task 3.1 focused on node-level performance and ii) 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 final year, the activities carried out by the partners have been focused on the development of the application demonstrators of the use-cases, mainly by the use of GPUs, emerging technologies and the use of Adaptive Mesh Refinement (AMR). At node level (Task 3.1), analysis and optimization of the linear solver Spliss on GPU architectures was conducted by DLR for CODA, and DLB in Alya. At the system level (Task 3.2), the focus has been on strong scaling analyses and on the optimization of the communication kernels. Regarding the advanced meshing techniques (Task 3.3), most of the work has been performed on the different codes from the project. It includes m-AIA with a mesh adaptive level-set method combined with dynamic load balancing, a parallel AMR strategy based on Space Filling Curves for mesh partitioning for Alya, TREEPART with domain decomposition library with dynamic load balancing for AVBP and AMR for Nek5000 using different graph partitioners ParMETIS and PARRSB. The advances in HPC algorithms and computational methodologies presented here are part of the expertise of the EXCELLERAT consortium and compile a set of services that EXCELLERAT is delivering to the engineering community.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

The present document is a summary of the progress of Exa-enabling enhancements and benchmarks performed on the EXCELLERAT core codes during the year three of the project. The report is divided into sections that refer to the different tasks of the EXCELLERAT WP3: node and system level performance optimization, meshing techniques, emerging technologies, benchmarking and testsuite, and data transfer and dispatching. 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 Node-level performance optimization - Task 3.1

Important changes in the evolution of HPC systems are occurring at node level [1]. Consequently, the complexity associated with unlocking the intra-node performance of computing systems has increased substantially. This task addresses all the aspects related to performance at node level, including code porting and algorithm refactoring on various architectures. Subsequently, the activities carried out in T3.1 for the third year of the EXCELLERAT project are presented.

2.1 Nek5000

In the third year of the project, we focused on getting Nek5000 and its proxy app Nekbone ready for use on AMD systems. As OpenACC support using the AMD toolchain is poor, Nekbone and eventually Nek5000 was re-implemented using OpenMP for the GPU offloading, and any CUDA replaced by HIP.

The main compute kernels of Nekbone are memory-bandwidth-bound, so replacing the main compute kernel with hardcoded implementations and with careful use of shared memory and avoiding bank conflicts when using shared memory, it was possible to significantly increase the Nekbone performance. These changes should also help the OpenACC version when backported to that version. The performance of the OpenMP + HIP version of Nekbone on a single Mi100 GPU[1] is shown in Figure 1.

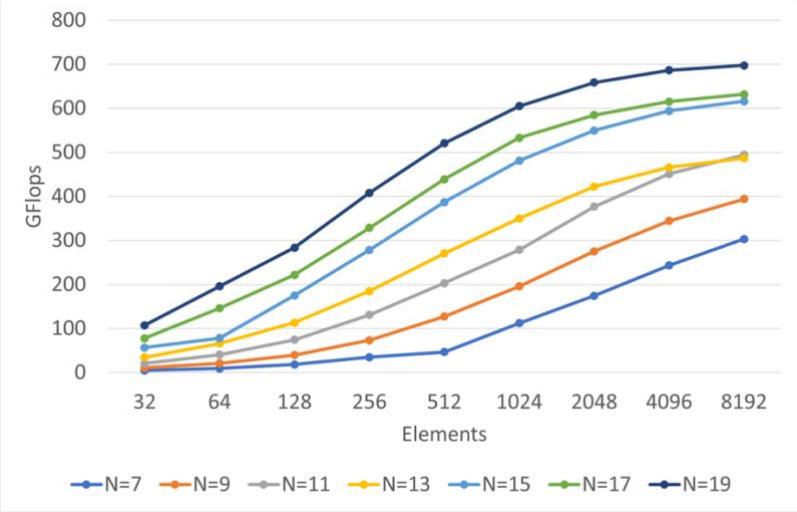


Figure 1: Performance results for OpenMP + HIP version of Nekbone on a single Mi100 GPU for

[1] https://www.amd.com/es/products/server-accelerators/instinct-mi100

different number of elements and polynomial order N.

2.2 AVBP

The activities of load balancing in AVBP during the Year 3 have been conducted in the context of ARM and GPU, and those will be described in sections 4 and 5.

2.3 Alya

The work in Alya during year 3 on node-level optimization was focused on the development and validation of a new load balancing mechanisms for chemistry integration based on the use of the Dynamic Load Balancing (DLB) library [2-3], which allows reusing CPU-cores associated with idle MPI processes by other processes running on the same node. It is a load balancing mechanism based on transferring idle resources at the node level rather than transferring workload subsets through message passing. DLB acts as an automatic runtime mechanism transparent to the user and requires minimum changes in the source code. DLB has already been successfully applied to increase the load balance for the assembly of the right-hand side terms in the Navier-Stokes equations [4] and it is extended to optimize the chemistry part in reacting flow simulations. The proposed solution with DLB, does not need to add extra data movement, because everything is done through the shared memory of the node. As it is a dynamic mechanism that reacts at the load imbalance, it does not need to predict the stiffness nor the computation load associated. And last but not least, it does not require a heavy implementation effort in the application.

The chemical integration is one of the most computationally demanding parts in the integration of the governing equations due to the high non-linearity of the Arrhenius-type reaction kinetics. It is, therefore, clear that the integration method for chemistry may play an important role in the total time for the simulation, especially when detailed chemistry models are considered.

In this work, to reduce the stiffness of the integration of the species governing equations, a splitting algorithm is used to separate the transport from the chemistry [5]. The solution of the chemistry problem is achieved by the integration of the open source Cantera software as an external library in the multiphysics code Alya. A Fortran to C++ wrapper was created to integrate Cantera[2] into Fortran for Alya, so internal functions from Cantera could be used in runtime. The reaction rates are obtained from in-built in- ternal functions from Cantera, and the chemical integration is obtained using the CVODE[3] algorithm. CVODE is a package written in C to solve IVPs (Initial Value Problems) defined by stiff and non-stiff ODEs.

In Figure 2, we show the speedup obtained by the hybrid version with and without DLB with respect to the MPI-only version (shown as a blue bar in the plot) when running in one node of MareNostrum4 for the case with detailed chemistry (left) and reduced chemistry (right). In the X axis we can see the different configurations of MPI threads and OmpSs threads to fill one node of 48 cores. For all the hybrid executions we use a grain size of 32, being the minimal size that does not show any significant overhead in this problem.

Adding DLB for the detailed chemistry integration in a counterflow flame configuration with detailed chemistry, the use of DLB generates a speedup of up to 2.3× versus the MPI-only implementation, and up to 1.5× versus the best hybrid configuration. This indicates that the use

[2] https://cantera.org

[3] https://cran.r-project.org/web/packages/sundialr/vignettes/my-vignette.html

of DLB can improve the performance and address the imbalance further than only the hybridization of the code. It is relevant that with the configuration 48 × 1, we obtain results that are not far from optimal, a speed up of 2×. For the reduced chemistry, see Figure 2 (right), it is observed that the speedup versus the MPI-only version rises up to 7×, which represents an additional 2× acceleration versus the best hybrid option. The speedup obtained by DLB in the reduced chemistry integration is higher than in the detailed one, as the load imbalance is also higher in the reduced chemistry case. Note that the speedup that DLB can obtain is related to the existing load imbalance of the application. As DLB re-distributes the tasks by the idle time from the processors, there is no need to predict the stiffness from the chemical problem as this is handled by DLB.

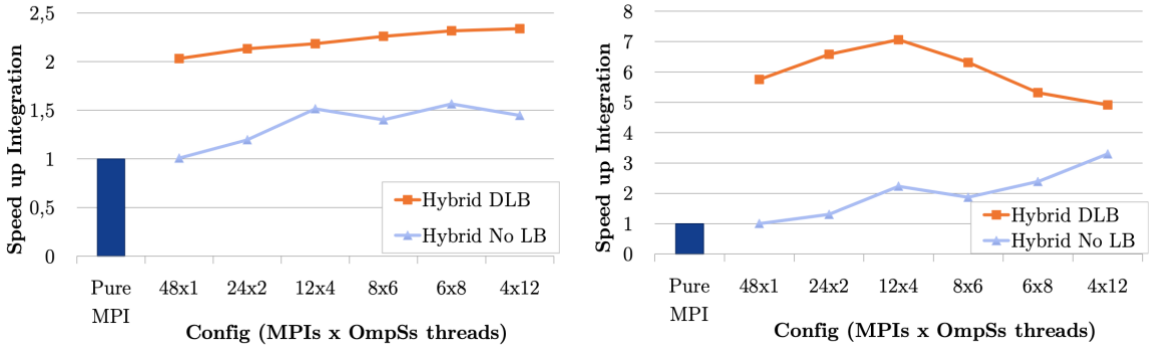


Figure 2: Comparison of time for detailed (left) and reduced (right) chemistry integration between only MPI and hybridization with and without DLB for different number of threads with grain size 32.

In Figure 3, we see the speedup (Y axis) of the chemistry integration stage normalized by the MPI-only execution in 48 cores as function of the number of MPI ranks used in the simulation. In these plots, solid lines represent contiguous binding, for which MPI ranks are placed contiguously in the nodes, while dashed lines are used to represent the Round Robin binding, where MPI ranks are spawned in a Round Robin mode among the compute nodes. MPI only executions are represented with blue lines, while hybrid DLB executions are represented with orange lines.

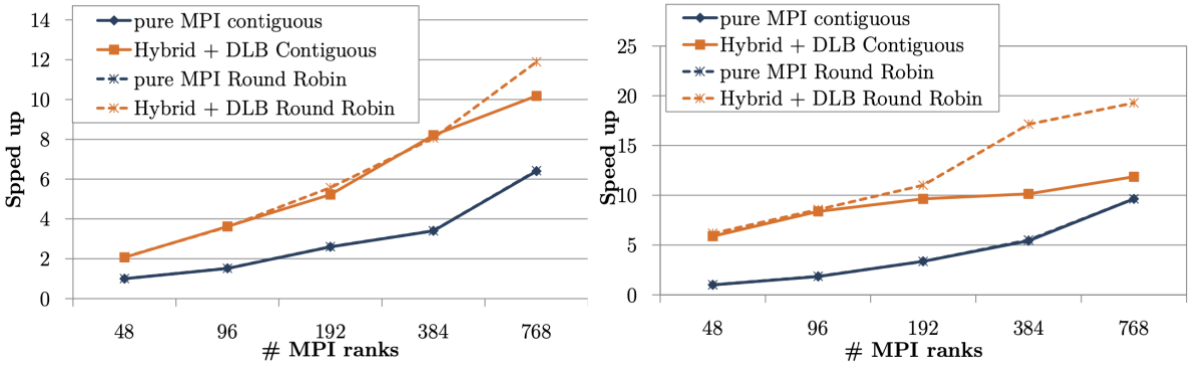


Figure 3. Speedup-up of detailed (left) and reduced (right) chemical integration up to 16 nodes using DLB and varying distribution of MPI ranks among nodes, with a configuration of 48 × 1 and grain size 32.

We can see that the binding has low impact on the performance of the pure MPI implementation (blue lines overlap). Contrarily, when DLB is used, the RR binding is helpful to break the subdomain's locality and avoid situations where the subdomains associated with processes of a node cover regions with similar conditioning. In the detailed chemistry case, see Figure 3 (left), we can see that for 2 nodes (96 MPI ranks) there is almost no difference. But for higher number of nodes DLB is able to obtain a better speedup when using a Round Robin distribution. With 768 MPI ranks, it obtains a 12× speedup with Round Robin versus a 10× speedup with contiguous distribution. Additionally, DLB improves the performance of the simulation by a factor of 2× with respect to the original pure MPI run when using 16 nodes. For the reduced chemistry case, it is observed that the distribution of MPI processes among nodes does not have an impact on the performance of the original pure MPI code (blue lines overlap), as it also occurs with the detailed chemistry case. However, we observe that the impact of the round Robin distribution in this case is even higher than for the detailed chemistry. This is due to the fact that the load was more localized in this use case, therefore, DLB benefits of distributing the most loaded processes among the different nodes. DLB achieves a speedup of 19× when running in 16 nodes (768 MPI ranks) with Round Robin, compared to the 12× speedup achieved with DLB with contiguous MPI distribution and 9× with the original pure MPI code.

2.4 CODA

During the third year of the project, the focus for CODA with regard to node-level performance optimization was on the analysis and optimization of the linear solver Spliss on GPU architectures.

Focusing on the specific task of linear-system solving within Spliss allows for integrating more advanced, but also more complex, hardware-adapted optimizations, while at the same time hiding this complexity from the CFD software CODA. One example is the usage of GPUs. Spliss enables the execution of the computationally intensive linear solver on GPUs. However, the Spliss interface design provides this capability to a user in a transparent way. By that means, CODA can leverage GPUs without the necessity of any code adaptation in CODA.

The performance analysis and optimization of CODA with Spliss running on GPUs was the main achievement during the third phase of the project. First, the numerical stability and correctness was validated. Second, the initial performance was evaluated on up to 16 Nvidia V100 GPUs. Third, a detailed performance analysis was carried out to identify potential performance bottlenecks.

After that, Spliss has been optimized and re-analyzed multiple times. Optimizations include among others the inclusion of CUDA-aware MPI and GPUDirect, the improvement of host-to-device copies and the inclusion of CUDA multi-process service (MPS) to allow more flexibility in the allocation of CPUs and GPUs.

Finally, CODA with Spliss running on GPUs was evaluated again. The current version achieves a speedup of 4.2 to 4.4 in a node-wise comparison of two Intel Xeon 6230 (40 cores) vs. 4 Nvidia V100 GPUs. This presents a speedup of 2.4 to the initial performance of the GPU porting. In addition, CODA with Spliss was evaluated on the Juwels Booster System at Julich Supercomputing Center and achieved a very good parallel efficiency of 63% on 128 Nvidia A100 GPUs.

3 System-level performance optimization - Task 3.2

This task is focused on identifying and overcoming bottlenecks at system level. Load balancing and communication/synchronization reductions are key aspects to achieve good parallel performance. Advanced features of MPI have been considered throughout the project. The developments carried out in this task include both implementation optimizations and algorithm refactoring. In the following the activities carried out in T3.2 for the third year of the EXCELLERAT project are presented.

3.1 Nek5000

System level optimizations were conducted during Year 1 and 2, and details were given in Deliverables D3.1 and D3.2 respectively.

3.2 AVBP

Strong and weak scaling in AVBP was detailed in D3.2.

3.3 Alya

Strong scaling tests for the two use-cases were reported in D3.1 and D3.2 for the case of full aircraft simulations and combustion and emissions respectively.

3.4 CODA

During the third year of the project, the focus for CODA with regard to system-level performance optimization was on the analysis and optimization of the scalability of CODA within the whole framework.

After DLR's new HPC cluster CARA[4] went operational in February 2020, CODA and the surrounding workflow were installed and intensively tested. The AMD Naples architecture introduces new characteristics that need to be considered in CODA, such as two-level NUMA domains. CODA uses classical domain decomposition to make use of distributed-memory parallelism (MPI) and additional sub-domain decomposition to make use of shared-memory parallelism (OpenMP) resulting in a hybrid two-level parallelization. Each sub-domain is processed by a dedicated software thread that is mapped one-to-one to a hardware thread to maximize data locality. Therefore, the performance of CODA was evaluated on the new architecture with particular focus on the hybrid setup of MPI ranks and OpenMP threads.

After identifying the ideal hybrid setup and adapting all workflow components to CARA, efforts were focused on improving the scalability of CODA on CARA using a test case that solves the Reynolds-averaged Navier-Stokes equations (RANS) with a Spalart-Allmaras turbulence model in its negative form (SA-neg). The test case runs on an unstructured mesh from the NASA Common Research Model (CRM[5]) with about 5 million points and 10 million volume elements. The mesh is a rather small mesh, which has been chosen for a strong scalability analysis (fixed problem size) of CODA. Production meshes are at least 20 times larger and accordingly achieve comparable efficiency on much higher scales.

Figure 4 highlights the scalability of CODA on the CARA HPC system as well as the progress that has been made during the third phase of the project. For the rather small mesh CODA achieves about 71% parallel efficiency (previously 60%) on the largest available partition on CARA with 512 nodes and 32,768 cores.

[4] https://www.dlr.de/content/en/research-facilities/hpc-cluster-en.html
[5] https://commonresearchmodel.larc.nasa.gov/

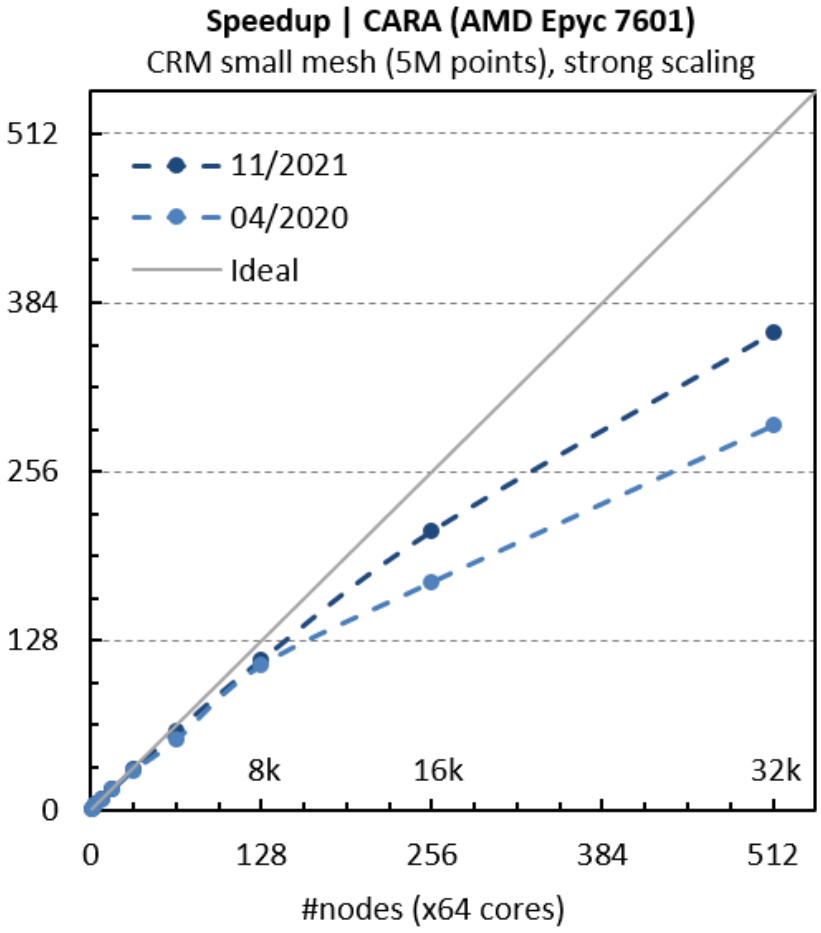


Figure 4: Scalability of CODA on CARA, DLR's HPC cluster based on the AMD Naples architecture.

4 Implementation of advanced meshing techniques - Task 3.3

This section describes the activities related to the meshing techniques that have been developed since M25. There are four partners (RWTH, BSC, CERFACS and KTH) involved in the development of the Adaptive Mesh Refinement (AMR) with their applications, namely m-AIA, Alya, AVBP and Nek5000, respectively. Note, m-AIA[6] is not currently a Reference Application.

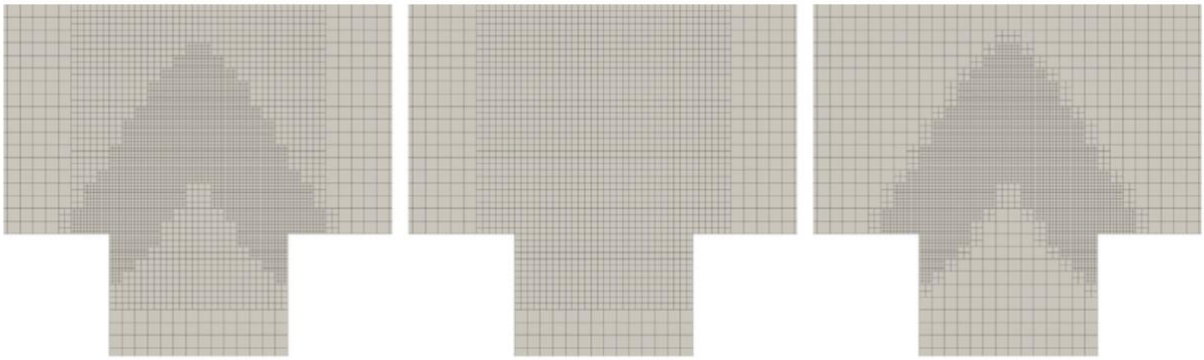
4.1 RWTH: Mesh adaptive level-set method combined with dynamic load balancing

The following describes the experience from RWTH in the field of AMR and dynamic load balancing. Although this work has not been part in the framework of code development in EXCELLERAT, it summarizes useful experience and algorithms shared with the partners. This section presents details of an AMR approach and the obtained enhancements implemented in the multi-physics simulation framework developed at RWTH, namely m-AIA. The discrete approximation in m-AIA is based on a hierarchically refined Cartesian mesh, with cells being organized in a cell-tree structure based on parent-child relations.

[6] https://www.hpccoe.eu/2021/06/04/m-aia/

The grid generation starts with a single cubic cell, which encloses the entire computational domain of all coupled systems and is the root of the cell-tree data structure. This zero-level cell is recursively refined by being subdivided into eight cube shaped child cells. Each cell can be separately refined or coarsened, regardless of the refinement level of surrounding cells. In the process of cell refinement, a cell is split into child cells and becomes a parent of these. Unrefined cells are referred to as leaf cells. The cell-tree data structure is completely stored from the leaf cells on the highest level of refinement all the way up to the cells of the isotropic start grid, i.e., parent cells are not deleted from the data structure. The data structure takes full advantage of AMR since coarsening operation can be performed without (re)creating coarse cells.

For the investigation of a turbulent lean premixed swirl flame a new numerical approach based on a finite-volume (FV) large-eddy simulation (LES) combined with a *G*-equation progress variable approach to model the combustion process using a solution adaptive level-set (LS) solver, was developed. The finite-volume and the level-set solver are parallelized and coupled on a joint hierarchical Cartesian mesh, where each solver can individually use and adapt a subset of the mesh cells (see Figure 5).



(a) (b) (c)

Figure 5: Computational mesh example for the LS and FV solver. (a) Joint mesh, (b) FV mesh, (c) LS mesh.

The LS solver adapts the mesh locally in a band close to the flame front location to automatically satisfy the high accuracy requirements of the LS solution. Various strategies exist to control the adaptation of the mesh, e.g., the AMR can be triggered due to the solution of the flow field or due to the location of a moving surface. In the turbulent flame simulation, the Cartesian LS mesh is adapted based on a distance criterion near the flame front location. By using an automatic refinement/coarsening of the LS solver to the varying flow structures the average number of required cells for the LS mesh was reduced by roughly 85% compared to a non-adapted mesh. The shape of the simulated turbulent swirl flame varies over time due to the turbulent fluctuations and especially due to a helical flow instability, i.e., a precessing vortex core (PVC), often observed in combustion chambers. Therefore, the location of the flame front in the computational domain changes dynamically and the number of LS cells, which resolve the flame front, varies on the different subdomains over time. This leads to increasing workload imbalances between the parallel subdomains unless a dynamic load balancing (DLB) method is applied periodically during the simulation.

In the newly developed approach, a DLB method is utilized to minimize load imbalances and consequently, to efficiently use all available computational resources. First, computational weights are determined for the different cell types, i.e., LS and FV cells. These weights are computed based on the measured computing times on each parallel subdomain required to complete one time step. That is, no *a priori* knowledge of the computational weights is required. Note that the measured computing times exclude any idle time that is required for communication of data between subdomains. Based on the dynamically determined cell

weights a new partitioning of the joint grid for both solvers are determined such that the workload is redistributed among the available compute cores. The impact of the new DLB method on the averaged load and the idle times, i.e., the time spent on the actual computations and the waiting time mainly due to the communication, is shown in Figure 6. It is obvious, that the discrepancies of the required overall computing time are caused by a clustering of the LS band cells on a few compute cores resulting in severe overload for the LS computation. With DLB, the peaks of the LS load are reduced considerably. The remaining idle time, i.e., the communication overhead, of about 0.42s, which is due to the iterative solution process involving communication in the LS solver required in each time step, cannot be further reduced by redistributing the compute load.

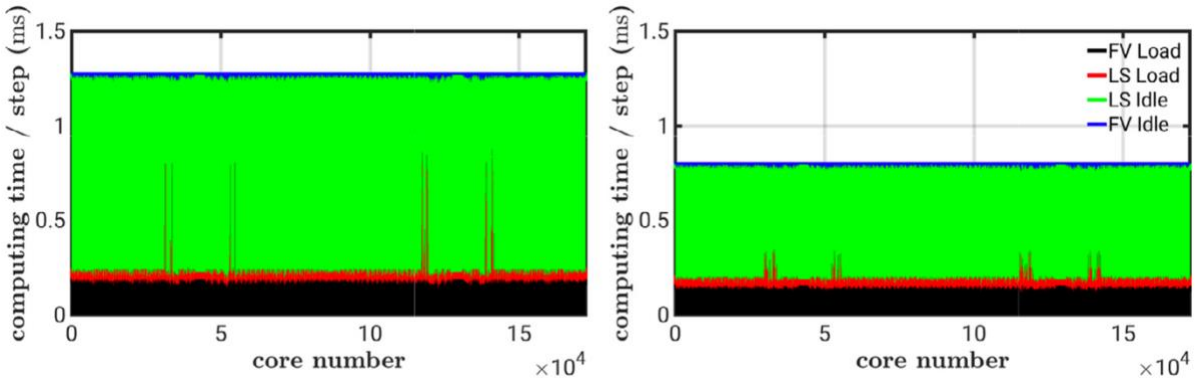


Figure 6: Average load and idle times of the finite-volume (FV) and level-set (LS) solvers for an unbalanced (a) and a balanced (b) cell distribution.

Due to the used solution adaptive mesh the simulation and the number of cells, i.e., the amount of workload, varies on the processes on which the mesh is adapted. This means that after an optimum workload distribution has been achieved the imbalance will grow again as the simulation progresses. Thus, the DLB method must be applied repeatedly to achieve an overall efficient computation. To determine the optimum DLB interval the potential performance gains have to be balanced against the time required to perform the DLB (see Figure 7).

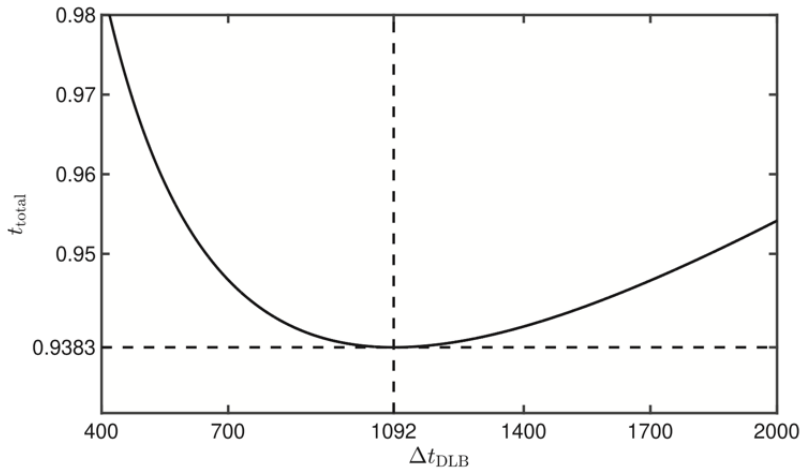


Figure 7: Estimated total average computing time required per time step total as a function of the DLB interval Δt_{DLB}.

Using the described dynamic load balancing scheme in the large-scale simulation of the swirl flame on 170,000 compute cores in the HAZELHEN supercomputer from HLRS, the load imbalances were lowered and consequently a reduction of the computing time by approximately 30% was achieved. A more detailed description of the developed AMR and DLB methods is given in [6].

4.2 Mesh adaptivity in Alya

In EXCELLERAT, BSC implemented a parallel AMR within Alya. This has required significant refactoring of the code to enable dynamic data-structures which can adapt to the variable size of the geometric discretization. The steps to migrate the simulation from one unstructured mesh into a new one is illustrated in blue in Figure 8. Firstly, the error obtained due to the former mesh is estimated in order to measure the adaptation requirements, then a new mesh is generated in parallel using MPI. Finally, the solution is interpolated from the former mesh to the new one. After this process, the domain decomposition may have become unbalanced since the refinement/coarsening is locally determined by the evolution of the simulated flow. If this is the case, a load balancing step is required, where this optional step is illustrated in green in Figure 8.

For error estimation, different strategies have been implemented. Tests revealed that basing the error evaluation on a Laplacian filter was the most robust approach. However, the error estimation is, in general, a physics-dependent aspect that requires problem-dependent specific treatment. Once the error is evaluated, a *sizing* formula, provided by the mesh generator *gmsh*[7], can be used to evaluate a *size field* as an input for *gmsh* to generate the new mesh. The size field is the target mesh size for each zone within the domain and is based on the given error estimate and the total number of elements of the mesh.

Regarding the mesh generation, as referenced above, Alya has been linked with the open source mesh generator *gmsh*. Each parallel process generates a new mesh within the boundaries of its subdomain surface. The main issue with the parallelization of the re-meshing is to deal with the mesh generation at the borders of the subdomains. Since a conformal approach has been implemented in Alya, the interface elements between subdomains must match. To ensure this, an *interface freezing* approach has been adopted. In this approach, the interface elements are not changed such that each process can adapt the rest of its subdomain without developing incoherent mesh configurations at the interface. However, this approach requires an iterative process, interleaving displacement of the interface and local remeshing, to ensure that the overall domain is re-meshed according to the adaptation requirements.

[7] https://gmsh.info

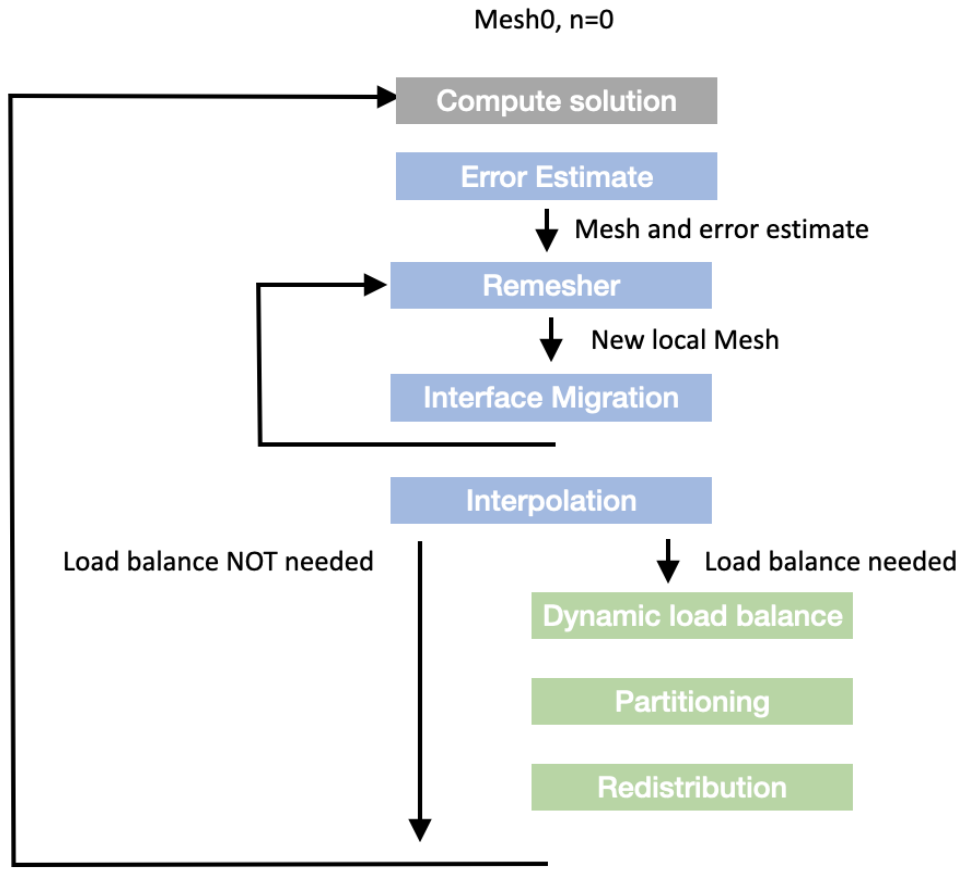


Figure 8: AMR parallel workflow.

Once a new mesh is generated in parallel, a parallel 3D interpolation is performed to migrate the solution from the former to the new mesh. This interpolation is based on point-to-point communications and the most expensive part is the evaluation of the interpolation coefficients that requires searching the element of the former mesh containing each node of the new mesh.

Regarding the load balancing, as described above, this is an optional stage that will depend on the load imbalance of the resulting mesh. A threshold is used to avoid the overhead of repartitioning the mesh when the imbalance is low. In Alya the partitioning is based on a parallel Space Filling Curve (SFC) method which can be very fast. Graph based approaches tend to provide better solutions but at a much higher cost.

All these steps have been solved during the course of the full EXCELLERAT project; however, since M25, the focus has been evaluating the performance of the different available supercomputers and also validating the results on different physics configurations.

Figure 9 shows strong scaling test performed in the Hawk supercomputer from HLRS, where Hawk is HPE Apollo platform with AMD EPYC CPUs. The mesh under consideration has 16M tetrahedra and the number of CPU-cores used ranges from 512 to 4096. The decrease of the time to solution is presented for both the AMR kernel and the overall time-step. The parallel efficiency for the overall time step using 4096 CPU-cores is 67%.

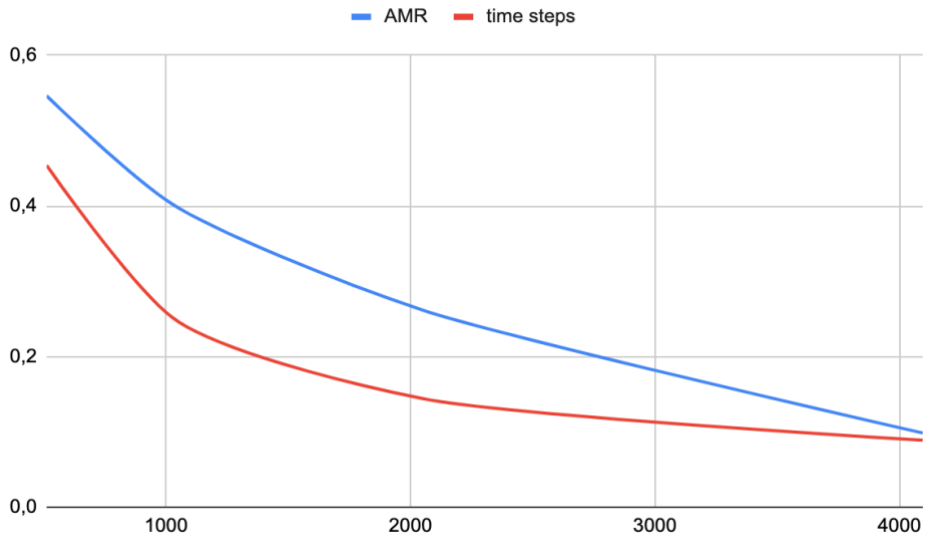


Figure 9: Strong scaling of the AMR implementation of Alya in a tetrahedral mesh of 16M elements, using the Hawk supercomputer from HLRS. Time per iteration reduction for the AMR kernels and the rest of the time step up to 4000 CPUs.

Finally, Figure 10 presents an illustrative snapshot of the mesh adaptation for the simulation of the flow around a cylinder at *Re*=120. It can be observed that the mesh is concentrated around the vortex structures of the velocity field.

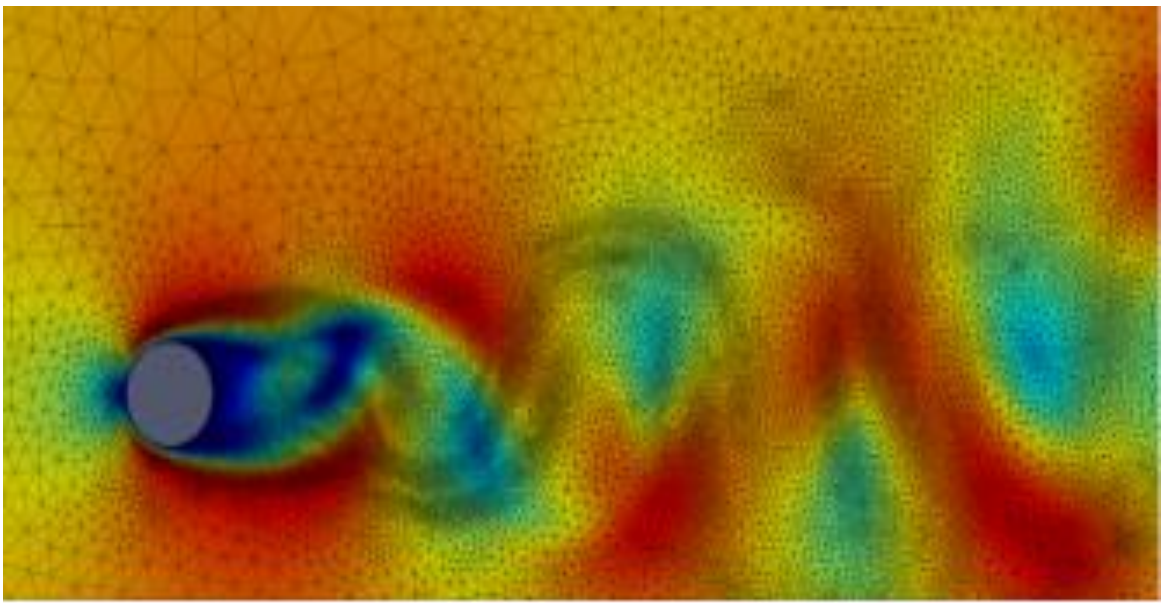


Figure 10 Mesh adaptation for the flow around cylinder

4.3 Mesh adaptivity in AVBP

In D3.2, we described the two AMR workflows developed for AVBP. One based on the CORIA-YALES2 library[8] (CORIA-CNRS), and the other an in-house library called TREEADAPT based on TREEPART [7], an internal partitioning library developed within the

[8] https://www.coria-cfd.fr/index.php/YALES2

EPEEC[9] project. Both workflows use the MMG[10] library (INRIA) as an underlying meshing application.

Since M18, we have focused on optimizing an improving, primarily, the in-house workflow and using it to validate the use case C3U1 about explosions (see D2.4).

Main efforts were concentrated on improving convergence and quality of the resulting mesh. Typically, the initial release of TREEADAPT converged within 5 or 6 adaptation steps. This was due to the need for frozen boundaries at the parallel interface to keep a watertight mesh. To reduce this convergence time, an edge weight, load balancing method was implemented, where interface edges from the current adaptation step are weighted for the next adaptation step to guarantee they will become internal nodes on the following iteration and, therefore, will be adapted accordingly to the input metric. Additionally, the input target metric field is now filtered to respect the user defined ratio between neighboring elements (graduation) to avoid conflicting inputs. With these added improvements, time to convergence reduced considerably, being reached in 2 steps on average. Our initial success story, adapting a 100 million tetrahedra case to 1.4 billion required 20 mesh adaptation steps and 30 minutes. With these improvements, convergence is reached in 7 iterations and 10 minutes on the RENE JOLIO CURIE from PRACE using 1024 cores (4 AMD nodes).

Finally, release of TREEADAPT v1.0 as opensource is expected for September 2022.

4.4 Mesh adaptivity in Nek5000

The AMR framework in Nek5000 was developed in context of two use cases C1U1 (a NACA0012 aerofoil with a rounded wing tip) and C1U3 (a simplified rotor in the rotating reference frame) performed in WP2. Lately, we focused on improving and testing the parallel performance of the nonconforming solver and we performed strong scaling test on the CPU partitions of the new supercomputers Dardel, at PDC in Sweden, and LUMI, at CSC in Finland. The results are similar for both machines and are presented in the Figure 11, which shows the initial super-linear scaling followed by the performance degradation. The super-linear scaling is caused by improved cache usage, and the performance degradation is mostly due to increasing cost of a coarse grid solver in a pressure preconditioner. The pressure preconditioner is one of the most important bottlenecks in the solver and we continued investigation of the stability of the additive Schwarz preconditioner for nonconforming meshes. We experimented as well with different graph partitioners including ParMETIS and PARRSB.

[9] https://epeec-project.eu
[10] https://www.mmgtools.org

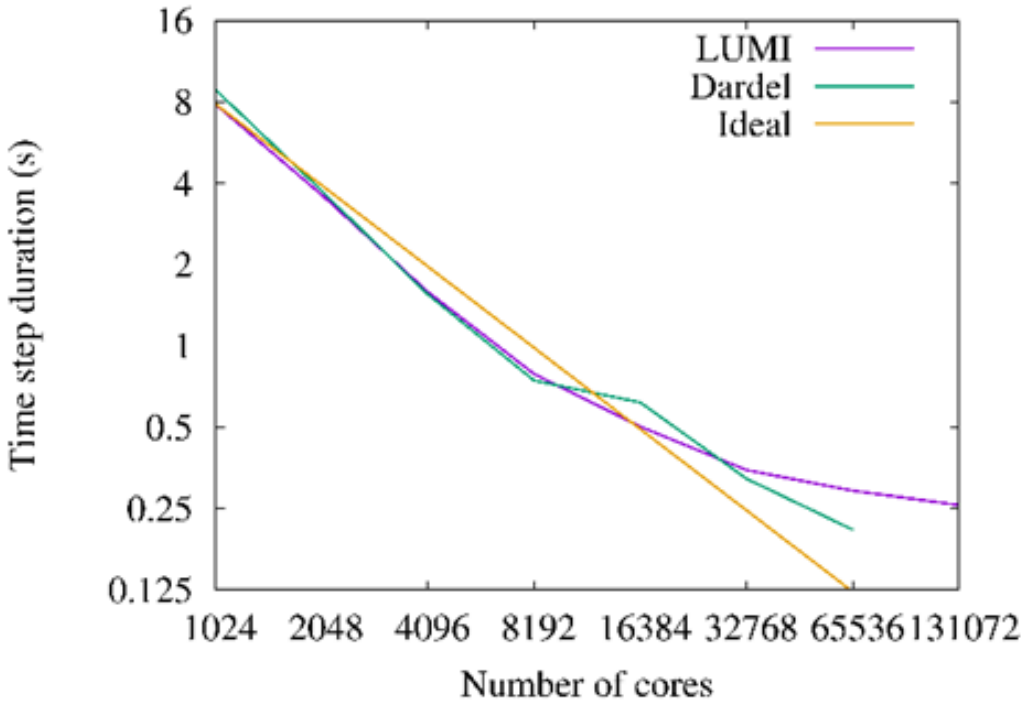


Figure 11: Strong scaling results for nonconforming Nek5000 solver performed on Dardel and Lumi using C1U3 test case.

In addition, we improved the surface projection scheme adding the support for parametrization given by set of splines. This way we can simulate more complex wing geometries not described by an analytical function. It is especially important for a realistic drone rotor simulation.

5 Test lab for emerging technologies - Task 3.4

5.1 Emerging technologies in Nekbone

This activity was for Nekbone was reported in D3.2

5.2 Emerging technologies in AVBP

For the AVBP application focus on the last period focused on the one side on GPU efficiency and portability to ARM architectures, especially GRAVITON[11] hardware from AWS. Building on a previous port for non-reactive simulations, GPU acceleration coverage was extended to the C3U2 use-case, an industrial gas turbine combustion chamber demonstrator from AKIRA technologies and SAFRAN.

In collaboration with NVIDIA, benchmarking and profiling activities were conducted and a factor 5 acceleration was obtained using 4 Nvidia A100 GPUs compared to the full 48 Core AMD Milan node on the PRACE JSC system JUWELS BOOSTER. Furthermore, strong scaling is maintained up to 1024 GPUs. Thus, a strong bottleneck has been lifted towards exascale computing for this type of simulations.

For the ARM architecture portability and benchmarking, a strong collaboration with AWS and Arm Ltd was stablished which allowed the porting and testing of the EXCELLERAT AVBP use cases on AMPERE and GRAVITON 2 architectures yielded interesting comparison on

[11] https://aws.amazon.com/ec2/graviton/

performance per $ and well time to solution per $ using AWS systems. A complete view can be found here [7]

5.3 Emerging technologies in Alya

Cache-aware Sparsity Patterns for the Factorized Sparse Approximate Inverse Preconditioner

Conjugate Gradient is one of the methods of choice for the iterative solution of symmetric and positive definite linear systems. Part of its effectiveness relies on finding a suitable preconditioner that accelerates its convergence. Factorized sparse approximate inverse (FSAI) preconditioners [8-9] are a prominent option with the advantage that, relying on the SpMV kernel for its application, they are easily parallelizable and portable to any computational architecture.

An essential element of FSAI preconditioners is the definition of the sparsity pattern where the inverse is approximated. This definition is generally based on numerical criteria. In EXCELLERAT, BSC has introduced complementary architecture-aware criteria that also increase the computational efficiency of the preconditioner. In particular, we define cache-aware pattern extensions that do not generate additional cache misses when accessing the multiplying vector.

An illustration of this idea is shown in Figure 12. Given a sparsity pattern based on numerical criteria (left), all the entries of the matrix that do not generate new cache misses on the multiplying vector are considered to extend the pattern (center), finally entries with small value are filtered out (right). The additional entries of the extended pattern do not generate new cache misses. A detailed description of this new algorithm proposed by BSC and referred as FSAIE was presented in the 30th International Symposium on High-Performance Parallel and Distributed Computing (Laut et al., 2021).

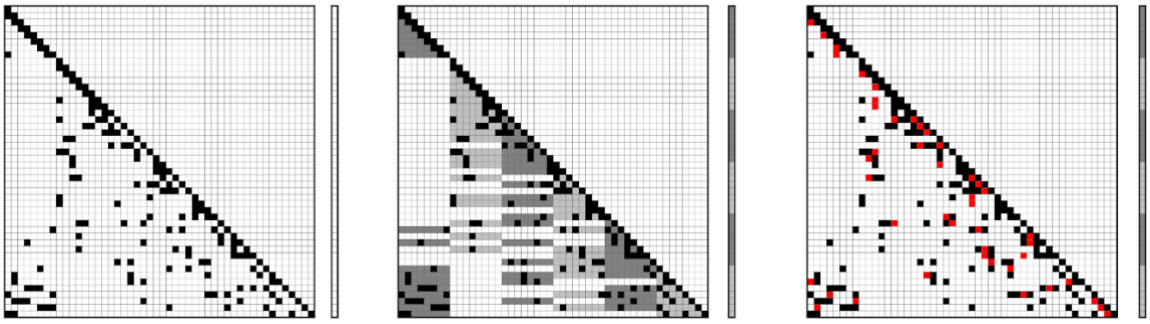


Figure 12: Graphical representation of the pattern extension strategy. Left: Initial lower triangular pattern of a given matrix, A (black squares) plus the multiplying vector x. Center: Cache-friendly pattern extension. Right: Filtered pattern.

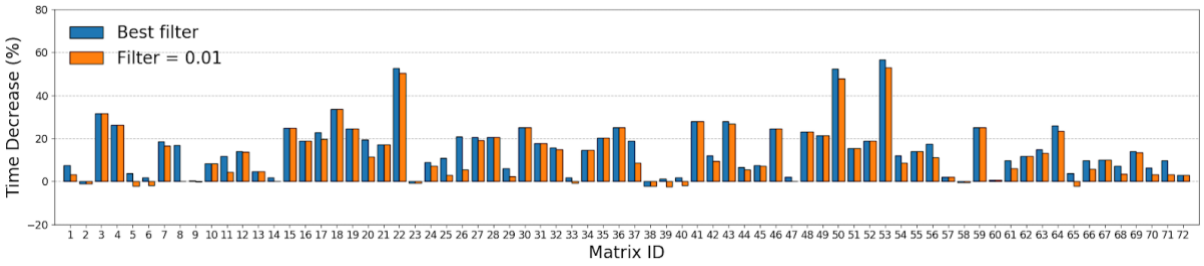


Figure 13: Time decrease of the FSAIE (full) vs FSAI using the best $filter$ value per matrix (blue

columns) and $filter$=0.01 value (orange columns) on the Skylake architecture.

An extensive evaluation campaign considering 72 matrices of the SuiteSparse Matrix Collection was performed and is shown in Figures 13 and 14. It demonstrates on the Skylake architecture, see Figure 13, average improvements of 15.02% in terms of time to solution, and time reductions of more than 50% for some matrices. We prove that the gains come from a better utilization of L1 cache level.

The FSAIE algorithm was also tested on ARM CPUs. The large 256 Bytes cache lines of A64FX produced the best results, namely, average improvements of 22.85% in terms of time to solution, and time reductions of more than 75% for some matrices, see Figure 14.

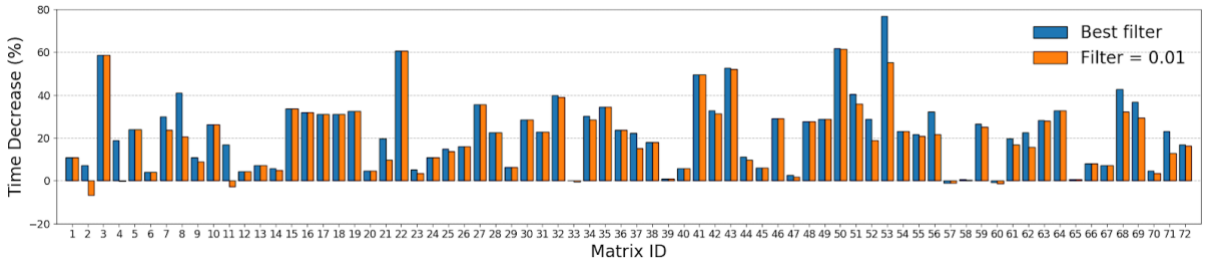


Figure 14: Time decrease of the FSAIE (full) vs FSAI for the best $filter$ value (blue columns) and for the 0.01 $filter$ value (orange columns) on the A64FX architecture.

In summary, we have observed that cache-aware optimizations produce substantial benefits across large sets of matrices and also on different architectures.

Communication-aware Sparse Patterns for the Factorized Approximate Inverse Preconditioner

In the context of distributed memory implementations, FSAIE can be applied to each process local matrix. However, FSAIE does not take into account any consideration regarding the communication pattern that the sparse matrix defines in a distributed memory scenario. In EXCELLERAT, BSC has developed FSAIE-Comm, an approach that successfully increases the efficacy of the FSAI preconditioner without introducing any significant communication overhead between the different parallel processes of the distributed memory execution. FSAIE-Comm achieves this low-overhead extension by just adding matrix entries that either correspond to the local data of each process, or involve communications between two processes for which the initial sparse pattern requires some degree of data exchange. Additionally, a method to eliminate the load imbalance that sparse pattern extensions may introduce is also proposed.

Figure 15 provides an example to illustrate the idea of FSAIE-Comm. It shows a sample 20x20 sparse matrix in a distributed memory scenario composed of 2 MPI processes. Rows belonging to the top half of the matrix are owned by one process, and bottom half rows by the other one. The light grey area depicts the two local regions, one per process. These regions represent couplings between local unknowns. The dark grey area represents the halo regions containing couplings between local and halo unknowns. Initial non-zero entries are represented by black squares. FSAIE-Comm exploits the structure of the halo area by adding additional non-zero entries corresponding to columns where there is already a non-zero halo entry, which does not increase communication costs since the corresponding x_i coefficient has to be exchanged when computing the SpMV product Ax with the initial sparse pattern S. For a symmetric and positive-

definite matrix $A = GG^T$, the preconditioning step involves two SpMV products with matrices G and G^T. Therefore, the potential halo extensions of FSAIE-Comm for matrix G are halo coefficients belonging to columns where there is already a non-zero halo entry, to avoid increasing communications when computing Gx, and halo coefficients belonging to rows where there is already a non-zero halo entry, to avoid increasing communications when computing $G^T x$. Red rectangles of Figure 4 represent halo regions where adding new entries does not increase communication costs.

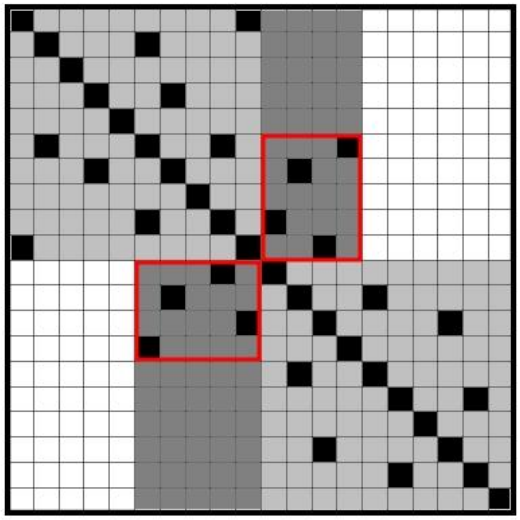


Figure 15: FSAIE-Comm. Graphical explanation of the halo region where entries can also be added in a cache-friendly communication-aware extension in a sample 20x20 matrix (red rectangles). Black squares correspond to initial entries.

Extending the preconditioning system on each MPI process independently might lead to workload imbalance in the SpMV products by G and G^T. We also propose a dynamic filtering strategy to avoid extension imbalance in distributed memory systems, opposed to the common one, which we will call static. With this method, each process computes a filtering value that leads to imbalance reduction.

We evaluate FSAIE-Comm with an extensive campaign on a heterogeneous set of 39 matrices achieving an average solution time decrease of 17.98%, 26.44% and 16.74% on three different architectures, respectively, Intel Skylake, Fujitsu A64FX and AMD Zen 2 with respect to FSAI. In addition, we consider a set of 8 large matrices running on up to 32,768 CPU cores, and we achieve an average solution time decrease of 12.59%. This recent publication [10] shows the positive effects of the dynamic filtering strategy with a case from our data set. It is shown that FSAIE-Comm outperforms FSAIE in all cases and we demonstrate FLOP/s and L1 cache miss reduction on accesses to multiplying vector for different configurations.

5.4 Emerging technologies in Alya: testing on FPGAs

In Year 3, the focus was given to Alya's *nastin* module, which solves the incompressible Navier-Stokes equations and is a key component of many model-run configurations. In this module the building of the matrix which is used to solve Navier-Stokes equations is a costly operation, for instance representing 64% of the overall model runtime for a representative benchmark. When profiling the matrix assembly code with Intel VTune we found that it was

stalling 32% of the time due to memory accesses and 11% due to other microarchitecture core-bound issues. Therefore, an important question is whether on the FPGA, by designing memory access in a bespoke manner, we could ameliorate these issues imposed by the general-purpose CPU architecture.

For this work we used a Xilinx Alveo U280 FPGA which also contains 8GB of High Bandwidth Memory (HBM2) and 32GB of DDR DRAM on the board, although for this work we used the HBM2 exclusively. The FPGA card is hosted in a system with a 26-core Xeon Platinum (Skylake) 8170 CPU. Codes for the Alveo are built with Xilinx Vitis framework version 2021.1. For comparison we run the code on a 24-core Xeon Platinum (Cascade Lake) 8260M CPU, and Nvidia Tesla V100 GPU. On the CPU, the code has been parallelised via OpenMP (using GCC 8.3) and on the GPU it uses OpenACC (using Nvidia compiler version 20.9). All reported numbers are averaged over three runs.

Table 1 illustrates the performance, for the Sphere 100K Alya benchmark, for different versions of our FPGA kernel as we optimised it. We include the kernel only execution time, the time taken for data transfers between the host and FPGA, and total execution time which is a combination of these first two measures. For reference the table also includes performance of the code running on both a single core of the CPU and the Xeon's entire 24 cores. For each FPGA configuration we provide the percentage of DSP and LUT resources required. This utilisation is reported on a Super Logic Region (SLR) basis, with three SLRs present on the U280. Details of our initial design is reported in Table 1 as *Initial FPGA dataflow design*, and achieving only 0.39% the performance of the 24-core CPU it can be seen that significant optimisations were required.

Description	Total execution time (ms)	Kernel execution time (ms)	Host-FPGA data transfer time (ms)	% CPU performance	% SLR DSP usage	%SLR LUT usage
1 core of Xeon CPU	351.05	351.05	-	-	-	-
24 cores Xeon CPU	61.72	61.72	-	-	-	-
Initial FPGA dataflow design	15714.99	15701.78	13.21	0.39%	15%	13%
Optimised II of loops	1508.60	1495.62	12.98	4.09%	91%	40%
Brought elements loop into DF functions	293.21	279.79	13.42	21.05%	91%	44%
Refactored code into engine	284.04	270.71	13.33	21.73%	97%	48%

Table 3: Architectural view of how the host, HBM2, and IP blocks interact with the streaming design, where chunks of data in the format required for the Alya incompressible flow matrix assembly engine are streamed onto the FPGA and results streamed back.

One of the major reasons for this initial poor performance was that our dataflow design was susceptible to deadlock, where if streams were written to in a different order than they were consumed from then the FPGA could hang. Furthermore, HLS automatically reorders stream writes and reads based upon dependencies in the code, so carefully laying out and manually ordering the stream accesses in code does not necessarily solve the problem. Therefore, whilst we had pipelined as many loops as we could, there were a number where it was not possible to pipeline due to this deadlocking and instead sub-loops were pipelined. This meant that the functions in our dataflow machine were rather asymmetrical in their pipelined behaviour, where some were pipelined on the outer loop, whereas others only pipelined on inner nested loops. As such it meant that, from a performance perspective, some stages had a tendency to stall preceding or subsequent stages, effectively reducing the amount of concurrency present in the design.

Whilst one approach would be to leverage HLS's *protocol* pragma, which enforces HLS not reordering contained code, this would still require careful manual ordering of accesses in code by the programmer. Instead, we increasing the stream FIFO depth in the *HLS STREAM* pragma, and also disallowed streams jumping ahead of subsequent stages. This involved routing streams through subsequent dataflow functions, irrespective of whether they utilised that streaming data or not in their calculations.

However, it was not just the issue of deadlocking that was limiting the pipelining of loops. In HLS when a loop is pipelined then all contained inner loops must be completely unrolled. In our code there are numerous nested loops and unrolling all these could result in significant DSP usage on the FPGA. Under certain conditions, for instance if there are no statements between outer and inner loops, then HLS can merge these automatically when the pipeline pragma is applied to the inner loop. In this manner, for some functions in our matrix assembly engine, we were able to limit the amount of unrolling required whilst still achieving a pipelined outer loop. However, for others this was not possible and the increased DSP usage had to be accepted.

Furthermore, functions contained a spatial dependency which was critical to fix because these account for a large portion of the overall matrix assembly runtime. A spatial dependency is where the calculations involved at one cycle depend on previous calculations which might not yet have completed. In our specific case there were accumulations on data, and these double precision additions required seven cycles to complete. Therefore, the pipelined loop is limited to an Initiation Interval (II), the number of cycles before the next value can start to be processed, of seven because this number of cycles must elapse before the next value can start to be processed due to the dependence on the previously accumulated value.

It is possible to address the spatial dependency via the *dependence* HLS pragma, because whilst the HLS compiler cannot guarantee there are enough cycles between one outer loop iteration and the next due to the dynamic indexes being calculated, this is obvious to the programmer. These optimisations enabled data to flow more effectively between the dataflow stages, improving the overall concurrency of our design and the performance benefits are illustrated by the entry *Optimised II of loops* in Table 1. It can be seen that this significantly improves the performance of our kernel, with it running over 10 times faster, but at a higher resource usage cost, where DSP usage has increased from 15% of an SLR in the previous version to 91% now, and LUT usage has also increased, albeit at a lower rate. However, the kernel was still only achieving around 4% of the CPU's performance, so clearly there were still further optimisation opportunities.

At this point the loop over the number of elements was outside the dataflow pragma, meaning that between each element the dataflow stages had to shut down and restart which resulted in overhead. To address this, we brought the elements loop inside each of the dataflow stages, ensuring that each dataflow stage could run continually from one element to the next. Moving the elements loop inside each dataflow region and ensuring that this loop fusion occurred sped the kernel up over five times, as reported by *Brought elements loop into DF functions* in Table 1, although the FPGA was still only achieving around 21% of the CPU's performance. We also refactored the kernel to better structure the computation, into an engine, and separate out the loading and writing of data. This is represented by *Refactored code into engine* in Table 1, and whilst this did not significantly improve performance it did help underly the activities that were to come next.

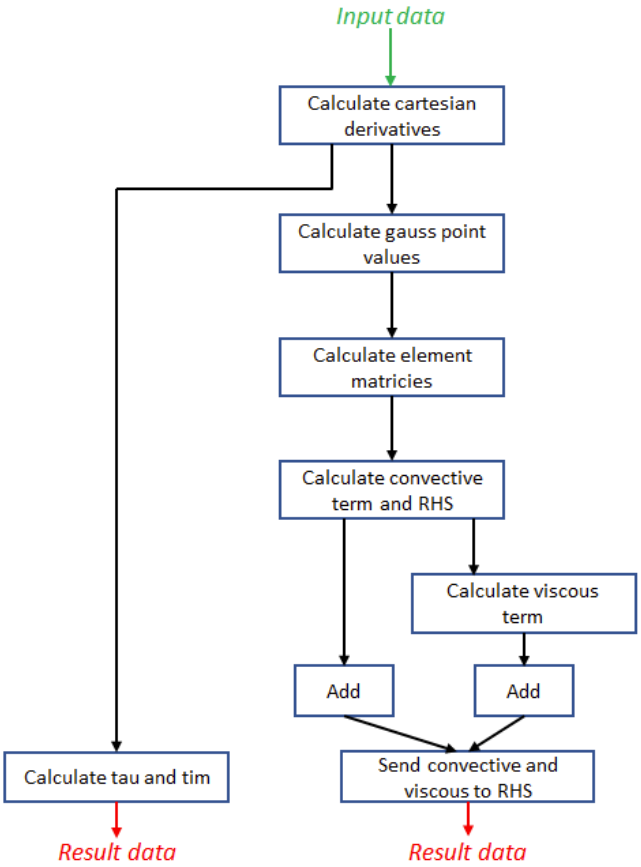


Figure 16: Revised matrix assembly engine dataflow design based on optimisations discussed in this section, most notably streams are always routed through subsequent stages regardless of the consumption of data from them.

Figure 16 illustrates the final dataflow design of our approach at this stage, however whilst we had spent considerable time optimising the computational engine of our design to ensure that it could continually stream data, performance was still falling significantly short of that delivered by the 24-core Xeon Platinum CPU. An important question was whether the design was being most efficiently fed with input data from, and results being delivered to, the HBM2 external memory. Put simply, whether the engine was stalling due to a lack of input data or stalling because it was unable to stream out results due to overhead on writing. The way the code was written resulted in two major disadvantages for performance, firstly there was a spatial dependency for result data, as the accumulation of array elements could be followed by an accumulation into that same element for the next node, resulting in an initiation interval of 7 imposed by the tooling, and secondly accesses to external memory were not contiguous.

For each separate external memory access, the HLS tooling has to add an explicit read request for input data and write response for output data, both costing 69 cycles. For contiguous external memory accesses, the compiler can fuse accesses together, effectively meaning that there is one of these expensive operations for many individual accesses. To address is we reorganised our code to adopt a more streaming approach, where input data is loaded and prepared on the host before being transferred over to the FPGA in chunks which are then fed into the matrix assembly engine. Result data is handled in a similar fashion, with chunks streamed out by our kernel and then when a chunk is ready it is transferred back to the host and handled accordingly.

Figure 17 provides a general architectural illustration of this streaming approach, where the matrix assembly engine is central but connected to a streaming input IP block and two streaming

output IP blocks that we had to write. The host and device communicate via the high bandwidth memory, with the host still sending data over PCIe.

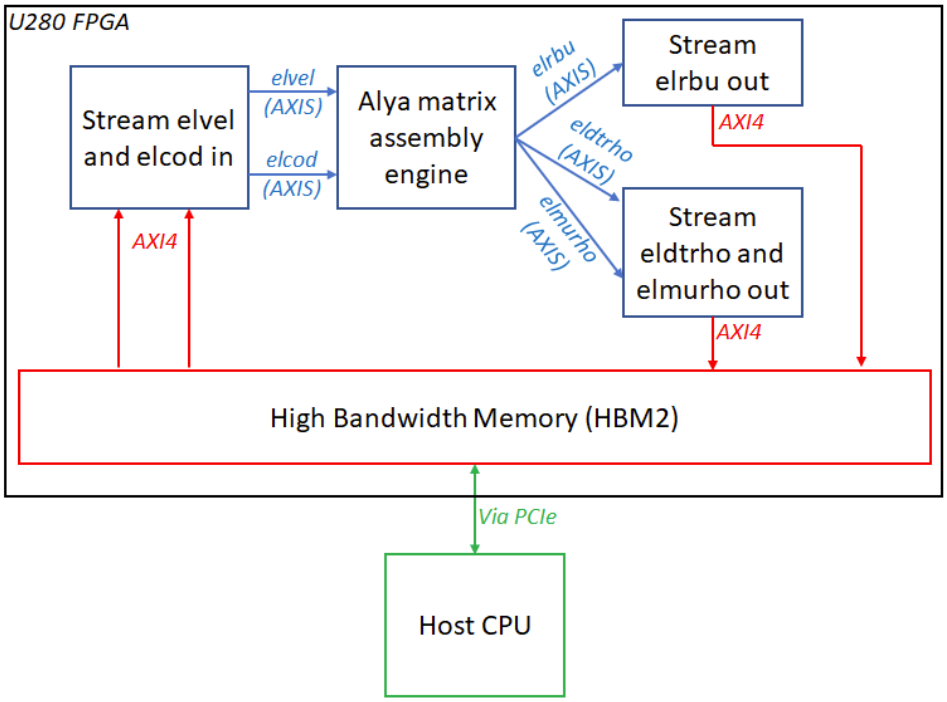


Figure 17: Architectural view of how the host, HBM2, and IP blocks interact with the streaming design, where chunks of data in the format required for the Alya in-compressible flow matrix assembly engine are streamed onto the FPGA and results streamed back.

The performance of this revised approach is reported in Table 2, where *initial streaming approach* is the first attempt at this and it can be seen that this is significantly faster than the previous non-streaming version of the kernel. However, there were further optimisations that could be applied, for instance where we pack data together into a single 512-bit width packet and undertake a read of 512-bit per cycle, thus ensuring that data is continually fed into the engine(s) and this is reported by *data streamed each cycle* which for the first time out-performs the 24-core Xeon Platinum Cascade Lake CPU. Lastly the *threaded result handling entry* in Table 2 is where the handling of results arriving on the host is threaded and-so provides concurrency for unpacking the data and placing it into the appropriate location of memory. This further improved performance of the FPGA, now out-performing the CPU by more than two times.

Description	Execution time (ms)	% CPU performance
24 cores Xeon CPU	61.72	-
Previous non-streaming FPGA	284.04	21.73%
Initial streaming approach	97.91	63.04%
Data streamed each cycle	48.19	128.08%
Threaded result handling	26.67	231.42%

Table 4: FPGA matrix assembly performance for Sphere 100K benchmark on Alveo U280 for our data streaming approach. Compares against the previous non-data streaming FPGA design and 24-core Xeon Platinum CPU.

6 Validation and benchmarking suites - Task 3.5

This activity was reported in D3.2.

7 Data dispatching and data transfer - Task 3.6

Like in the previous deliverables, SSC is combining their two work package efforts “Data dispatching through data transfer” from Work Package 3 and “Data Management” from Work Package 4 into one deliverable, which also fits to their motivation in combining data transfer and data management into their newly designed data exchange platform. The detailed contribution can be found in deliverable D4.6 and the final report in 4.7.

8 Conclusions and outlook

As a conclusion, the progress on the development of Exascale enabling technologies on the EXCELLERAT core codes for the third year of the project has been presented. While in Year 1 most of the work was dedicated to node-level and system-level performance optimizations, this final year substantial effort was dedicated to load balancing, AMR and emerging technologies, mainly GPUs. The activities carried out by the partners on these tasks have been focused on running on Nvidia GPUs ARM, and FPGAs and working on parallel solvers and preconditioners. Finally, aspects related to intra-node parallelization, such as load balancing and OpenMP threading optimizations, have been considered. At the system level, the focus has been the strong scaling and optimizing the communication kernels. Additional focus has been given to improving the strong scaling of the codes and designing and implementing new distributed memory load balancing strategies. The data transfer and dispatching strategy has been extended outside the project consortium to the medical sector.

The developments presented here along with the demonstrators based on the use-cases described in D2.4 evidence a clear progress to bringing the engineering world closer to exascale. These activities are the central part of the technical core of EXCELLERAT and intimately connected to the applications in WP2 and the services in WP4. These advances in HPC algorithms and computational methodologies are the building blocks not only for the use-cases described in EXCELLERAT, but also beyond and could be applied to other applications of the engineering realm. These advances in HPC technologies for Exascale are part of the expertise of the EXCELLERAT consortium and are ultimately defined as services that EXCELLERAT can deliver to the engineering community. This WP has contributed with the following services to the EXCELLERAT services portfolio, further details are given on the EXCELLERAT service portal[12]:

- Co-Design Engineering Software- and System-Design.
- Data management for large scale simulation result and input data.
- Efficient and modern implementation of Exascale ready engineering software.
- Efficient execution of large-scale engineering simulation workflows.
- Holistic Testing and Validation for the Engineering Workflow.
- Meshing and re-meshing techniques, methodologies and Software.
- Modelling of Engineering Problems.
- Numerical Solution methods for Engineering Problems.
- Performance Engineering for the Complete Large-Scale Engineering Workflow.

[12] https://services.excellerat.eu

- Strategies for Load-Balancing and Data-distribution.

As an outlook on future developments beyond the project timeline, futher work on Nek5000 will focus on both the solver and data analysis. It will require an evaluation and improvement of the parallel performance of AMR version of Nek5000 further developing the pressure preconditioner for nonconforming meshes, and modernising communication kernels merging interpolation operator with communication step. Moreover, multiple new features including immersed boundary method (IBM), wall-modelling and streaming data reduction algorithms are to be developed. That is why a special attention will be paid to the proper coupling of Nek5000 with external solvers and Catalyst for in-situ operations.

Regarding CODA, CODA CFD software and the workflow framework FlowSimulator have achieved a high level of code maturity and usability during the project and are about to enter into service and replace the predecessor TAU (developed by DLR) in production in the European aircraft industry, research organizations and academia, most prominently at DLR for aerodynamics data production and validation and at the Airbus Group. The final steps to release CODA into production will be one of the main next steps.

In addition, CODA will be continuously analyzed and improved in numerical as well as computational performance. CODA will be adapted and evaluated on new architectures and systems (e.g. DLR's new HPC system based on the AMD Rome architecture) as well as emerging hardware technologies.

Regarding Alya, the work will be focused on the development of multiphysics simulation platform that integrates advanced simulation techniques in the CFD codes and massively parallel workflows with the capabilities of leading-edge HPC architectures. The accomplishment of such simulations involves different level of hierarchies and parallelism in the algorithms responsible to control the different tasks. This requires the revision of the load balancing strategies for inter- and intra-nodes in the CFD, but also communication with the other tasks during post-processing and data analytics. A multicode strategy will be used to perform coupled multiphysics simulations, which brings additional challenges to ensure not only high computational performance, but also flexibility and portability. The I/O will be adapted to facilitate the reading/writing of files during runtime and parallel algorithms will be used to communicate the data.

Finally, regarding AVBP; during the project a new milestone was reached with the release of a fully accelerated version for typical simulations using NVIDIA GPUs. Focus will shift to increasing the coverage of the GPU acceleration to other complex workflows like the particle tracking and detailed chemistry models as well as using AMR and GPUs. This requires major analysis of the data existing data structures for efficient acceleration both inter and intra node. Also bringing this version closer to the user by simplifying the workflow when using GPUs will be a key topic to ensure its acceptance in the community.

All these aspects are to be addressed in the future EXCELLERAT-II in case the project is finally funded.

9 References

- [1] Susheel Tadikonda, Scott Durrant, Scott Knowlton, Ruben Molina**, Trends driving the future of high-performance computing (HPC), https://www.embedded.com,** January 19, 2022.
- [2] Marta Garcia, Julita Corbalan, and Jesus Labarta. Lewi: A runtime balancing algorithm for nested parallelism. In *2009 International Conference on Parallel Process- ing*, pages 526–533. IEEE, 2009.
- [3] Marta Garcia, Jesus Labarta, and Julita Corbalan. Hints to improve automatic load balancing with lewi for hybrid applications. *Journal of Parallel and Distributed Computing*, 74(9):2781–2794, 2014.
- [4] Marta Garcia-Gasulla, Guillaume Houzeaux, Roger Ferrer, Antoni Artigues, Victor López, Jesús Labarta, and Mariano Vázquez. Mpi+ x: task-based parallelisation and dynamic load balance of finite element assembly. *International Journal of Computational Fluid Dynamics*, 33(3):115–136, 2019.
- [5] Z.Ren and S.B.Pope. Second-order splitting schemes for a class of reactive systems. *Journal of Computational Physics*, 227(17):8165–8176, 2008.
- [6] S. Herff, A. Niemöller, M. Meinke, W. Schröder, "LES of a turbulent swirl flame using a mesh adaptive level-set method with dynamic load balancing", Computers & Fluids, **221** (2021).
- [7] P. Mohanamuraly and G. Staffelbach. 2018. Hardware locality-aware partitioning and dynamic load-balancing of unstructured meshes for large-scale scientific applications. In PACS '20: Platform for Advanced Scientific Computing, June 29–July 01, 2020, Geneva, Switzerland. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456
- [8] Edmond Chow. 2000. A Priori Sparsity Patterns for Parallel Sparse Approximate Inverse Preconditioners. SIAM Journal on Scientific Computing 21, 5 (2000), 1804–1822.
- [9] Edmond Chow. 2001. Parallel Implementation and Practical Use of Sparse Approximate Inverse Preconditioners With a Priori Sparsity Patterns. International Journal of High Performance Computing Applications 15 (05 2001).
- [10] Sergi Laut, Ricard Borrell, and Marc Casas. 2021. Cache-aware Sparse Patterns for the Factorized Sparse Approximate Inverse Preconditioner. In Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21). Association for Computing Machinery, New York, NY, USA, 81–93.