D4.1– First report on methodology development and tool improvement Version 1.0

Document Information

Contract Number	101143931
Project Website	www.pop-coe.eu
Contractual Deadline	M12
Dissemination Level	PU
Nature	R
Authors	Joachim Jenke (RWTH), Ondrej Vysocky (IT4I@VSB), Germán Llort (BSC), Marc Schlütter (JSC), José Morgado (INESC-ID)
Contributors	Judit Giménez (BSC), David Bernal (BSC), Brian Wylie (JSC)
Reviewers	Xavier Teruel (BSC)
Keywords	Performance analysis, Energy efficiency, Correctness

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101143931. The JU receives support from the European Union's Horizon Europe research and innovation programme and Spain, Germany, France, Portugal, and the Czech Republic.

Change Log

Version	Author	Description of Change
v0.1	Ondrej Vysocky (IT4I@VSB)	Initial version of the POP LaTeXtemplate
v0.2	Joachim Jenke (RWTH)	Document structure
v0.3	Germán Llort (BSC)	Adds BSC Tools content
v0.4	Marc Schlütter (JSC)	Score-P/Scalasca/Cube content
v0.5	Joachim Jenke (RWTH)	POP methodology extension
v0.6	Ondrej Vysocky (IT4I@VSB)	Tools task collaborative work report
v0.7	Ondrej Vysocky (IT4I@VSB)	MERIC tools development content
v0.8	José Mogado (INESC-ID)	CARM tool content
v1.0	Ondrej Vysocky, Joachim Jenke	Finalised document based on internal review

Contents

Executive Summary

This document presents an extension to the POP methodology and development done in POP tools during the first year of the POP3 project. Moreover, it reports the status of the POP3 flagship codes deployment to EuroHPC systems status, and activities we did to support the CASTIEL project.

1 Introduction

The objective of POP is to analyze and quantify the performance of parallel applications. A key characteristic of parallel applications is the parallel performance. Figure 1 displays the hierarchy of POP metrics, as defined in phase 1 of the POP project. The idea of using a metrics hierarchy to understand parallel performance issues is immensely powerful, as it allows users to immediately see which issue or issues are impacting performance, e.g., poor computational scaling versus inefficient parallelism. In particular, a hierarchy where top-level metrics are split into individual child metrics allows users to drill down and quickly get a detailed understanding of the relative importance of a range of issues. The hierarchical view of metrics also helps the user to focus on the most severe performance issue of a code. As also shown in the figure, the child metrics in this hierarchy multiply to get the parent metric.

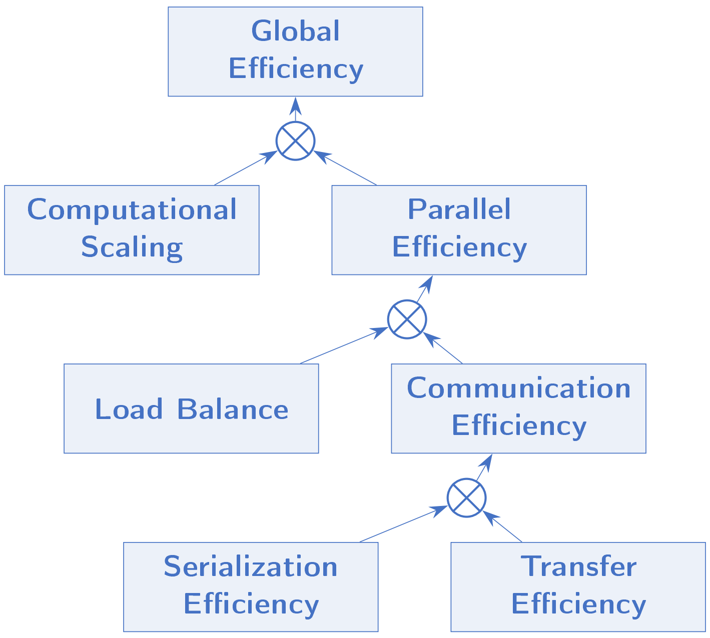


Figure 1: Hierarchy of POP metrics.

In textbooks, we can find a typical definition of parallel efficiency:

$$parallel\ efficiency = \frac{serial\ runtime}{parallel\ runtime \times execution\ units}$$

In the definition of parallel efficiency in POP, we assume that serial runtime is equal to the useful computation time, measured as the execution time outside of parallel runtime implementation (e.g., MPI runtime library calls). Starting from

$$parallel\ efficiency = \frac{avg(useful\ computation)}{parallel\ runtime}$$

we break down into the factors

$$load\ balance = \frac{avg(useful\ computation)}{max(useful\ computation)}$$

and

$$communication\ efficiency = \frac{max(useful\ computation)}{parallel\ runtime}.$$

In this document, we first describe our extensions made to these metrics. We consider different ways to visualize the calculated hierarchy of metrics.

To support the increased use of GPUs accelerating the computation in more or less hybrid settings, we consider different extensions to the existing methodology, aiming to describe these execution setups best. In this document, we cover the most common scenarios of hybrid MPI+GPU setups and describe open questions about how to handle specific corner cases not yet covered.

Since we rely on performance analysis tools to collect and provide the data for POP audits, we describe our development efforts that have been done to extend our existing analysis tools in the second part of this document. This includes the applicability to previously not supported features of parallel programming paradigms as well as the integration of our proposed extensions to the POP methodology to enable the POP services to use them.

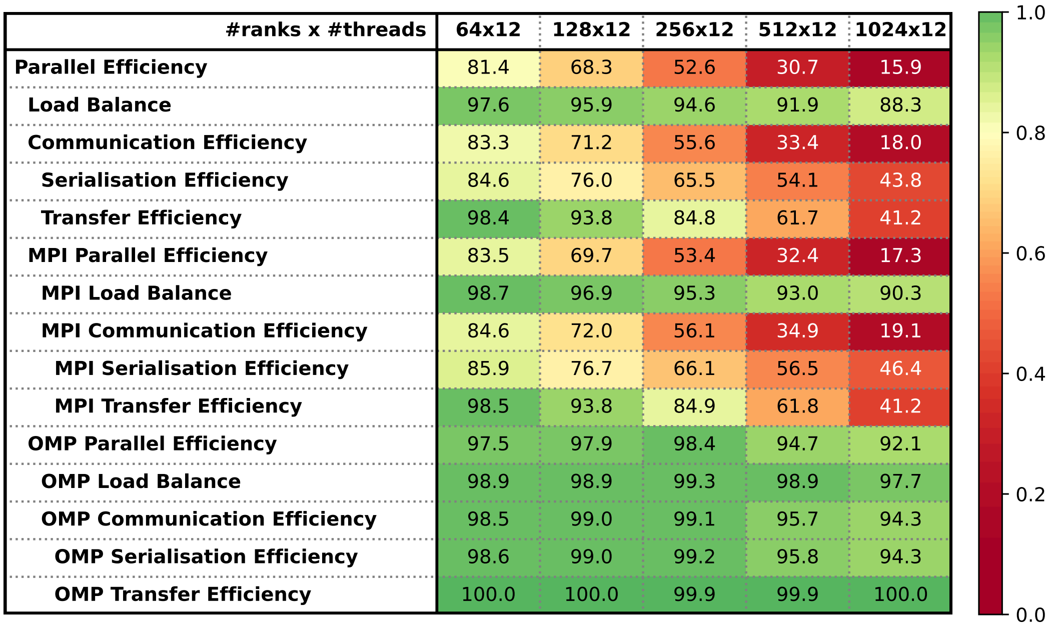
2 Extensions made to the methodology

2.1 Visualization of POP metrics

When reporting performance results to the application developer, presentation of the data is important. We developed different views for the same set of data that we will compare in the following.

2.1.1 Table of hierarchical efficiencies

The table in Figure 2 presents the concrete metric numbers of all efficiencies. The first five rows show the hybrid efficiency values of the hierarchy introduced in Figure 1. The following rows show the same hierarchy for MPI and OpenMP. The coloring of the cells guides the reader toward sources of the highest inefficiency. The colors for values above 75 % scale from green to yellow, while the colors for values below 75 % scale from yellow to red.



#ranks x #threads	64x12	128x12	256x12	512x12	1024x12
Parallel Efficiency	81.4	68.3	52.6	30.7	15.9
Load Balance	97.6	95.9	94.6	91.9	88.3
Communication Efficiency	83.3	71.2	55.6	33.4	18.0
Serialisation Efficiency	84.6	76.0	65.5	54.1	43.8
Transfer Efficiency	98.4	93.8	84.8	61.7	41.2
MPI Parallel Efficiency	83.5	69.7	53.4	32.4	17.3
MPI Load Balance	98.7	96.9	95.3	93.0	90.3
MPI Communication Efficiency	84.6	72.0	56.1	34.9	19.1
MPI Serialisation Efficiency	85.9	76.7	66.1	56.5	46.4
MPI Transfer Efficiency	98.5	93.8	84.9	61.8	41.2
OMP Parallel Efficiency	97.5	97.9	98.4	94.7	92.1
OMP Load Balance	98.9	98.9	99.3	98.9	97.7
OMP Communication Efficiency	98.5	99.0	99.1	95.7	94.3
OMP Serialisation Efficiency	98.6	99.0	99.2	95.8	94.3
OMP Transfer Efficiency	100.0	100.0	99.9	99.9	100.0

Figure 2: POP efficiency metrics at different scales for a specific application use case.

2.1.2 Line diagram

The diagrams in Figures 3-5 present the values from Figure 2 for the global, MPI and OpenMP hierarchy of metrics. This representation directly presents all values as they are. If values are very similar, some data points can be hard to spot as they overlap or are hidden behind another data point. We do not show a diagram with all data points plotted at once because such a diagram becomes too convoluted.

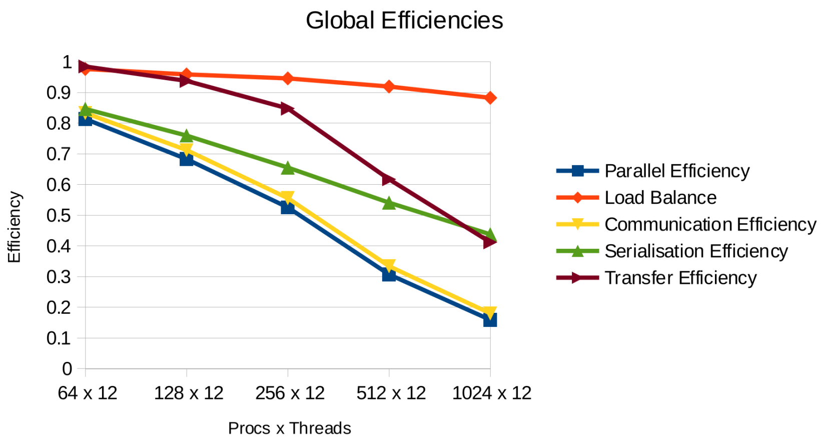


Figure 3: Global efficiency metrics.

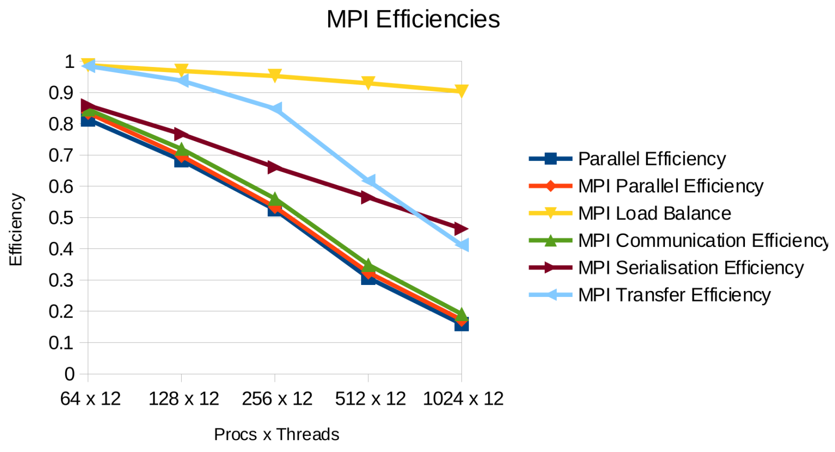


Figure 4: MPI efficiency metrics.

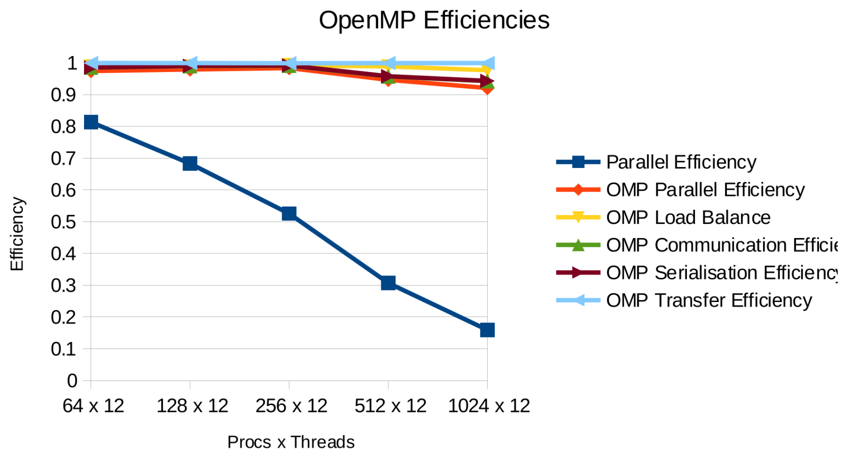


Figure 5: OpenMP efficiency metrics.

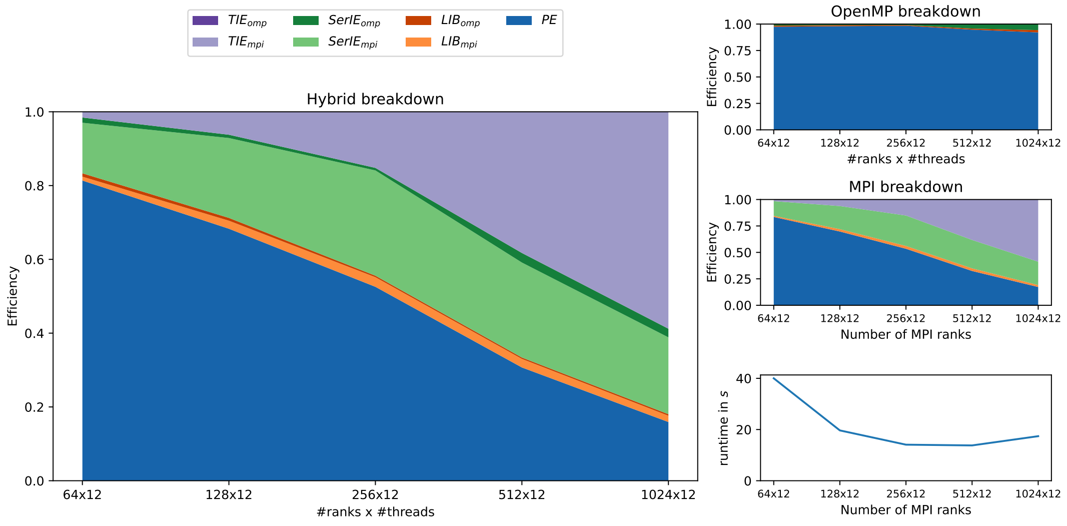


Figure 6: Parallel Efficiency with stacked impacts from different inefficiencies.

2.1.3 Stacked inefficiency plot

Figure 6 presents the same data in a slightly different view. The figure shows four diagrams at once. The diagram at the lower right plots the total runtime at each scale and, in this case, shows increasing execution time when going from 512 to 1024 processes. The other three diagrams show a hybrid breakdown of efficiencies on the left and an OpenMP and MPI breakdown on the right. In all three diagrams, the blue area represents the parallel efficiency (global(=hybrid), OpenMP, MPI) at different scales. The remaining areas correspond to inefficiencies.

Figure 7 explains how the metrics are stacked. All values relate to the $1024x12$ experiment,

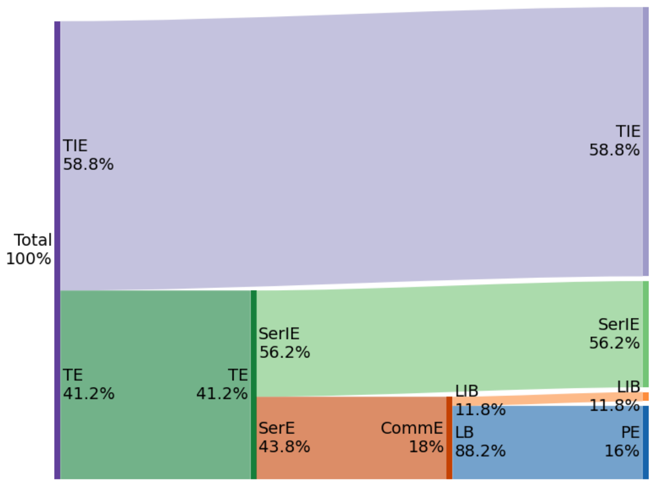


Figure 7: Sankey diagram of efficiencies and inefficiencies for the 1024x12 data point in Figure 6.

i.e. the last data point in Figure 6. If we start from left and always follow the lower path, we can see that $TE * SerE * LB = PE$. Now, we start with the blue flow on the lower right. While this bar is labelled with PE, is also represents all compute time spent in application code (useful execution), i.e., before impacts from parallel execution. The first impact results from imperfect load balance, measured to be 88.2 %, so 11.8 % are put on top. The next impact comes from alternating dependencies in the parallel execution, i.e., serialization inefficiencies, measured to be 56.2 %, which come on top next. The last impact factor is transfer inefficiency, which is put on top with a factor of 58.8 %. The result is normalized to a total of 100 %. Now, if we go back to the parallel efficiency on the right, we can see a slightly different interpretation of the values: we first see a transfer efficiency of 41.2 %, next we see a communication efficiency of 18 % and finally the parallel efficiency of 16 %. These are the only absolute values that are displayed in Figure 6:

- PE: the area at the bottom, marked in blue (16 % for the last data point)
- CommE: the area below the light green (blue+red+orange, 18 % for the last data point)
- TE: the area below the light purple, i.e., everything that remains after removing OpenMP and MPI transfer inefficiency (41.2 % for the last data point)

All other values are only visible as a ratio of other data points in the diagram, underlining the multiplicative nature of the plot. Looking at the values from Figure 2, MPI Serialization Efficiency of 46.4 % is a significantly high number. Where can we find this value in Figure 6? The light green area represents the impact of MPI Serialization Inefficiency. The value of the lower end of the light green area divided by the value of the upper end of the light green area shows exactly the MPI Serialization Efficiency of 46.4 %. If we would normalize the graph to the upper end of the light green area, the graph would directly show the MPI Serialization Efficiency as an absolute value. Similarly, if we normalize the graph to the upper end of the dark green area, the graph would directly show the Serialization Efficiency as an absolute value, still at the lower end of the light green area.

2.1.4 Scaled inefficiency plot

A big disadvantage of Figure 6 is that the size of the area depends on the ordering of the stacking. Since we rescale to 1 after each stacking step, the area of serial inefficiency is much smaller than the area consumed by transfer inefficiency, although both inefficiencies account for about 40 %.

Figure 8 presents the inefficiencies scaled to their impact. The absolute height of the stack represents the parallel inefficiency. The relative height within the stack represents the weighted contribution of the individual inefficiencies to the overall situation. We can easily spot that in the last scale, MPI serial inefficiency and MPI transfer inefficiency have the highest and almost equal impact.

2.2 GPU-aware POP metrics

In multi-level parallel programming, we observe different kinds of parallelism at different levels. Concurrent execution units constitute the parallelism at the different levels. Many classical HPC applications use only MPI as the level of parallelism with a single thread per MPI process. In multi-threaded MPI applications, the threads spawned by each process constitute the next level of concurrency. In hybrid MPI+GPU applications, the highest level of concurrency still comes from MPI processes. The next level of concurrency depends on the concrete application.

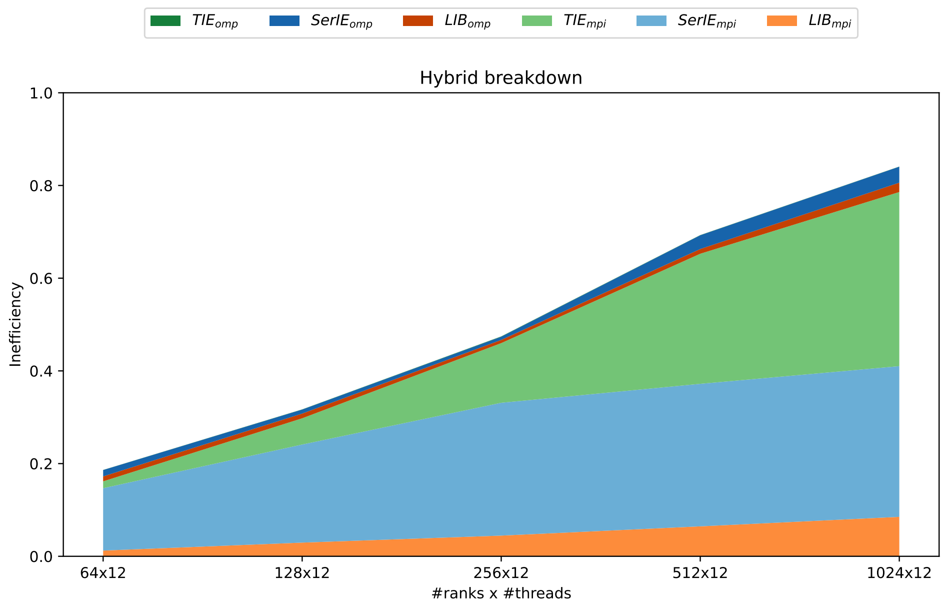


Figure 8: Parallel inefficiency with scaled contributions from the other inefficiencies.

Some applications might consider the CPU as a computing resource, while other applications only consider GPU as their computing resource.

2.2.1 Single GPU-Only

If the application considers that useful computation is completely performed by the GPU, the code executed on the host CPU can be considered to drive the parallel execution on the GPU. In such a case, the POP model will consider this driver code as an overhead of the GPU offloading approach. With a single GPU per process, all this overhead represents GPU transfer inefficiency since, with a single GPU, we have no concurrent execution at this abstraction level. Load imbalance or serialization between GPUs used in the execution reflects imbalances or serialization between MPI processes and will be reflected in the MPI metrics rather than the GPU metrics.

2.2.2 Mixed CPU-GPU execution

If we consider CPU and GPU as concurrent resources at the same level, we might think of an alternating use in terms of serialization or load imbalance. In such setup, we can consider the whole CPU and the whole GPU as a dedicated resource at the same level. The concurrency internal to the CPU (e.g., OpenMP multi-threading) and to the GPU (e.g., scheduling of kernels to device queues) constitutes an additional level.

Interleaving CPU-GPU Applications that are only partially prepared for GPU offloading, will alternatingly fail to utilize CPU or GPU resources. As a result, the combination of load balance and serialization will never exceed 50 %, highlighting that at least half of the resources are wasted all the time.

Concurrent CPU-GPU Overlapping GPU offloading with computation on the CPU improves the utilization of compute resources and will result in higher efficiency numbers.

2.2.3 Multi-GPU

With multiple GPUs connected to each MPI process, we observe concurrency at the GPU abstraction level. As a result, we will be able to observe GPU serialization and load imbalances in addition to transfer inefficiencies. Accessing multiple GPUs from the same process is sensitive to the data placement relative to the GPU. For execution on the CPU, accessing data through the interconnect will impact the IPC and, therefore, the computational scalability factor. For execution on GPU, streaming data to the GPU through the interconnect will result in increased data transfer times and, therefore, in reduced transfer efficiency.

2.2.4 Shared utilization of GPU

In a different usage scenario, multiple processes on the same compute node use the same GPU to offload certain well-suited tasks. In such a scenario, the GPU becomes an over-subscribed and shared resource.

2.2.5 Summary GPU-aware POP metrics

While we consider *single GPU-only* to be well-understood, further research and discussion are necessary to consider different aspects of GPU offloading effects for the POP methodology.

2.3 Euro HPC CoE flagship codes supported

Different versions of GPU-aware POP metrics were applied during various performance audits. `POP3_AR_001`, `POP3_AR_002`, `POP3_AR_004`, and `POP3_AR_008` analyze GPU codes with one GPU per MPI rank and no or poor CPU utilization. The code in `POP3_AR_005` can potentially use multiple GPUs from one process using asynchronous OpenMP target regions. Due to tool limitations, the analysis was limited to the CPU-only version of the code. The code analyzed in `POP3_AR_006` uses CUDA graphs and explicit stream synchronization. The analysis was affected by limited tool support for these programming approaches.

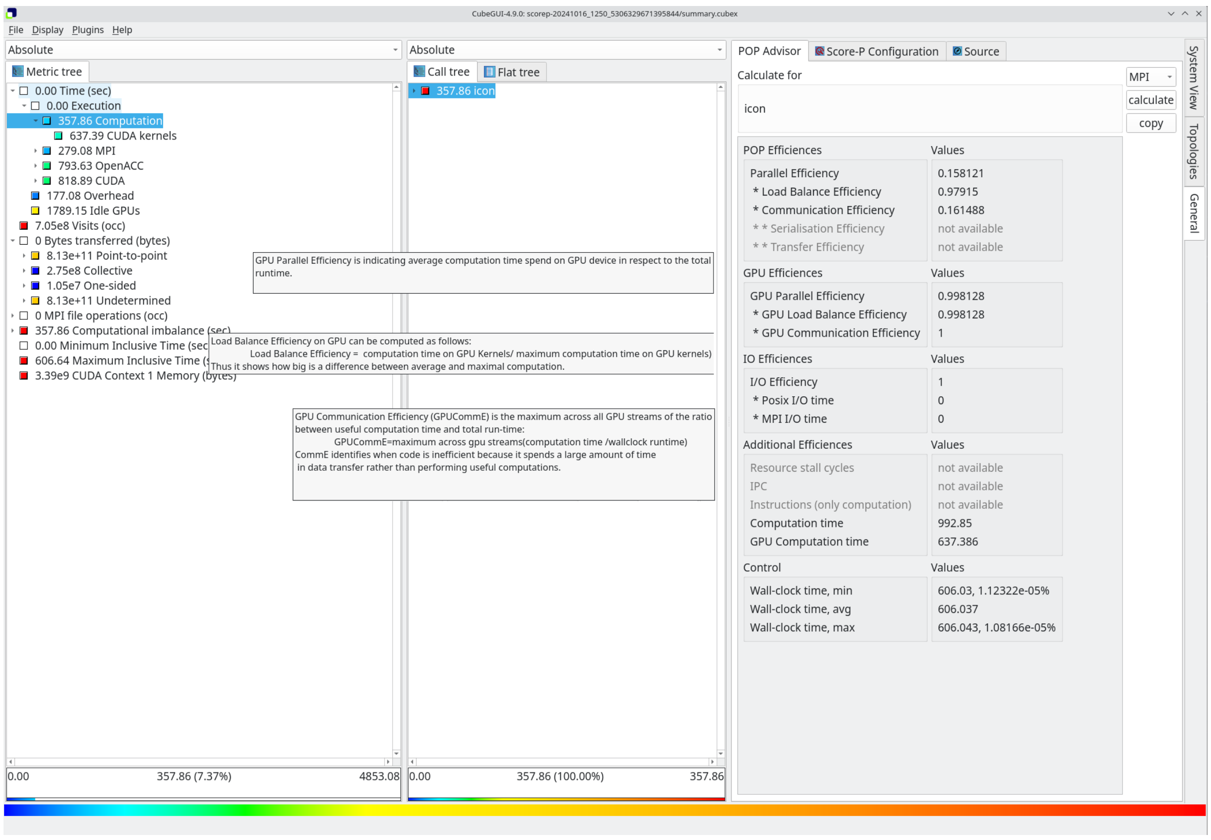
Overall, about 50 % of the Euro HPC CoE flagship codes that we have seen in performance audits use the CPU for their computation, but not the GPU. The POP methodology fully supports the analysis of these codes. Most ($> 50\,\%$) of the flagship codes that utilize the GPU for computation completely rely on the computation executed on the GPU and no or negligible code to be executed on the CPU. The POP methodology fully supports the analysis of these codes. Therefore, we support $> 75\,\%$ of the flagship codes with the POP methodology.

3 Improvements made to the tools

The POP3 consortium develops a wide range of tools for parallel application performance investigation and optimization. This list consists of Score-P, Scalasca, Cube (developed by JSC), Extrae, Dimemas, DLB, TALP, Paraver, Clustering, Folding (BSC), MAQAO, ONE View (UVSQ), MUST, Archer, OTF-CPT (RWTH), MERIC, RADAR visualizer (IT4I@VSB), and CARM (INESC-ID). These tools and some vendors' tools (such as NVIDIA Nsight Systems) are used in the POP3 assessments. However, the development of all of these tools is not funded by the project. As a EuroHPC Center of Excellence, we committed to deploying and validating POP tools to EuroHPC systems (KPI 4.3). This KPI is set for a subset of the tools, which we identify as POP3 flagship codes. The flagship codes are Extrae, TALP, Score-P, Scalasca, MAQAO, and MERIC.

3.1 Score-P-based tools (Score-P, Scalasca, Cube)

The Score-P based performance analysis workflow consists of three major tools: Score-P, Scalasca, and Cube. Score-P is a community-maintained instrumentation and measurement infrastructure to collect performance data from HPC applications. Score-P is easy to use, highly scalable, and able to generate both summarised call-path profiles and detailed event traces. Score-P's event traces can be manually examined using the Vampir trace visualizer or



```
0.00 Time (sec)
  0.00 Execution
    357.86 Computation
      637.39 CUDA kernels
    279.08 MPI
    793.63 OpenACC
    818.89 CUDA
  177.08 Overhead
  1789.15 Idle GPUs
7.05e8 Visits (occ)
0 Bytes transferred (bytes)
  8.13e+11 Point-to-point
  2.75e8 Collective
  1.05e7 One-sided
  8.13e+11 Undetermined
0 MPI file operations (occ)
357.86 Computational imbalance (sec)
0.00 Minimum Inclusive Time (sec
606.64 Maximum Inclusive Time (
3.39e9 CUDA Context 1 Memory (bytes)
```

0.00 357.86 (7.37%) 4853.08

357.86 icon

0.00 357.86 (100.00%) 357.86

GPU Parallel Efficiency is indicating average computation time spend on GPU device in respect to the total runtime.

Load Balance Efficiency on GPU can be computed as follows:
Load Balance Efficiency = computation time on GPU Kernels/ maximum computation time on GPU kernels)
Thus it shows how big is a difference between average and maximal computation.

GPU Communication Efficiency (GPUCommE) is the maximum across all GPU streams of the ratio between useful computation time and total run-time:
GPUCommE=maximum across gpu streams(computation time /wallclock runtime)
CommE identifies when code is inefficient because it spends a large amount of time
in data transfer rather than performing useful computations.

Calculate for

icon

POP Efficiences	Values
Parallel Efficiency	0.158121
* Load Balance Efficiency	0.97915
* Communication Efficiency	0.161488
* * Serialisation Efficiency	not available
* * Transfer Efficiency	not available
GPU Efficiences	**Values**
GPU Parallel Efficiency	0.998128
* GPU Load Balance Efficiency	0.998128
* GPU Communication Efficiency	1
IO Efficiences	**Values**
I/O Efficiency	1
* Posix I/O time	0
* MPI I/O time	0
Additional Efficiences	**Values**
Resource stall cycles	not available
IPC	not available
Instructions (only computation)	not available
Computation time	992.85
GPU Computation time	637.386
Control	**Values**
Wall-clock time, min	606.03, 1.12322e-05%
Wall-clock time, avg	606.037
Wall-clock time, max	606.043, 1.08166e-05%

Figure 9: CUBE GUI showing POP efficiency metrics for a Score-P profile of ICON on JEDI, the JUPITER test system, using MPI+OpenACC generating CUDA kernels. The Advisor plugin in the right column has a new layout showing the GPU-related metrics.

automatically analyzed using the Scalasca Trace Tools. Its distinctive feature is the scalable automatic trace-analysis component, which provides the ability to identify wait states that occur, for example, as a result of unevenly distributed workloads. Besides merely identifying and quantifying wait states in communication and synchronization operations, the trace analyzer is also able to pinpoint their root causes (i.e., excess computation or communication, called "delays") as well as their impact. Cube is the performance analysis report explorer that visualizes results from Score-P's profile measurements as well as the extended analysis reports of the Scalasca Trace Tools. Cube is a generic tool for displaying a multi-dimensional performance space consisting of the dimensions of performance metrics, call-paths, and system resources. Aside from an array of analysis plugins, the specific focus in the context of this project is its ability to calculate the POP performance efficiency metrics for a selected focus of analysis (i.e., a given set of call-paths).

While Score-P and Scalasca have added features from outside contributions that POP analysts can profit from during their work, the major objective for development for this project has been improving the workflow on the Cube side.

The CubeGUI Advisor plugin for POP metrics has been extended to correctly calculate the metric set for measurements that contain GPU computation (i.e., kernels). By adding new metrics for GPU Parallel Efficiency, Load Balance Efficiency and Communication Efficiency, analogous to the existing set, the Advisor can now calculate metrics for both host and device side, where kernel computation data is available. Also, the Advisor plugin has been visually updated, and new utility metrics separated GPU computation time and reference wallclock values have been added. By moving the POP metric calculation to the library backend, the same calculation can be used in the GUI and, by extension, the client-server approach, as well as the newly created command line tool to calculate the metrics. The new cube_pop_metrics tool produces the same metrics in textual form with the general layout of the Advisor plugin.

3.2 BSC Tools (Extrae, Paraver, Dimemas)

The BSC performance analysis ecosystem consists of three major tools: Extrae, Paraver, and Dimemas. Extrae is the instrumentation framework that generates traces for Paraver. It supports a wide range of HPC platforms, programming models, and languages. Extrae employs various interposition mechanisms to inject probes into the target application, collecting performance data regarding the application's activity. Most of these mechanisms operate directly on production binaries, eliminating the need for special compilation or linking.

Paraver is a data browser based on event traces, offering extensive flexibility to explore collected data. It provides detailed analyses of the variability and distribution of multiple metrics to help users understand the application's behavior. Paraver offers two main views: the timeline view, which displays application behavior over time, and the statistics view, which includes histograms and profiles to complement the analysis by providing quantitative measurements.

Dimemas is a simulator designed for message-passing programs. It reconstructs the temporal behavior of a parallel application using a recorded event trace. Dimemas enables users to simulate the application's parallel behavior on a different system and facilitates parametric studies with ease.

These three tools form an integrated ecosystem, allowing interaction between them. For instance, Paraver can invoke Dimemas to simulate a trace currently being analyzed, and Dimemas can generate a Paraver trace file, enabling the user to conveniently examine and compare the original and simulated runs for a deeper understanding of the application behavior. To facilitate extracting insight from detailed performance data, these core tools are complemented by additional modules designed for data analytics: Clustering, Tracking, and Folding allow the

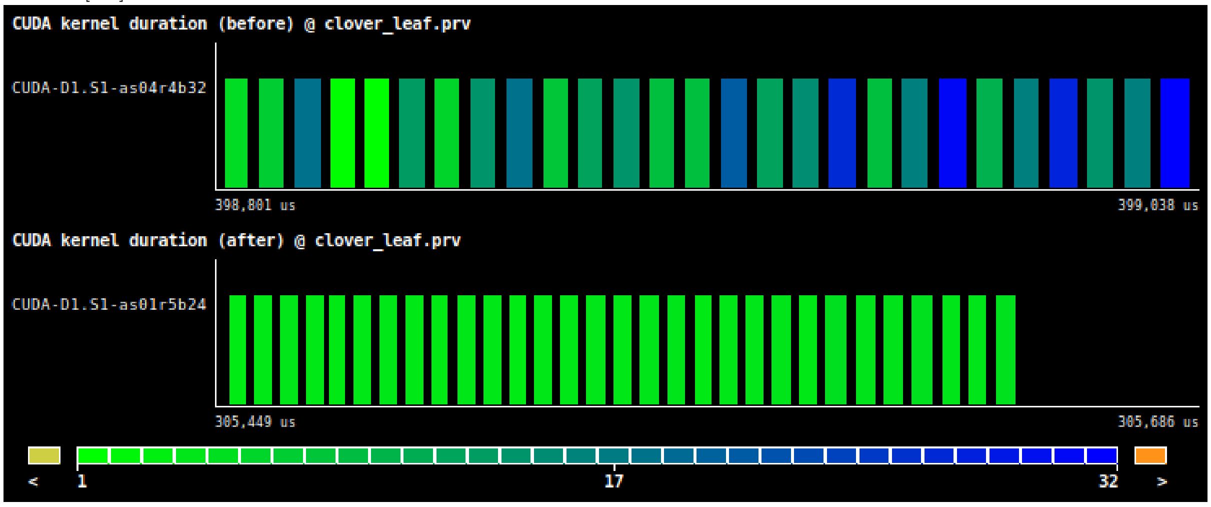


Figure 10: Comparison of measured CUDA kernel durations before (top) and after (bottom) computing latencies between closely occurring device events with `cudaEventElapsedTime`. The latter shows reduced 20 % timing overhead.

performance analyst to identify the program structure, study its evolution, and examine the internal structure of computation phases. Finally, BasicAnalysis computes the POP efficiency metrics for Paraver traces, providing plots, *CSV* files, and text output. It supports traces of applications that make use of a single parallel programming model, as well as hybrid MPI+X combinations, where X is a second parallel programming paradigm, such as OpenMP or CUDA.

The primary focus of developments in the scope of POP3 across all tools has been to extend their functionality for greater reach and to meet the requirements of target applications in POP studies, focusing on better coverage of hybrid codes, CUDA accelerators, and scalability and usability improvements.

3.2.1 Extrae

During the reporting period, Extrae development focused on four main areas, with the first two being already available and the latter two in earlier stages of development. First, enhancing instrumentation support for CUDA accelerators. Second, extending support for MPI Fortran 2008. Third, a prototype to enable tracing of applications using HIP/ROCm. Fourth, resuming work on a lightweight version of the Extrae tracing infrastructure.

Improved CUDA tracing support Regarding improvements made to the instrumentation support for CUDA, the primary achievement has been improving the accuracy of kernel duration measurements, reducing timing overhead by approximately 1.5 microseconds per measurement, as illustrated in Figure 10. This represents a significant improvement, achieving a threefold reduction relative to the system's 0.5-microsecond resolution margin, as stated in the CUDA specification manual. This was accomplished by utilizing `cudaEventElapsedTime` to measure latencies between closely occurring device events rather than relying on an initial reference event taken during initialization. This approach resolves, in turn, clock synchronization issues between streams, which previously caused inconsistencies in the sequence of captured stream activities.

Moreover, several changes have been made to improve the scalability and reduce the overhead of CUDA tracing. Previously, the memory buffer used to store GPU events had a fixed size and required recompilation to resize it. This has now been replaced with a dynamically allocated buffer that grows in chunks as needed, allowing for the instrumentation of much larger applications. Tracing overhead has also been reduced by optimizing the management of the tracing buffer. This includes minimizing the number of synchronizations needed to process collected GPU events and relocating the flushing of trace data to occur before explicit synchronization points. This approach leverages idle time to perform the flush operation more efficiently, overlapping the flush overhead with the wait time for active kernels to complete.

In terms of the information collected, the traces have been enriched by adding instrumentation for `cudaEventRecord` and `cudaEventSynchronize` calls. The mapping of information has also been modified by removing synchronization events from the device streams, and storing this data exclusively on the host process. Additionally, streams without activity have been excluded from the trace to enhance clarity. Finally, Extrae has been adapted to enable the host process to read PAPI counters from the CUDA component, capturing metrics from the devices. While accessing CUDA counters through PAPI is seamless for Extrae, changes were necessary because calls to CUDA during PAPI initialization were captured by Extrae, causing recursion issues. This has been resolved by improving controls in Extrae to prevent nested instrumented calls.

In relation to usability enhancements and bug fixes, several minor issues have been resolved, including improved handling of the default stream and refactoring code related to trace data flushing for better maintainability.

Current efforts are focused on evaluating the transition from the Callback API to the Activity API as the source of information. Preliminary results indicate a potential reduction in overhead and an improvement in the accuracy of kernel timing measurements. This improvement is primarily due to the elimination of calls to `cudaEventRecord`, `cudaElapsedTime`, and `cudaEventSynchronize`, which are no longer required to obtain timing measurements for the devices.

MPI Fortran 2008 support Although the number of users of MPI Fortran 2008 is relatively small, we have encountered use cases over the years that justify enhancing its support. To address this, we have added new wrappers to support *mpi_f08* bindings. These wrappers handle optional *ierror* arguments and *choice buffers*. To foster extensibility, we have included a wrapper generation script that automatically generates code to intercept the *mpi_f08* bindings, making it straightforward to extend support to additional MPI calls.

HIP instrumentation support Regarding instrumentation support for applications using accelerators on AMD-based platforms, a prototype has been developed to extend Extrae by intercepting calls to the HIP (Heterogeneous-Compute Interface for Portability) programming interface. This prototype has explored the use of the RocTracer Callback API and Activity API and has enabled the generation of traces on the CTE-AMD cluster at BSC using AMD Radeon Instinct MI50 GPUs. An example is shown in Figure 11, illustrating a run of the CloverLeaf benchmark with 2 MPI ranks using one device and stream per rank. However, integration into Extrae for a production release has not been started yet, as significantly more effort has been dedicated to CUDA improvements than originally anticipated. Nonetheless, many aspects of the HIP instrumentation are shared with CUDA instrumentation, which will allow it to benefit from the improvements already made to CUDA support.

Lightweight tracing framework We have resumed development on Extrae-lite, a lightweight version of Extrae, originally conceived in a previous project, to collect basic statistics with minimal overhead. Unlike Extrae, which generates traces, this version aggregates metrics such as the

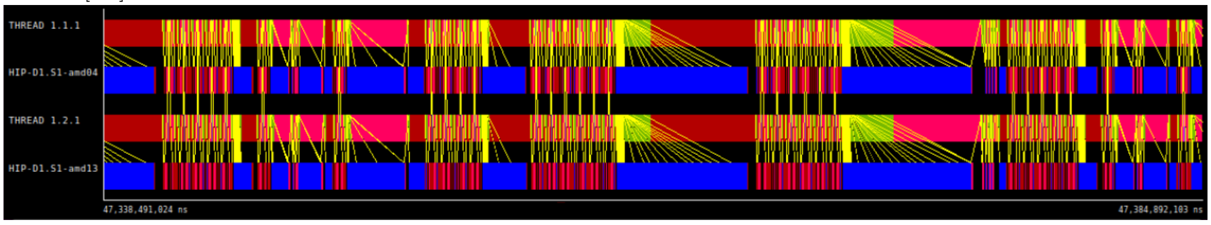


Figure 11: Application states defined by instrumented HIP calls for execution of CloverLeaf benchmark on CTE-AMD, with 2 MPI ranks and one device and stream per rank.

number of MPI calls, cumulative time spent in MPI, computation time, and hardware counters. These statistics, collected per MPI rank and saved as CSV files, enable the computation of some of the POP efficiency metrics for extremely long runs at large scales. This tool uses dynamic interposition via the `LD_PRELOAD` environment variable and allows default data collection for the entire execution or manual instrumentation to mark specific phases, with statistics reported per phase. It can also merge the statistics into a single trace for Paraver visualization. Future plans include extending support to additional runtimes and automating the collection of information.

Other developments In addition to the efforts above, two additional areas of development, carried out in the scope of the Barcelona Zettascale Lab project [1], are worth highlighting. Although the cost of this work is not charged to the POP3 project, its outcomes are highly relevant to POP studies and are closely related to improving OpenMP tracing support. These developments include extending the summarized tracing mode (burst mode) to support OpenMP and hybrid MPI+OpenMP applications, as well as updating OMPT support to comply with the latest specifications.

Work has been done to extend the burst mode tracing to support summarized traces for OpenMP and hybrid MPI + OpenMP applications. Burst mode significantly reduces the size of traces by recording only information related to long computation regions and collecting statistics for the omitted regions. This enhanced support for OpenMP has been included in Extrae version 4.2.0, which is already publicly available.

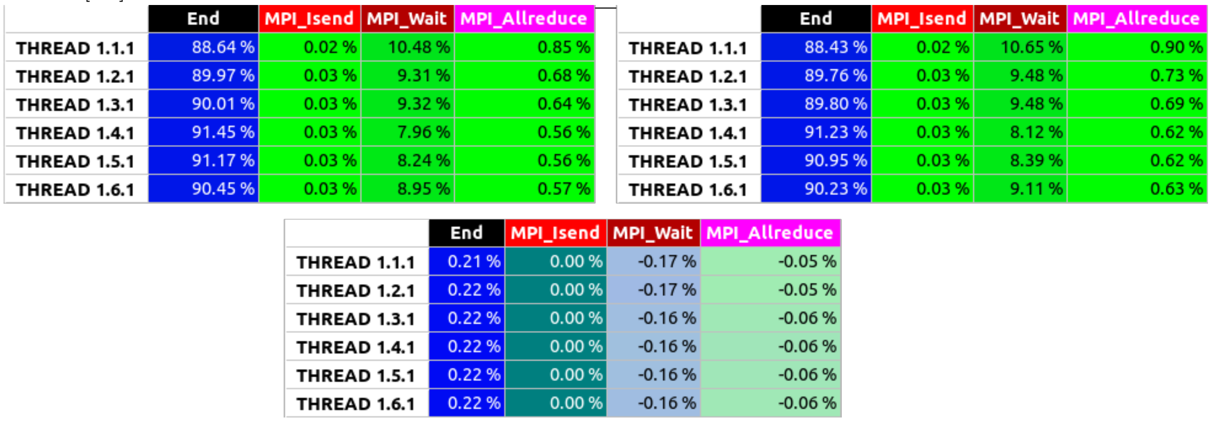
Following this, efforts have shifted toward standardizing OpenMP tracing to rely on OMPT rather than on vendor-specific implementations. This development has been integrated into Extrae as a module that generates a tracing library, activating OMPT instrumentation for the following OpenMP pragmas: *parallel*, *barrier*, *parallel for* with all scheduling variants, *parallel sections*, *task*, and *taskwait*. Work is ongoing to extend support to additional pragmas.

3.2.2 Paraver

During the reporting period, Paraver development primarily focused on adding support for deriving histograms, along with implementing various functionality enhancements and addressing some bug fixes.

Derived histograms This development extends to histograms the capability to perform operations between two histogram windows, as illustrated in Figure 12, similar to the existing functionality in Paraver that allows users to derive a third timeline window as the result of an operation between two others.

Users can perform this operation by dragging and dropping one histogram from the list of open windows onto another. For the resulting derived histogram, the operations currently supported include sum, subtract, multiply, divide, minimum, and maximum.



	End	MPI_Isend	MPI_Wait	MPI_Allreduce
THREAD 1.1.1	88.64 %	0.02 %	10.48 %	0.85 %
THREAD 1.2.1	89.97 %	0.03 %	9.31 %	0.68 %
THREAD 1.3.1	90.01 %	0.03 %	9.32 %	0.64 %
THREAD 1.4.1	91.45 %	0.03 %	7.96 %	0.56 %
THREAD 1.5.1	91.17 %	0.03 %	8.24 %	0.56 %
THREAD 1.6.1	90.45 %	0.03 %	8.95 %	0.57 %

	End	MPI_Isend	MPI_Wait	MPI_Allreduce
THREAD 1.1.1	88.43 %	0.02 %	10.65 %	0.90 %
THREAD 1.2.1	89.76 %	0.03 %	9.48 %	0.73 %
THREAD 1.3.1	89.80 %	0.03 %	9.48 %	0.69 %
THREAD 1.4.1	91.23 %	0.03 %	8.12 %	0.62 %
THREAD 1.5.1	90.95 %	0.03 %	8.39 %	0.62 %
THREAD 1.6.1	90.23 %	0.03 %	9.11 %	0.63 %

	End	MPI_Isend	MPI_Wait	MPI_Allreduce
THREAD 1.1.1	0.21 %	0.00 %	-0.17 %	-0.05 %
THREAD 1.2.1	0.22 %	0.00 %	-0.17 %	-0.05 %
THREAD 1.3.1	0.22 %	0.00 %	-0.16 %	-0.06 %
THREAD 1.4.1	0.22 %	0.00 %	-0.16 %	-0.06 %
THREAD 1.5.1	0.22 %	0.00 %	-0.16 %	-0.06 %
THREAD 1.6.1	0.22 %	0.00 %	-0.16 %	-0.06 %

Figure 12: Derived histogram (bottom) computing the subtraction of MPI time percentage between two runs. Negative values indicate an increase in the percentage of time in the second scenario.

In this initial implementation the two histograms must have the same dimensions, meaning the same number of columns and rows. All functionalities available for standard histograms, such as saving a configuration (CFG), exporting text data, or saving as an image, are also available for derived histograms.

This feature is expected to be available in the development branch of Paraver in early 2025. Ongoing work includes ensuring that parent histograms, which serve as operands for a derived histogram, cannot be deleted while the derived histogram depends on them.

Functionality enhancements Several functionalities have been developed to enhance histograms, semantic and filter modules, external utilities, and visualization; and are included in Paraver version 4.12.0, which is already publicly available.

- Redesigned software counters summarization tool: This tool enables the counting or accumulation of counter event values at sampling intervals. Previously, it could perform either counting or accumulation, but not both simultaneously. The utility's performance has been improved through the use of better structures, and it now supports both counting and accumulating events at the same time, enabling it to convert a detailed trace into a burst mode trace.

- Improved code color palette generation: The color palette is automatically expanded from the designated colors in the trace *PCF* file, making each one distinct and easy to identify when the mouse pointer is over it. However, the former algorithm did not take into account the timeline background color. Since users have the option to change the background color, this could result in a low contrast combination. In the following iteration, the algorithm was extended to assign colors based on the background. Nonetheless, for different users with different background settings, the same view was drawn with a different choice of palette colors. This represented a significant difficulty in comparing views and sharing analysis between users. Through the replacement of conflicting palette colors with ones that have a proper relative luminance to the current background color, the new palette generation creates more visually comparable colors across different backgrounds.

- New communication filter option: The filter module already included options to select communications based on their tag, size, or sender/receiver object. The newly introduced intra-node and inter-node options enable the selection of communications between objects either within the same node or across different nodes. This enhancement makes connectivity pattern views and the information generated by semantic functions for communications more flexible.

- New semantic functions: Semantic functions *Number of receives* and *Number of receive bytes* were already available. The addition of two new functions, *Number of sends* and *Number of send bytes*, which accumulate values over the lifetime of communication lines, completes the set of semantic functions for ongoing communications.

- Added new histogram column statistic: The histogram view presents summary statistics per column, including *Total*, *Average*, *Maximum*, and *Minimum*. In histograms with a high number of rows, it can be difficult to determine how many cells have values contributing to the summary statistics. To address this, a new statistic, *Num cells*, has been introduced to count the number of rows with valid values, providing analysts with this additional measurement.

- Dynamic context menu *Run*: The *Run* option in the context menu enables the execution of external applications to operate on data for the selected region of the current trace. Previously, the list of external applications was displayed unfiltered, regardless of whether a given one was applicable to the current trace. The recent improvement adapts the displayed list to match the external applications that are actually installed and relevant to the current trace information, avoiding suggestions for incompatible tools. The future integration with CARM will also benefit from this feature.

3.2.3 Dimemas

During the reporting period, Dimemas development focused on two main areas. The first was adding support for hybrid applications using OpenMP to guarantee that the simulated runs keep the threads alignment. The second area was improving CUDA accelerator support, including enabling the simulation of multiple streams and updating Dimemas to the recent changes in the CUDA events recorded by Extrae.

Support for hybrid MPI+OpenMP The goal of this Dimemas extension is to ensure that simulations of MPI+OpenMP traces maintain synchronization between threads by aligning them at fork-join and barrier primitives. This extension must not be considered a simulation of the OpenMP runtime but facilitates the analysis of Paraver traces generated by Dimemas for MPI+OpenMP applications.

To maintain thread alignment, the synchronization mechanism has been extended to operate with punctual events within the traces. Specifically, synchronization based on OpenMP events has been introduced, assuming that all threads in a task will emit events at the same synchronization point (as guaranteed by Extrae). This enhancement ensures proper synchronization for fork-joins and barriers, as illustrated in Figure 13.

Improved simulation for CUDA accelerators Former CUDA simulations were limited to a single stream per task. As the most frequent scenario is to use multiple streams, the first enhancement in the CUDA support upgrade has been adding the necessary data structures to support the use of multiple streams per task.

Nevertheless, most of the effort has been directed toward simulating synchronization points. Previously, the simulation of `cudaStreamSynchronize` and `cudaDeviceSynchronize` relied

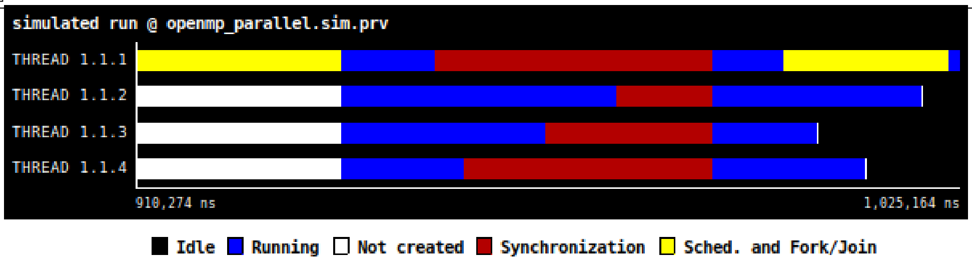


Figure 13: Simulated OpenMP parallel showing a perfectly synchronized start of the parallel region (blue) and end of the barrier (red).

on point-to-point communications between host and stream, limiting synchronization to one stream. This mechanism has been extended to enforce synchronization across streams by adapting the OpenMP event synchronization mechanism described in the previous section to also operate with CUDA events.

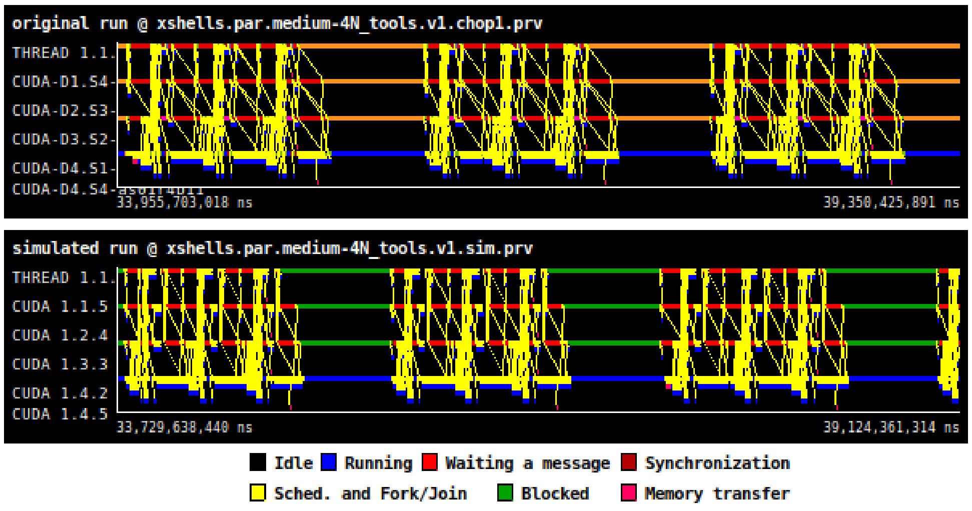


Figure 14: Comparison of real and simulated runs of a CUDA application with ideal settings (zero latency, bandwidth) employing multiple streams and maintaining synchronization coherence.

Additional adjustments have been made to ensure that recent changes in the traces generated by Extrae are correctly handled in Dimemas. The most relevant scenario corresponds to the synchronization between the host and the device. As noted in Section 3.2.1, the events previously included in the streams to mirror host activity, which enabled the simulation, have been removed. CUDA synchronization calls are now implemented by correlating `cudaLaunch` calls with their kernels execution to determine during the simulation the end of the synchronization call. Modeling with latency and bandwidth parameters is now limited to memory copy operations. An example of the current simulation support for CUDA is illustrated in Figure 14.

Moreover, a new configure option, `--disable-cuda`, has been introduced. This option allows Dimemas to ignore CUDA events during the simulation, forwarding those unmodified to the resulting simulated trace, speeding up the simulation process when the user does not need to generate CUDA metrics.

Finally, new CUDA calls are now recognized by the simulator, including `cudaDeviceReset`, `cudaFree`, `cudaMalloc`, and `cudaStreamDestroy`, maintaining all their information in the Paraver tracefile generated as output.

3.2.4 BasicAnalysis

During the reporting period, BasicAnalysis development has focused on maintaining the tool, with the most relevant efforts directed toward improving the integration with Dimemas, and increasing the tool's robustness.

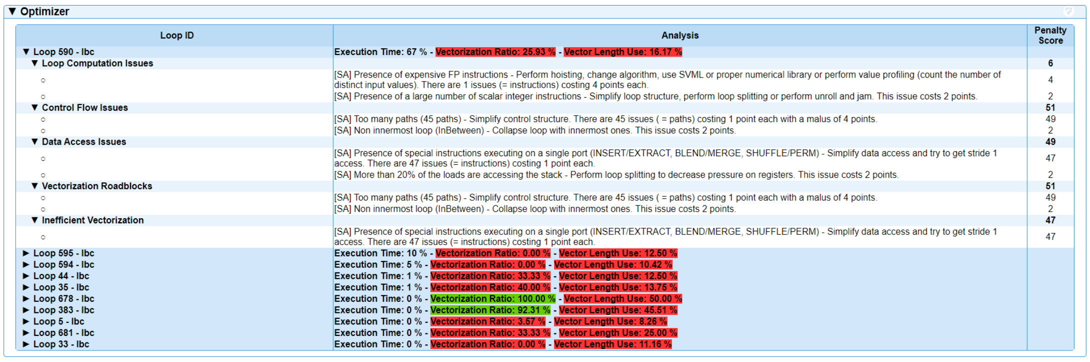
Regarding the integration with Dimemas, the configuration file for ideal simulations has been fine-tuned using the execution resources from the input trace file to adjust the definition of the architecture with respect to the number of nodes and number of CPUs and GPUs per node, minimizing the simulator memory footprint.

Additionally, the robustness in computing the efficiency metrics has been improved, covering corner cases that were not previously considered. For instance, the computation of *Frequency scalability* now exclusively considers computing phases that include hardware counter measurements, thereby avoiding distortion caused by phases lacking metric data. This is particularly beneficial for MPI+CUDA scenarios, where streams contain computations without associated counter metrics.

3.3 MAQAO

During the reporting period, the main development efforts in MAQAO have focused on improving the quality and ease of use of the information provided to the user. This was organized along two main axes:

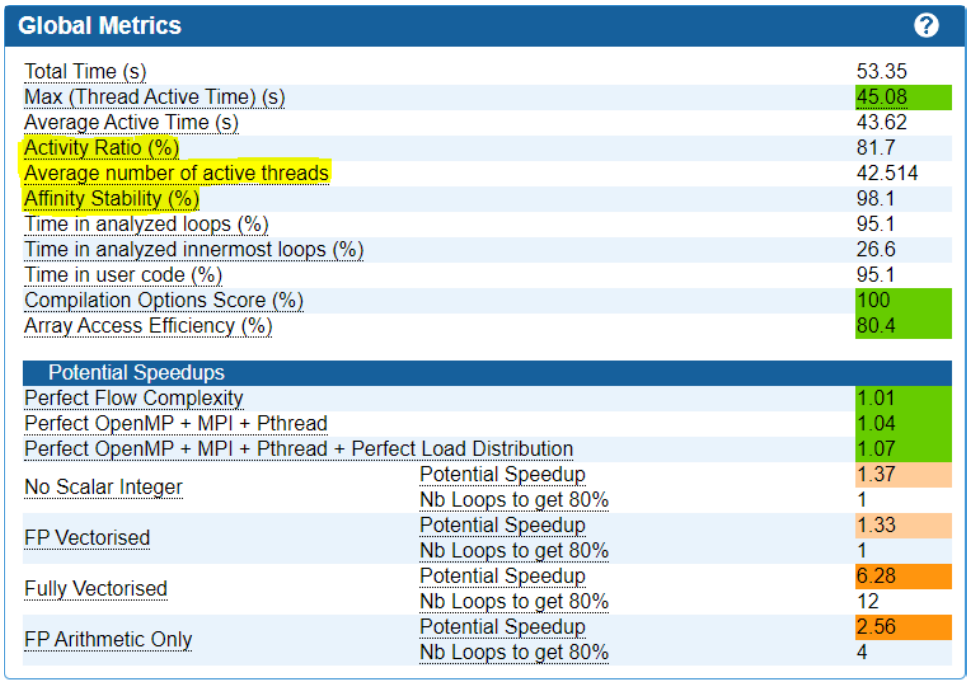
Enhanced detailed summary The detailed summary of MAQAO ONE View presents the main performance issues identified in the hottest loops, along with an estimation of the cost for their resolution. This summary was reorganized for better clarity, with a focus on the various categories of issues that could be encountered: vectorization, control flow, etc. A new aggregated mode has also been implemented, allowing a list of the main issues identified in an application with their number of occurrences. This mode can be especially useful in comparison reports, as it allows one to list the performance issues inherent to the application, regardless of compilers or architectures.


▼ Optimizer

Loop ID	Analysis	Penalty Score
▼ Loop 590 - lbc	Execution Time: 67 % - Vectorization Ratio: 25.93 % - Vector Length Use: 16.17 %	
▼ Loop Computation Issues		6
○	[SA] Presence of expensive FP instructions - Perform hoisting, change algorithm, use SVML or proper numerical library or perform value profiling (count the number of distinct input values). There are 1 issues (= instructions) costing 4 points each.	4
○	[SA] Presence of a large number of scalar integer instructions - Simplify loop structure, perform loop splitting or perform unroll and jam. This issue costs 2 points.	2
▼ Control Flow Issues		51
○	[SA] Too many paths (45 paths) - Simplify control structure. There are 45 issues (= paths) costing 1 point each with a malus of 4 points.	49
○	[SA] Non innermost loop (InBetween) - Collapse loop with innermost ones. This issue costs 2 points.	2
▼ Data Access Issues		49
○	[SA] Presence of special instructions executing on a single port (INSERT/EXTRACT, BLEND/MERGE, SHUFFLE/PERM) - Simplify data access and try to get stride 1 access. There are 47 issues (= instructions) costing 1 point each.	47
○	[SA] More than 20% of the loads are accessing the stack - Perform loop splitting to decrease pressure on registers. This issue costs 2 points.	2
▼ Vectorization Roadblocks		51
○	[SA] Too many paths (45 paths) - Simplify control structure. There are 45 issues (= paths) costing 1 point each with a malus of 4 points.	49
○	[SA] Non innermost loop (InBetween) - Collapse loop with innermost ones. This issue costs 2 points.	2
▼ Inefficient Vectorization		47
○	[SA] Presence of special instructions executing on a single port (INSERT/EXTRACT, BLEND/MERGE, SHUFFLE/PERM) - Simplify data access and try to get stride 1 access. There are 47 issues (= instructions) costing 1 point each.	47
▶ Loop 595 - lbc	Execution Time: 10 % - Vectorization Ratio: 0.00 % - Vector Length Use: 12.50 %	
▶ Loop 594 - lbc	Execution Time: 5 % - Vectorization Ratio: 0.00 % - Vector Length Use: 10.42 %	
▶ Loop 44 - lbc	Execution Time: 1 % - Vectorization Ratio: 33.33 % - Vector Length Use: 12.50 %	
▶ Loop 35 - lbc	Execution Time: 1 % - Vectorization Ratio: 40.00 % - Vector Length Use: 13.75 %	
▶ Loop 678 - lbc	Execution Time: 0 % - Vectorization Ratio: 100.00 % - Vector Length Use: 50.00 %	
▶ Loop 383 - lbc	Execution Time: 0 % - Vectorization Ratio: 92.31 % - Vector Length Use: 45.51 %	
▶ Loop 5 - lbc	Execution Time: 0 % - Vectorization Ratio: 3.57 % - Vector Length Use: 8.26 %	
▶ Loop 681 - lbc	Execution Time: 0 % - Vectorization Ratio: 33.33 % - Vector Length Use: 25.00 %	
▶ Loop 33 - lbc	Execution Time: 0 % - Vectorization Ratio: 0.00 % - Vector Length Use: 11.16 %	

Figure 15: Optimizer section of the summary report (LBC kernel) showing various issues on the main loop.

CPU activity metrics A new set of metrics has been collected by MAQAO and displayed in the ONE View reports. These metrics focus on quantifying the percentage of time spent by the processor actually working for the application. In particular, they should allow to detect the time spent in I/O interruptions, context switching, etc., and to present an estimation of the number of threads actually active simultaneously at a given time.



Global Metrics

Total Time (s)		53.35
Max (Thread Active Time) (s)		45.08
Average Active Time (s)		43.62
Activity Ratio (%)		81.7
Average number of active threads		42.514
Affinity Stability (%)		98.1
Time in analyzed loops (%)		95.1
Time in analyzed innermost loops (%)		26.6
Time in user code (%)		95.1
Compilation Options Score (%)		100
Array Access Efficiency (%)		80.4
Potential Speedups		
Perfect Flow Complexity		1.01
Perfect OpenMP + MPI + Pthread		1.04
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.07
No Scalar Integer	Potential Speedup	1.37
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.33
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	6.28
	Nb Loops to get 80%	12
FP Arithmetic Only	Potential Speedup	2.56
	Nb Loops to get 80%	4

Figure 16: Global metrics report with highlighted new metrics presenting the percentage of time threads were computing, the average number of active threads, and the average percentage of time threads were computing without performing context switches.

3.4 MERIC-based tools (MERIC, RADAR visualizer)

In the context of the POP3 project, the *MERIC runtime system* is used as a user-level tool for energy-efficiency analysis as described in the deliverable on second-level services D3.3. And *RADAR visualizer* graphically interprets data measured by the MERIC runtime system.

3.4.1 MERIC runtime system

MERIC is currently distributed in a recent release version 3.0.2, which contains the development done under the POP3 project during the first year, during which we focused on extending support for hardware platforms used in the EuroHPC systems (AMD MI250X GPU, Nvidia Grace CPU), providing new functionalities for analysis and optimization of GPU-accelerated codes, and creating new continuous integration (CI) pipelines.

Extending MERIC support about a new architecture means three steps – (1) energy consumption measurement, (2) hardware-specific power-management knobs tuning, and (3) energy-efficiency-related performance counters reading. At this moment, the list of monitored metrics across supported hardware platforms is not unified. We plan to focus on this issue during the following two years.

We implemented support for AMD GPUs, for which MERIC now provides the same functionality as for Nvidia GPUs. The capability for the GPUs is also being extended to provide more information about the executed workload. At this moment, this functionality is being implemented as a stand-alone code, which will be merged into the MERIC main development branch when reaching the necessary maturity.

To perform dynamic tuning of GPU streaming multiprocessor frequency, we should understand how often it is possible to change the frequency configuration. For that purpose, we defined a methodology to measure latency related to GPU frequency change and implemented a tool called LATEST, which allows the measurement of latencies of Nvidia GPUs [1].

In the context of CPU support, we implement the capability to scale the core frequency of a CPU, which is using `intel_cpufreq` scaling driver (one of the possible scaling drivers used for Intel CPUs). Traditionally, `userspace` scaling governor should reflect the user's requests to set frequency level specified in `scaling_setspeed`[2]. This is not the case with the `intel_cpufreq` driver. Thus, we implemented an alternative solution, which controls the CPU core frequency by changing `scaling_max_freq` when the `intel_cpufreq` driver is used.

Before the project, MERIC CI tests were simple tests checking measurement outputs to be in the correct form. MERIC relies on many hardware-specific interfaces that provide access to energy measurement or hardware power management knobs, and these cannot be tested without having a proper test bed. After activation of GitLab runner in the IT4Innovations' Complementary systems (more details in section 4), we focused on extending CI pipelines to execute the tests directly in a variety of available hardware platforms (CI pipelines in preparation for Intel Xeon Sapphire Rapids, AMD EPYC Zen2, IBM Power10, Fujitsu A64FX, Nvidia Grace [3], Nvidia Ampere, AMD MI100), a variety of software dependencies (msr-safe, AMD E-SMI, OCC, sysfs, etc.), and compilers. Compilation with specific dependencies and execution of a basic test in each of the supported hardware platforms has been established. A complex set of test cases to check all MERIC features is in its early stages.

In the second year of the project, we plan to implement or start work at:

- Intel/AMD RAPL energy measurement using sysfs interface, which will allow us to measure CPU energy consumption of the majority of EuroHPC systems,
- energy consumption measurement of Nvidia Grace CPU,
- complex set of CI tests for IT4Innovations' Complementary systems platforms,
- GPU dynamic tuning integration to the main development branch,
- unification of measured metrics across supported hardware platforms.

3.4.2 RADAR visualizer

For the development of the RADAR visualizer, we identified it is necessary to write a developer guide that describes the MERIC output data format and how to interpret the data and visualize it in a variety of graphical representations. This guide was recently released in pre-final version v0.12 (55 pages). This document should simplify communication between MERIC and RADAR visualizer development teams. According to the developer guide, several inconsistencies in data loading were identified and fixed.

RADAR visualizer is implemented in Python3 using PyQt5 and over twenty additional packages. When deploying as a software module in an HPC system it requires each package to have its own module. To simplify the tool distribution, we decided to implement compilation to a binary form, which includes all the dependencies.

[1]Journal paper on the topic of GPU frequency scaling latency in preparation.
[2]Sysfs interface located in `/sys/devices/system/cpu/cpu<threadID>/cpufreq/`.
[3]MERIC support for the Nvidia Grace CPU is currently very limited.

The source code was developed in an IT4Innovations' GitLab repository from 2018. At the beginning of the development, the developers added a large test dataset, which increased the size of the repository to 1.1 GB. For this reason we decided to create a new clean repository, where the development now continues. In the new repository, we started with a new stable release, which has been cleaned from the unused code.

In 2025, we plan to work at

- developer guide v1.0,
- GPU kernel visualisation support,
- switch from Qt5, which reaches the end of support in May 2025, to the recent Qt6.

3.5 Correctness Tools

3.5.1 Archer

The Archer tool is used to detect data races in OpenMP applications. It builds on Thread-Sanitizer in LLVM as the analysis back-end and provides OpenMP-specific synchronization information. This results in improved analysis results, as most false alerts can be avoided.

Archer has been successfully up-streamed into the LLVM project starting with LLVM 10. Continuous updates of Archer in LLVM ensure support for recently added OpenMP features like `all_memory` dependencies or `taskwait nowait` with dependencies. The broad support of Archer by most vendor compilers (e.g., AMD, HPE/Cray, Intel) demonstrates the broad acceptance of the tool and its impact on the HPC code development tool landscape.

3.5.2 MUST

The MPI runtime correctness tool MUST is used to detect errors in the use of the MPI interface. The tool observes the MPI function calls and provides analysis based on the provided function arguments, but also the sequence of MPI function calls. The performed analyses range from interval checks of provided integer arguments to deadlock detection in the MPI communication pattern.

The 1.10 release contains improved integration of MUST and Archer analysis to allow the detection of conflicts between memory accesses in non-blocking MPI communication concurrent with local memory accesses.

Integration of MUST in CI workflows was improved:

- The severity level of reported issues is represented by different exit codes, which allows CI frameworks to flag the results of individual tests.
- Additionally, specific messages can be actively suppressed, if they should not impact the CI pipeline result.
- Alternative JSON output allows integration of MUST reports into CI views.

Several tool-internal race conditions caused by multithreaded execution were fixed.

3.6 CARM

The development of assembly-level microbenchmarks that are automatically generated to fully utilize hardware capabilities in x86-64, AARCH64, and RISCV64 CPUs that support all major vector extensions on all vendors (SSE, AVX2, AVX-512, Neon, RVV0.7, RVV1.0, and SVE partially) integrated into the developed "CARM Tool". These microbenchmarks allow for the measurement of the peak bandwidth of the different memory levels and the peak floating-point performance of the underlying system. Furthermore, the automation of CPU feature detection such as underlying platform, available ISA extensions, and cache sizes to facilitate the usage of the tool, alongside the development of a graphical user interface (GUI) to further ease the process of running the necessary benchmarks for Cache-Aware Roofline Model (CARM) generation and the visualization of results. The development of CARM-based application profiling using performance counters via PAPI and dynamic binary instrumentation using DynamoRIO and Intel SDE. Inclusion of the "region of interest" profiling of applications in the scope of the CARM, in order to allow developers to obtain the performance optimization hints that the CARM model can provide for their applications, all encapsulated and integrated into the "CARM Tool" and its GUI. Initial integration with the Paraver and Extrae tools was conducted to provide CARM-based analysis of application traces produced by Extrae and visualized in Paraver. Benchmarking all available supercomputers using the CARM Tool under a variety of settings to later facilitate application profiling on these systems, by making benchmark results needed to generate the CARM available for all the available supercomputers.

3.7 OTF-CPT

During POP2, we developed OTF-CPT (the on-the-fly critical path tool) to easily collect POP metrics during the execution of an application and to prototype the calculation of more fine-grained hybrid performance metrics. Due to the easiness of use, the possibility to automate the complete workflow from execution of scalability experiments to generating diagrams like Figures 2 and 6 OTF-CPT became a valuable tool for our performance audits.

In the context of POP3, we fixed bugs (wild-card receives) and improved coverage of MPI features (communication involving MPI messages) whenever we ran into limitations during a performance audit. The lightweight nature of the tool allows the implementation of such extensions within a few days so that the few remaining bugs or missing features do not block the assessment for a long time.

4 CASTIEL CI/CD support

The POP-3 project actively supports CASTIEL-2 activities in Continuous Integration and Continuous Deployment (CI/CD).

1. POP helped to activate the automatic deployment process of public software modules in the Karolina system. Currently, developers of CoE codes may request access to `https://code.it4i.cz/kru0052/easyconfigs-coe` repository of EasyBuild recipes. These recipes describe where to get source files, how to compile the code, and dependencies to other system software. Also, it is possible to specify that the compilation will be performed in the compute node. In case a developer has already compiled the code, the compilation step is skipped, and the archive of the code installation is downloaded from the internet or from a dedicated system directory. If these compilation (download) steps succeed, a public system software module will be automatically created.

   This approach was presented in a CASTIEL CI/CD meeting in November and in an EPICURE [2] meeting in December.

2. POP cooperated with SPACE CoE in activating and testing continuous integration processes in the Karolina system. IT4Innovations deployed the Jacamar-CI [3] software tool that allows the sharing of Gitlab runners for all users of the system. These runners are available in the IT4Innovations' GitLab `https://code.it4i.cz/`. Codes developed or mirrored in the IT4Innovations' GitLab may execute CI pipelines in production systems Karolina, Barbora, and Complementary systems, which provide a range of non-traditional hardware platforms with respective software toolchains.

3. Extrae, TALP (DLB), and Score-P are distributed in the European Environment for Scientific Software Installations (EESSI) [4], which is currently available in Vega and Karolina non-accelerated partitions and Deucalion ARM partition.

4. We mirrored POP3 flagship codes to HLRS CASTIEL repository of CoE codes `https://codehub.hlrs.de/coes/pop`.

5 EuroHPC systems deployment status

POP3, as one of the Centers of Excellence projects, has a goal of deploying POP flagship codes to the EuroHPC systems. The project has specified KPI in deploying and validating POP tools to these systems.

To support POP tools development, integration tests, deployment, co-design activities, and application assessments (if having enough resources), we have submitted two projects providing project members access to EuroHPC systems.

1. IT4Innovations regular access call project OPEN-30-41 providing resource from 29.1.2024 till 28.1.2027 in:
   - EuroHPC LUMI
   - EuroHPC Karolina
   - IT4Innovations Barbora
   - IT4Innovations Complementary systems
2. EuroHPC development call project EHPC-DEV-2024D09-054 providing resources from 16.9.2024 for the following 12 months in:
   - EuroHPC Leonardo
   - EuroHPC MareNostrum5 (MN5)
   - EuroHPC Vega
   - EuroHPC MeluXina
   - EuroHPC Discoverer
   - EuroHPC Deucalion

Moreover, under the EHPC-DEV-2024D09-054 project, we asked the EPICURE Application Support Team (AST) for help with deployment to the EuroHPC systems – mostly for identification of the primary software toolchains and guidance in the automatization of the deployment process.

	Extrae	TALP	Score-P	MAQAO	MERIC
AMD Zen 2/3	Yes	Yes	Yes	Yes	Yes
Intel ICX/SPR	Yes	Yes	Yes	Yes	Yes
Nvidia Grace	No	Yes	Yes	Yes	No
Fujitsu A64FX	Yes	Yes	Yes	Yes	Yes
Nvidia Ampere/Hopper	Yes	Yes	Yes	No	Yes
AMD MI250X	No	Yes	Yes	No	Yes

Table 1: POP3 flagship codes support for CPUs and GPUs used in EuroHPC systems. No – no or partial support, Yes – full support, green identifies support added in 2024.

The current status of the POP3 flagship tools support of the computing hardware used in the EuroHPC systems is presented in Table 1. The Scalasca is not listed because it is platform-independent. As presented in the section 3, since the beginning of the project, the support has been extended in Score-P about Nvidia Grace CPU and AMD GPUs, in MAQAO about Nvidia Grace, and in MERIC about AMD GPUs.

		Extrae	TALP	Score-P	Scalasca	MAQAO	MERIC	MUST
LUMI	CPU			7.1				
	GPU							
LEONARDO	CPU	4.0.6		8.4	2.6.1			
	GPU							
MN5	CPU	4.2.3	3.4.1	8.1	2.6.1			
	GPU			8.4				
Vega	CPU	4.2.0	3.4	8.3				
	GPU							
Meluxina	CPU	4.0.6		8.1	2.6.1			
	GPU							
Karolina	CPU	4.2.3	3.5.0	8.4	2.6.1	2.20.1	3.0.3	1.10.0
	GPU							
Discoverer	CPU							
Deucalion	x86						3.0.2	
	GPU							
	ARM	4.2.0	3.4	8.4				

Table 2: Versions of the POP tools public software modules available in EuroHPC system. The red color identifies additional SW modules available as part of the EESSI distribution.

Table 2 presents which versions of the POP3 flagship codes are available in the public software module in which EuroHPC system. Tools developers are in contact with system administrators to deploy to other systems based on supported hardware with a focus on pre-exascale systems.

So far, MERIC has been deployed to Karolina and Deucalion only, which are systems that allow users to control hardware power management knobs to improve energy efficiency of their jobs. CINECA provides this functionality to its users in Galileo and Marconi systems, while deployment of the necessary Slurm plugins to Leonardo is pending due to technical issues. We expect to deploy MERIC also to remaining systems, where the MERIC will provide energy efficiency metrics measurement only. This requires modifications in the MERIC source code to support additional, slower, but more open APIs that expose relevant counters. Such a modification is a work in progress.

6 Conclusions

In this document, we presented work performed in the POP Methodology and Tools tasks in the context of WP4. We introduced various graphical representations for the metrics calculated with the POP methodology. For the analysis of hybrid MPI + GPU applications, we identified different approaches to calculate metrics depending on how the application uses the GPU. Based on the current status, we have already covered most of the MPI+GPU application use cases using the POP methodology.

Further research and testing are necessary to find concrete formulations for the remaining application use cases. The critical-path tool will allow us to quickly prototype performance factor models for the different GPU usage scenarios and confirm the results meet the expectations of the experts.

The effort spent on these two tasks is crucial in order to enable the work done by the POP services in WP3. The POP methodology and the analysis tools need to be kept up to date for all the different challenges faced by the POP services. Our proposed extensions to the POP methodology have already partially been implemented in our tools. At the same time programming models are also continuously evolving. For example, with the OpenMP 6.0 release this year, several tasking features were added to the standard, which has an immediate impact on the POP metrics. We expect application developers to include these new features in their codes in the near future. Thus, we plan to update our tools to enable performance analysis of such application codes accordingly.

List of abreviations

- API: Application Programming Interface
- BSC: Barcelona Supercomputing Center
- CI/CD: Continuous Integration, Continuous Deployment
- FZJ: Forschungszentrum Jülich GmbH
- D: deliverable
- HLRS: High Performance Computing Centre (University of Stuttgart)
- HPC: High Performance Computing
- INESC-ID: Instituto de Ennenharia de Sistemas e Computadores, Investigacao e Desenvolvimento em Lisboa
- IT4I: IT4Innovations, Technical University of Ostrava
- KPI: Key Performance Indicator
- POP: Performance Optimization and Productivity
- RWTH Aachen: Rheinisch-Westfaelische Technische Hochschule Aachen
- TERATEC: TERATEC
- USTUTT (HLRS): University of Stuttgart
- UVSQ: Universite de Versailles Saint-Quentin-en-Yvelines
- WP: Work Package

List of Figures

List of Tables

References

[1] BZL, Barcelona Zettascale Lab, project with reference REGAGE22e00058408992, https://bzl.es/en, "[Online; accessed 2024-12-27]".

[2] EPICURE, EuroHPC Application Support Project, https://epicure-hpc.eu/, "[Online; accessed 2024-12-27]".

[3] Exascale Compute Project, Jacamar CI, https://ecp-ci.gitlab.io/docs/admin.html#jacamar-ci, "[Online; accessed 2024-12-27]".

[4] EESSI, European Environment for Scientific Software Installations, https://www.eessi.io/, "[Online; accessed 2024-12-27]".