D8.2– Final report on methodology development and tool improvement
Version [1.0]

Document Information

Contract Number	824080
Project Website	www.pop-coe.eu
Contractual Deadline	M36, extended to M42
Dissemination Level	PU - Public
Nature	R
Authors	Joachim Protze (RWTH), Jonathan Boyle (NAG), Brian Wylie (JSC), Judit Gimanez (BSC), Andres S. Charif-Rubial (UVSQ), Phil Tooley (NAG), Radita Liem (RWTH), Fabian Orland (RWTH)
Contributors	William Jalby (UVSQ) and others of the WP8 group
Reviewers	Bernd Mohr (JSC)
Keywords	OpenMP, MPI, performance, POP methodology

Acknowledgements: *The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement n°824080.*

Change Log

Version	Author	Description of Change
v0.1	Joachim Protze	Initial version of the deliverable. Document structure definition.
v0.2	all	Providing content for sections
v0.3	Fabian Orland	Implementing suggestions from WP8-internal reviewing
v1.0	Fabian Orland	Final version for submission

Contents

Executive Summary

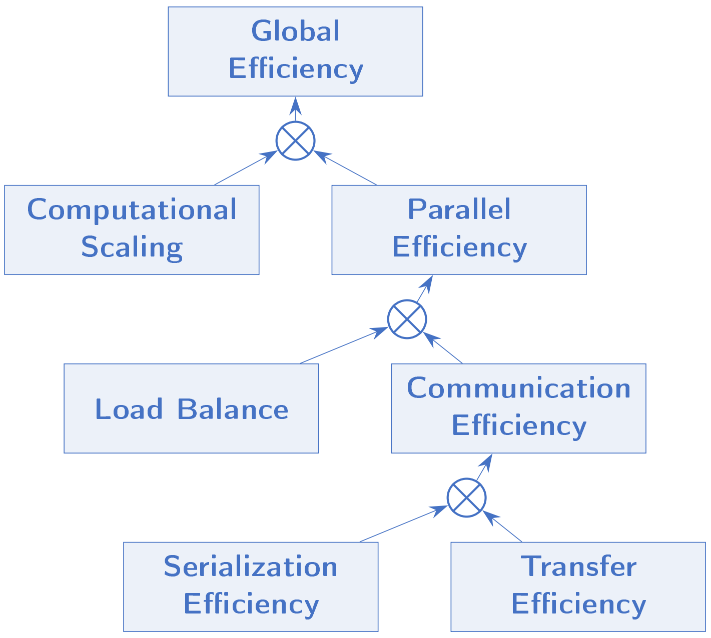
This deliverable reports about the overall work done in Work Package 8 *Tools and Methodology* in the 36+6 months of the POP-2 *Center of Excellence*. It supersedes the previous D8.1 document that reported on the progress after the first 18 months of the project. As suggested in the project proposal, the document first summarizes the efforts that we made towards extending the POP methodology to cover additional areas of interest for performance analysis. The extended POP methodology now incorporates hybrid inter- and intra-node parallelism and vectorization, but also the cost of I/O. We developed three concrete proposals to integrate hybrid use cases into the POP methodology. We also propose additional metrics for vectorization. Moreover, we make proposals for I/O metrics as complementary metrics to our existing hierarchy of metrics. We applied the hybrid metrics to POP assessments to derive a better understanding about the expressiveness and the advantages of the three different proposals. However, for the proposed I/O metrics we did not find enough use cases in the performance assessment to conduct a thorough investigation on the impact of I/O metric to improve performance. We discuss our experiences in this report.

Finally, we report about the tool development performed in the context of this work package. The development effort includes support for POP specific use cases, the release of new features in the analysis tools, as well as improvements towards better usability of the tools. In addition to the existing tools, two new tools have been developed. The first one called PyPOP based on Jupiter Notebook has the potential to greatly improve the sustainability of POP. PyPOP supports the POP analysis and semi-automatically prepares a report. The second one is a prototype tool that implements a lightweight calculation of hybrid POP metrics, according to our third proposal for a hybrid efficiency model, in an online fashion during runtime of a parallel program.

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:

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

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

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

we break down into the factors

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

and

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

In this work, we first describe our extensions made to these metrics. These extensions include three proposals of hierarchical metrics that support hybrid use-cases such as MPI+X codes. Moreover, we propose new metrics, which fit into the *Computation Efficiency* subtree of our existing hierarchy, to assess the vectorization of a code. We also propose metrics that cover the influence of file I/O. Our extensions have been used in performance assessments by workpackage 5. To conclude the first part of this document we demonstrate the practical usefulness of our extensions by highlighting some interesting assessments from workpackage 5 and discuss the advantages and disadvantages of the three hybrid metrics proposals.

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 using them.

2 Extensions made to the methodology

2.1 Metric definition for hybrid MPI + OpenMP

To exploit the full potential of today's HPC machines, many applications adopt a hybrid distributed and shared memory programming model. The most commonly used approach is to use MPI for distributed memory and OpenMP for shared memory parallelization. Since the two paradigms are conceptually orthogonal, we also propose to reflect them orthogonally in the hybrid metric model. Extending the original POP-1 metrics to include hybrid parallelism proved to be more complicated than expected before the start of the project. The combination of two or even more different programming paradigms results in a lot of corner cases that an extended model needs to cover. On the one hand the model should be general enough to cover different programming paradigms. On the other hand the model should also provide meaningful insights into a specific programming paradigm. A key reason POP methodology allows a clear understanding is that individual inefficiency contributions are attributed to a single low-level metric, allowing unambiguous interpretation. Thus, we propose three sets of hybrid metrics to assess these kinds of applications. All of them also take full advantage of this hierarchical methodology. One of the proposed methodologies deviates from the current POP MPI metrics, where the hybrid metrics are additive instead of multiplicative. Our second and third proposal maintain the approach of original POP MPI metrics, where the parent metric is the product of its child metrics. The additive methodology adopted by the hybrid scheme is described first and has two advantages. Firstly, each hybrid efficiency metric measures the total cost of the issue(s) under consideration, i.e. relative to the runtime. And secondly, this additive scheme gives more freedom when defining child metrics. This hybrid hierarchy can be used with pure OpenMP or pure MPI codes. For the latter case, it gives an additive version of the POP MPI hierarchy, i.e., load balance efficiency and serialization efficiency are redefined while all other metrics have the same definition for multiplicative and additive schemes. The multiplicative methodology adopted by the hybrid scheme is described afterward and follows the existing POP MPI metrics more closely. The third proposal was developed towards the end of the project after experiences with the previous two models were already made. It follows the existing POP MPI metrics very similarly but also allows to breakdown all of the lowest level child metrics of parallel efficiency (i.e. *load balance*, *serialization* and *transfer efficiency*) for the MPI and OpenMP programming models separately. Both the additive as well as the multiplicative metrics are designed so that they can be calculated using Extrae as well as ScoreP trace data. The third proposal relies on critical path analysis that has been implemented in a prototype tool but should be transferable to our existing tools with manageable effort. In the current state, the metrics discussed in this section ignore the influence of I/O, respectively, taking the influence of I/O as given. Therefore, I/O is interpreted as useful computation or MPI time. To understand the expressiveness of each of the multiplicative and additive sets of hybrid metrics, we apply both for POP assessments of hybrid applications. Based on the results of such assessments, we discuss the expressiveness of all three hybrid metrics proposals. Afterwards, we give recommendations which of our models is best used for which specific use-case.

2.1.1 Glossary of metric terms

Notion of execution units

$$
\begin{aligned}
P &= \text{number of processes} \\
p &\in \{1 \dots P\} \\
t_p &= \text{number of threads of process p} \\
T &= \sum_p (t_p) = \text{total number of threads} \\
t &\in \{1 \dots t_p\} \\
\square_{p,t} &= \text{property of thread t on process p}
\end{aligned}
$$

Notion of time

$$
\begin{aligned}
R = runtime &= \text{total execution time} \\
inOmp_{p,1} = \text{openmp}_p &= \text{time spent in parallel region (typically on master)} \\
inMpi_{p,t} = \text{mpi} &= \text{time spent in MPI} \\
useful_{p,t} &= \text{useful computation on thread t of process p} \\
\text{total useful} &= \text{total useful computation} = \sum_p (\sum_t (useful_{p,t})) \\
\text{serial mpi} &= \text{MPI outside of openmp} \\
suc_{p,1} = \text{serial useful} &= \text{useful computation outside of openmp} \\
ucomp_{p,t} = \text{omp useful} &= \text{useful computation inside of openmp}
\end{aligned}
$$

Notion of aggregation

$$
\begin{aligned}
avg(\text{useful}) &= \frac{\text{total useful}}{T} \\
avg(\text{serial useful}) &= \frac{\sum_p (t_p * suc_{p,1})}{T} \\
avg(\text{omp useful}) &= \frac{\text{total omp useful}}{T} = \frac{\sum_p (\sum_t (ucomp_{p,t}))}{T} \\
avg(\text{openmp}) &= \frac{\sum_p (t_p * inOmp_{p,1})}{T}
\end{aligned}
$$

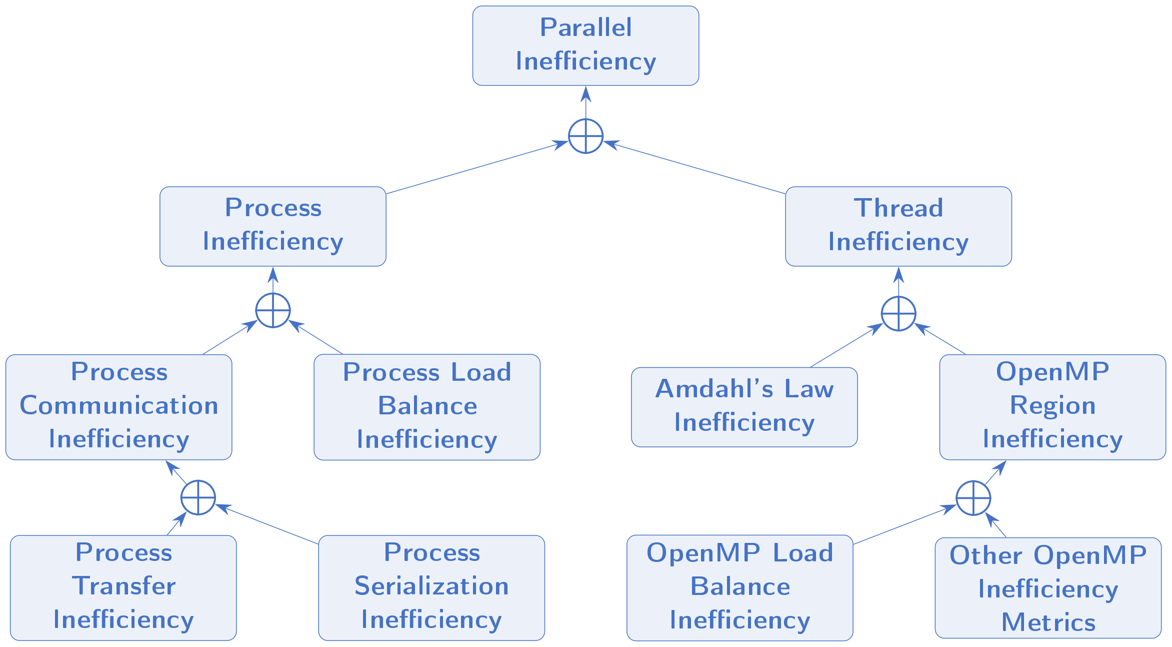
2.1.2 Additive hybrid metrics

The metrics hierarchy can be expressed either in terms of efficiencies or inefficiencies:

$$\text{Efficiency} = \frac{\text{Ideal Runtime}}{\text{Actual Runtime}}$$
$$\text{Inefficiency} = 1 - \text{Efficiency} = \frac{\text{Time spent in inefficiency}}{\text{Actual Runtime}}$$
$$\text{Parent inefficiency} = \sum \text{Child inefficiencies}$$

Ideal runtime is context-specific. It is the runtime that would be measured if the source(s) of inefficiency being considered were removed. For example: For parallel efficiency, it is the runtime achieved with zero imbalance and no overheads from parallelization, i.e., ideal runtime = average useful computation. Or for communication efficiency in pure MPI code, it is the runtime with imbalance but no MPI overheads, i.e., ideal runtime = maximum useful computation. In other words, what is considered as useful and what is considered inefficiency/overhead is totally context-specific.

Figure 2: Hierarchy of additive hybrid metrics


Parallel Inefficiency
Process Inefficiency
Thread Inefficiency
Process Communication Inefficiency
Process Load Balance Inefficiency
Amdahl's Law Inefficiency
OpenMP Region Inefficiency
Process Transfer Inefficiency
Process Serialization Inefficiency
OpenMP Load Balance Inefficiency
Other OpenMP Inefficiency Metrics

Parallel efficiency The definition of *parallel efficiency* in POP metrics assumes that parallel execution does not introduce additional computation. Under this assumption, each of the following definitions is equivalent:

$$\begin{aligned}\text{Parallel efficiency} &= \frac{\text{useful computation}}{\text{resource allocation}} = \frac{\text{total useful}}{T \cdot R} \\ &= \frac{avg(\text{useful})}{R} \\ &= \text{Process efficiency} + \text{Thread efficiency} - 1\end{aligned}$$

The latter reflects that we can split the parallel inefficiency into two contributions from process and thread inefficiency.

Process efficiency At a process level, we completely ignore the thread behavior. This means there are only three possible states:

1. Outside of OpenMP regions and in MPI
2. Outside of OpenMP regions and in useful computation
3. In OpenMP regions

At a process level, both the time within OpenMP regions (openmp) and the time within serial computation (serial comp) is considered useful, i.e., we only measure inefficiency arising from the MPI. This means thread inefficiencies (i.e., due to serial computation outside OpenMP and due to any inefficiency within OpenMP regions) need to be calculated elsewhere. The ideal runtime, in this case, would be achieved if the time within OpenMP regions and serial computation is split evenly over the processes with zero overhead from the MPI:

$$\begin{aligned}\text{Process efficiency} &= \frac{\text{time outside MPI}}{\text{resource allocation}} = \frac{avg(\text{openmp}) + avg(\text{serial useful})}{R}\\ &= \text{MPI comm efficiency} + \text{Process LB efficiency} - 1\end{aligned}$$

The latter reflects that we can split the process inefficiency into two contributions from MPI communication and process load balance inefficiency.

MPI comm efficiency In the absence of MPI communication and serialisation, the runtime would be defined by the process with the maximum amount of serial computation and OpenMP:

$$\text{MPI comm efficiency} = \frac{max(\text{openmp} + \text{serial useful})}{R}$$

This can be split into transfer efficiency and serialization efficiency in a similar way to the pure POP MPI metrics (subject to the necessary functionality being available in Dimemas) with the definition of the serialization efficiency modified to give an additive (in)efficiency.

Process LB efficiency When considering load imbalance the time cost (i.e., the difference between ideal runtime and actual runtime) is the difference between avg(openmp + serial useful) and max(openmp + serial useful), i.e.

$$\begin{aligned}&\text{Actual runtime} - \text{process load balance ideal runtime}\\ &\qquad = max(\text{openmp} + \text{serial useful}) - avg(\text{openmp} + \text{serial useful})\end{aligned}$$

$$\text{Process LB efficiency} = 1 - \frac{max(\text{openmp} + \text{serial useful}) - avg(\text{openmp}) - avg(\text{serial useful})}{R}$$

Thread efficiency At the thread level we have two sources of inefficiency to account for, which are serial computation on the master outside OpenMP, i.e., Amdahl's law inefficiency, and Efficiencies within OpenMP.

Also consider that we want:
Thread inefficiency = Parallel inefficiency - Process inefficiency

$$\text{Thread efficiency} = 1 - \frac{avg(\text{openmp}) + avg(\text{serial useful}) - avg(\text{useful})}{R}$$

Amdahl efficiency Any load imbalance at a process level is already accounted for, so we consider the time cost of thread idle time when the master threads are executing avg(serial useful). The time cost of this inefficiency (i.e., the time that would be gained by parallelizing this serial computation) is

$$\text{Amdahl efficiency} = 1 - \frac{\frac{\sum(\text{serial useful}\cdot(t_p - 1))}{T}}{R}$$

OpenMP efficiency If OpenMP regions were 100% efficient the contribution from the OpenMP regions would equal the average OpenMP useful computation (omp useful) over all threads:

$$\text{OpenMP efficiency} = 1 - \frac{avg_t(\text{openmp}_t - \text{omp useful}_t)}{R}$$

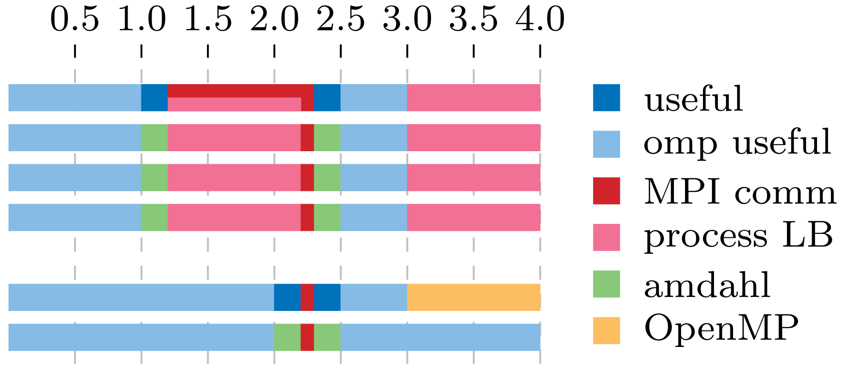
Note: $\text{openmp}_t \geq \text{omp useful}_t$ is always true!

Extensibility Ignoring the thread behavior at process level allows the model to be easily extended to include other paradigms than OpenMP (e.g. accelerators) as well as long as suitable trace data exists. We could think in terms of CUDA instead of OpenMP or even the hybrid combination of MPI+OpenMP+CUDA. However, including a paradigm for accelerators first requires to identify the concepts that are special for accelerator programming and the inefficiencies that may be introduced by them. If inefficiencies are identified they have to be modeled by defining corresponding formulas. These formulas can then be added to the metric hierarchy shown in Figure 2 as a third subtree of *Parallel Inefficiency* next to *Process-* and *Thread Inefficiency*.

2.1.3 Multiplicative hybrid metrics

The multiplicative model assumes that the efficiency metrics computed for MPI are, in fact, applicable to any parallel programming model, i.e., that parallel efficiency, communication efficiency, and global load balance are intrinsic to any parallel paradigm, with different interpretations depending on the specific programming model they refer to. Following the spirit of the initial model, the first step of the multiplicative model is to compute the metrics at the hybrid level. Those are computed like in the original model with MPI. For a hybrid code, the parallel efficiency is the average time outside both parallel runtimes MPI and OpenMP, the communication efficiency is the maximum

Figure 3: Blaming inefficiencies according to hybrid additive metrics highlighted in a trace. The calculated inefficiencies give the percentage of each color in the trace.



percentage of time outside the parallel runtimes, and global load balance at the hybrid level is computed as the ratio of the two previous metrics.

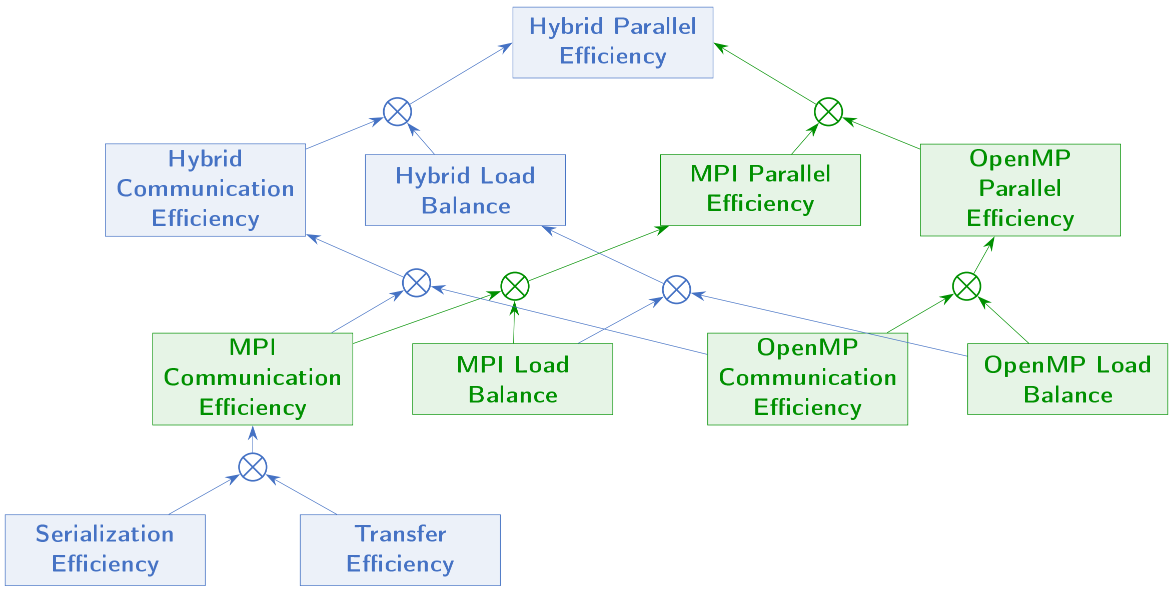
$$\text{Hybrid Parallel efficiency} = \frac{\text{total useful}}{T \cdot \text{runtime}} = \frac{avg(\text{useful})}{runtime} = \frac{\sum_{i=1}^{P} \sum_{j=1}^{t} useful_{i,j}}{P \cdot t \cdot R}$$

$$\text{Hybrid Communication Efficiency} = \frac{max(\text{useful})}{\text{runtime}}$$

$$\text{Hybrid Load Balance} = \frac{\text{Parallel efficiency}}{\text{Communication Efficiency}} = \frac{avg(\text{useful})}{max(\text{useful})}$$

Starting from these metrics, the approach targets to determine the contribution of each programming model to these three efficiencies. The next figure describes the hierarchy of the hybrid model with the multiplicative approach.

Figure 4: Hierarchy of multiplicative hybrid metrics


Hybrid Parallel Efficiency
Hybrid Communication Efficiency
Hybrid Load Balance
MPI Parallel Efficiency
OpenMP Parallel Efficiency
MPI Communication Efficiency
MPI Load Balance
OpenMP Communication Efficiency
OpenMP Load Balance
Serialization Efficiency
Transfer Efficiency

The three-level hierarchy can be traversed through two orthogonal paths: first, the programming model, then the components of the parallel efficiency or vice-versa. As a difference with the previous model, the same efficiency factors are computed for the hybrid level and each programming model. Being the same metrics allows us to apply the same model to other hybrid parallelizations like MPI+CUDA without any modification except in the interpretation of the insight provided by the results.

The approach first isolates the MPI contribution as in many cases hybrid codes also follow a hierarchical parallelization where MPI is the most external programming model. **MPI efficiencies** are determined considering only the processes or threads that call to MPI and considering that from the MPI point of view, the OpenMP parallel runtime is useful computation. Considering a simplified case where only the master threads call to MPI, the formulas are:

$$\begin{aligned}
\text{MPI Parallel efficiency} &= \frac{\text{total time outside MPI for all MPI processes}}{P \cdot R} \\
&= \frac{\sum_{(i=1)}^{P} OutsideMPI_{i,1}}{P \cdot R} \\
\text{MPI Communication efficiency} &= \frac{\text{max(time outside MPI for MPI processes)}}{\text{runtime}} \\
&= \frac{max(\{OutsideMPI_{1,1}, .., OutsideMPI_{P,1}\})}{R} \\
\text{MPI Load Balance} &= \frac{\text{Parallel efficiency}}{\text{Communication Efficiency}} \\
&= \frac{\frac{\sum_{(i=1)}^{P} OutsideMPI_{i,1}}{P}}{max(\{OutsideMPI_{1,1}, .., OutsideMPI_{P,1}\})}
\end{aligned}$$

For codes with non-hierarchical communication the model has been applied manually but the adaption of the formulation is work in progress.

For the MPI processes, it is possible to split the MPI Communication efficiency between Serialization and Transfer using the Dimemas simulator.

$$\begin{aligned}
\text{MPI Serialization efficiency} &= \frac{max(\{OutsideMPI_{1,1}, .., OutsideMPI_{P,1}\})}{R_{ideal}} \\
\text{MPI Transfer efficiency} &= \frac{\text{MPI Communication efficiency}}{\text{Serialization efficiency}}
\end{aligned}$$

Finally, **OpenMP efficiencies** (and in general any other programming model that is combined with MPI) are computed to blame any loose of efficiency that cannot be justified by the MPI activity:

$$\begin{aligned}
\text{OpenMP Parallel efficiency} &= \frac{\text{Hybrid Parallel efficiency}}{\text{MPI Parallel efficiency}} \\
\text{OpenMP Communication efficiency} &= \frac{\text{Hybrid Communication efficiency}}{\text{MPI Communication efficiency}} \\
\text{OpenMP Load Balance} &= \frac{\text{Hybrid Load Balance}}{\text{MPI Load Balance}}
\end{aligned}$$

Isolating first the MPI component implies that any time spent in MPI is blamed to MPI, although the source for it may be in OpenMP. For example, if an OpenMP loop in one of the MPI ranks is highly unbalanced causing a delay to reach the MPI synchronization, the model would report there is a problem with MPI indicating it is possible to improve the application not only modifying the OpenMP unbalance, but also modifying the MPI synchronization. In fact, it is the same effect we have when splitting the Communication efficiency between Serialization and Transfer. The simulation will blame any improvement that can be obtained, improving the network, to Transfer.

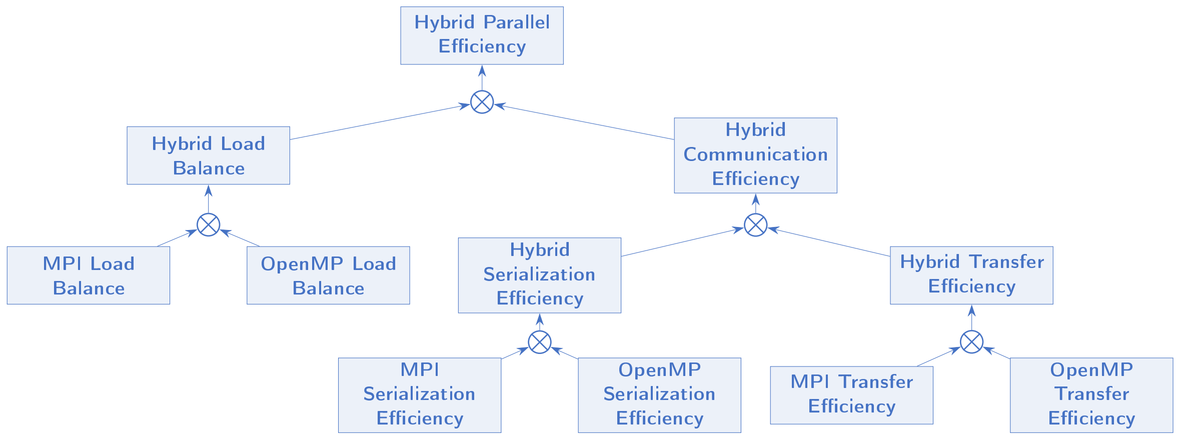
The concepts may be more directly related to the MPI paradigm, although it is essential to remark that MPI load balance efficiency is obtained computing the load balance for the whole execution. Temporal unbalance is reported as serialization. As a difference with the additive model, MPI load balance is computed as in the original MPI model with the ratio between the average and the maximum percentages, not as the average rated by the execution runtime.

In the multiplicative model, the OpenMP load balance efficiency reports the balance at a global level, making no difference if it is caused by unbalance on some of the OpenMP loops or due to serial regions not parallelized with OpenMP. The OpenMP Communication efficiency groups both temporal unbalance that is compensated along time as well as synchronization time due to barriers or overhead of the OpenMP library due, for instance, to a very low grain scheduling.

2.1.4 Critical path-based hybrid metrics

The general concept of the critical path-based hybrid metrics is to break down the global parallel efficiency metric into submetrics for each level of parallelism. In the following, we focus on the combination of MPI and OpenMP. The hierarchy of efficiency metrics is shown in Figure 5. The same concepts would also apply to combinations of MPI and CUDA or the combination of more parallel programming paradigms but will require additional formulations of submetrics for the corresponding paradigm. The definition of the critical path-based metrics will be based on an implicit representation of the *critical path*.

Figure 5: Hierarchy of critical-path based hybrid metrics


Hybrid Parallel Efficiency
Hybrid Load Balance
Hybrid Communication Efficiency
MPI Load Balance
OpenMP Load Balance
Hybrid Serialization Efficiency
Hybrid Transfer Efficiency
MPI Serialization Efficiency
OpenMP Serialization Efficiency
MPI Transfer Efficiency
OpenMP Transfer Efficiency

Critical path In literature the *critical path* is commonly understood as the event path with the longest duration in the execution history of a program. For our purpose we want to use the term critical path in a broader sense, in that we define the critical path for the execution graph of an actually observed execution of the application. The edges in the execution graph represent:

a) thread-level synchronization (e.g., from OpenMP)

b) process-level synchronization (e.g., from MPI communication) or

c) sequential execution within a process unit (e.g., executing application- or parallel runtime code).

Such execution graph is always directed and acyclic. The *critical path* between two connected nodes in a directed acyclic graph is the path with the highest sum of weights on the edges of the path. Based on this definition, we can define various critical paths through an execution graph by carefully selecting the weights for the edges.

First, we want to consider only the cost of useful execution and ignore all time in parallel runtime libraries. The resulting *critical path of useful execution* (CUE) follows all dependencies caused by synchronization but neglects the cost of communication and synchronization.

Similarly, we can define graphs, where only time in the MPI or time in the OpenMP runtime library is ignored. The *critical path of outside MPI time* (COM) has zero weights for time spent in the MPI runtime. The *critical path of outside OpenMP time* (COO) has zero weights for time spent in the OpenMP runtime.

In the following, we will not only consider the global execution graph (G-), but also process-local execution graphs (PL-) limited to the specific process and ignoring synchronization with other processes. For symmetry, we also consider thread-local execution (TL-) as a special case that ignores all synchronization with other execution units.

Definition of separated hybrid metrics In the following, t_i indicates TL-CUE on thread $i \in T$ with T the set of all threads in the execution. $PT_i \subset T$ contains all threads of process $i \in P$ with P being the set of all processes in the execution.

For load balance, we can calculate the threading load balance for each process and then take the weighted average across all processes. The weighted average of process-local averages in the numerator of the LB formula is equal to the global average across all threads. Under this consideration, we can split the global load balance into these two factors:

$$LB_{omp} = \frac{|T| \cdot \text{avg}(t_{i\in T})}{\sum\limits_{j\in P} (|PT_j| \cdot \max(t_{k\in PT_j}))} \quad \text{and} \quad LB_{mpi} = \frac{\sum\limits_{j\in P} (|PT_j| \cdot \max(t_{k\in PT_j}))}{|T| \cdot \max(t_{i\in T})} \tag{1}$$

For serialization efficiency, the question is what would be the maximum runtime of the different MPI processes, if MPI data transfers took no time. At the same time, this value should indicate the ideal process-local runtime of all processes. Based on this consideration, we can use PL-CUE to split the global serialization efficiency into two factors:

$$SerE_{omp} = \frac{\max(t_{i\in T})}{\max(\mathit{PL\text{-}CUE}_{j\in P})} \quad \text{and} \quad SerE_{mpi} = \frac{\max(\mathit{PL\text{-}CUE}_{j\in P})}{runtime_{ideal}} \tag{2}$$

G-COO contains all potential waiting time for MPI communication, and all synchronization, while OpenMP synchronization cost is dropped. Therefore, we can use this metric to split the transfer efficiency into the following two factors:

$$TE_{omp} = \frac{\mathit{G\text{-}COO}}{runtime} \quad \text{and} \quad TE_{mpi} = \frac{\mathit{G\text{-}CUE}}{\mathit{G\text{-}COO}} \tag{3}$$

Properties of separated hybrid metrics We want to highlight that all of our critical path-based hybrid metrics are values from 0 to 1, similiar to all other hybrid metrics presented in this section. In order to prove this claim, we first note that all values $t_{i\in T}$ = TL-CUE on thread $i \in T$, PL-CUE, G-COO, and G-CUE represent time on different critical paths of the execution. As such, they are always non-negative (zero or larger) since execution can only evolve forward in time. The same holds for *runtime* and $runtime_{ideal}$ of course. In the split of global load balance (1), we sum over products of TL-CUE either with $|T|$, the number of all threads in the execution, or with $|PT_i|$, the number of threads of a process i. Since both thread counts cannot be negative, the resulting products will always be non-negative. Note that the average or maximum of a set of non-negative times is trivially also non-negative. As all factors in the split of global serialization efficiency (2) and transfer efficiency (3) are simply quotients of non-negative times they are also non-negative.

To further prove that all of our critical path-based hybrid metrics cannot be larger than 1.0, we have to show that for each quotient, the numerator is less than or equal to the denominator.
For TE_{omp} we know that G-COO is the time on the global critical path only considering useful execution and MPI execution. In contrast, *runtime* is the time on the critical path additionally considering OpenMP execution. Thus we have G-COO $\leq$ *runtime*.

Similarly, for TE_{mpi} the G-CUE is time on the critical path only considering useful execution while G-COO is time on the critical path considering useful execution and MPI execution. Again we get G-CUE $\leq$ G-COO.

For $SerE_{mpi}$ assume $runtime_{ideal} < max(\text{PL-CUE}_{j\in P})$. This means there exists a process $j \in P$ that needs more time to perform its useful execution than given by $runtime_{ideal}$. By definition of ideal runtime, this cannot be as each process has to be finished with useful execution before the execution of the whole application can end. So by contradiction we get $max(\text{PL-CUE}_{j\in P}) \leq runtime_{ideal}$.

For $SerE_{omp}$ we can argue analogously assuming $max(\text{PL-CUE}_{j\in P}) < max(t_{i\in T})$. This means there exists at least one thread that spends more time on useful execution than any of the processes. However, this is not possible as each thread cannot spend more time on useful execution than the corresponding process it belongs to. So by contradiction we get $max(t_{i\in T}) \leq max(\text{PL-CUE}_{j\in P})$.

For LB_{omp} we can argue by definition of the average

$$|T| \cdot avg(t_{i\in T}) = \sum_{i\in T} t_i = \sum_{j\in P}\left(\sum_{k\in PT_j} t_k\right) \leq \sum_{j\in P}\left(|PT_j| \cdot \max(t_{k\in PT_j})\right).$$

For LB_{mpi} we can argue that we have

$$\text{for each } j \in P \,:\, \max(t_{k\in PT_j}) \leq \max(t_{i\in T_i})$$

and thus also

$$\sum_{j\in P}\left(|PT_j| \cdot \max(t_{k\in PT_j})\right) \leq \sum_{j\in P}\left(|PT_j| \cdot \max(t_{i\in T})\right) = |T| \cdot \max(t_{i\in T}).$$

2.1.5 Comparison of the different hybrid metrics

We want to briefly compare our proposed hybrid metrics by highlighting strengths and weaknesses of the different models. First of all, a big disadvantage of having multiple hybrid efficiency models is the difficulty to properly summarize all the metric data that was collected and analyzed during the lifetime of the project. With the original metric set it is easily possible to compare multiple applications based on their obtained efficiencies. For the hybrid metrics only the top level metrics may be directly compared to each other.

The *process efficiency* and *thread efficiency* in the additive hybrid model may be compared to *MPI parallel efficiency* and *OpenMP parallel efficiency* in the multiplicative model. But further down the metric hierarchy on the thread level the *OpenMP load balance* in the multiplicative model cannot clearly be related to either *Amdahl efficiency* or *OpenMP region efficiency* in the additive model because the effects related to both of the mentioned additive metrics are captured by the multiplicative *OpenMP load balance*.

However, there are also important strengths of the proposed hybrid models that justify the coexistence of both the additive and the multiplicative models. While we focus to demonstrate the hybrid models by the example of MPI+OpenMP applications we want to stress that all of the models support other combinations of programming paradigms, e.g., MPI+CUDA, as well. The strength of the multiplicative model is that it naturally supports any combination of MPI+X, where X may be any kind of threading or accelerator programming paradigm. The additive hybrid model as well as the critical path-based model will need to explicitly reformulate the OpenMP related metrics to be applied to CUDA, for example.

Moreover, the definition of the critical path-based hybrid metrics using the implicit critical path allows to calculate the metrics in an online fashion during program execution. Since the critical path only needs to be determined implicitly during runtime a performance analysis tool implementing this approach only needs to collect a neligable amount of timers. No trace of the whole application run needs to be tracked and stored such that the runtime overhead introduced by this approach is also neligable. This is a very important feature in order to avoid distortion of the original program behavior by the analysis tool.

While the multiplicative hybrid model might be the easiest to be applied to a multitude of different programming paradigms combined with MPI, it only offers a global view on the performance of a hybrid program execution. In most cases this is enough to already indicate from which part of the hybrid programming model inefficiencies arise. For a deeper analysis the additive hybrid model might reveal the root cause of inefficiencies more easily. On the OpenMP level, for example, the initial multiplicative model may reveal that there is a load imbalance among all the threads. In contrast the additive model easily allows to split the inefficiency coming from load imbalance into individual contributions from each OpenMP region. This way both the POP analyst as well as the customer know on which part of the application code the analysis can be focused. The critical path-based hybrid metrics might also provide more insight compared to the multiplicative model because it additionally enables the split of *communication efficiency* into *serialization*- and *transfer efficiency* for each programming model individually.

Lastly, we want to give some guidelines on choosing the most appropriate model. To get a first overview of the efficiency of a hybrid parallel application all three models can be used equally good. The only exception here are hybrid applications using other combinations of programming paradigms than MPI+OpenMP. In the current state at the end of the project only the multiplicative hybrid metrics support such cases per default. If inefficiencies are identified to arise from the thread level we recommend to use the additive hybrid metrics because being able to break down the inefficiencies for each parallel region easily identifies the inefficient regions of the code. If the hybrid application implements a non-hierarchical MPI communication scheme such as `MPI_THREAD_MULTIPLE` or heavily makes use of OpenMP tasks with task dependencies the critical path-based metrics should be used as they can break down the *communication efficiency* into *serialization* and *transfer efficiency* separately for each programming model.

2.2 I/O metrics

The advent of big data and machine learning has increased the importance of efficient I/O performance inside applications. Currently, the application's I/O performance is addressed in the special section of the POP assessment only when it is already clear that the I/O part of the application is the main factor for the inefficiencies. Therefore, a quantitative way to show the I/O impact is needed as part of the POP metrics. We used as a starting point the work done under the DEEP-ER project[1], where the I/O efficiency was computed considering its weight with respect to the computations. While the details of how to integrate file I/O into POP metrics are still under consideration, we present the first concepts derived from the discussion in the following.

[1]https://www.deep-projects.eu/images/materials/DEEP-ER_D72_Report_on_projections_and_improvements_for_the_DEEP_DEEPER_concept_v20-ECapproved-.pdf

File I/O Efficiency. A quantitative definition of File I/O efficiency might look like:

$$\text{File I/O Efficiency} = \frac{avg(\text{useful comp. time})}{avg(\text{useful comp. time} + \text{file I/O time})}$$

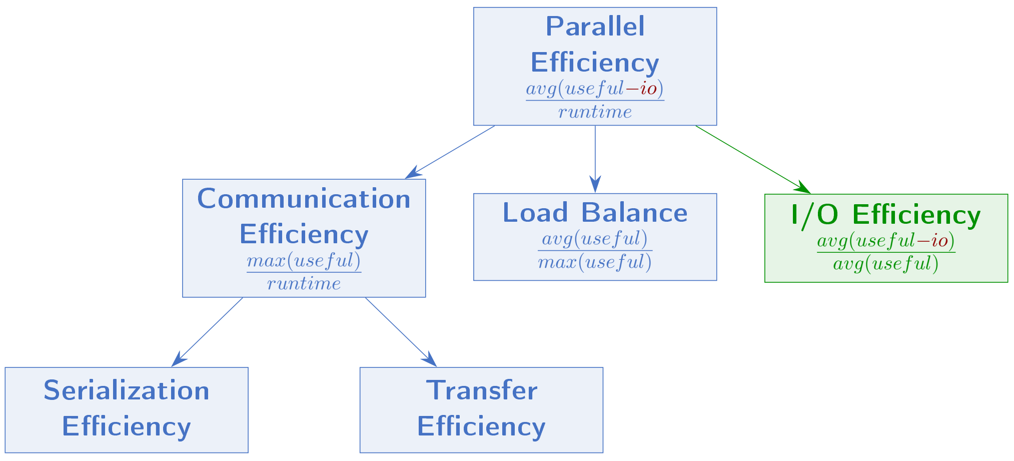
This assumes that we do not account for File I/O to be useful computation time. As an alternative, if we account File I/O to be useful computation time, we could also write the same term as:

$$\text{File I/O Efficiency} = \frac{avg(\text{useful comp. time} - \text{file I/O time})}{avg(\text{useful comp. time})}$$

Integration in POP metrics. We can find arguments for two different ways to integrate File I/O into the tree of POP metrics. It is arguable whether File I/O is a characteristic of a parallel application, or whether it is a separate performance problem and should not be mixed with parallel performance.

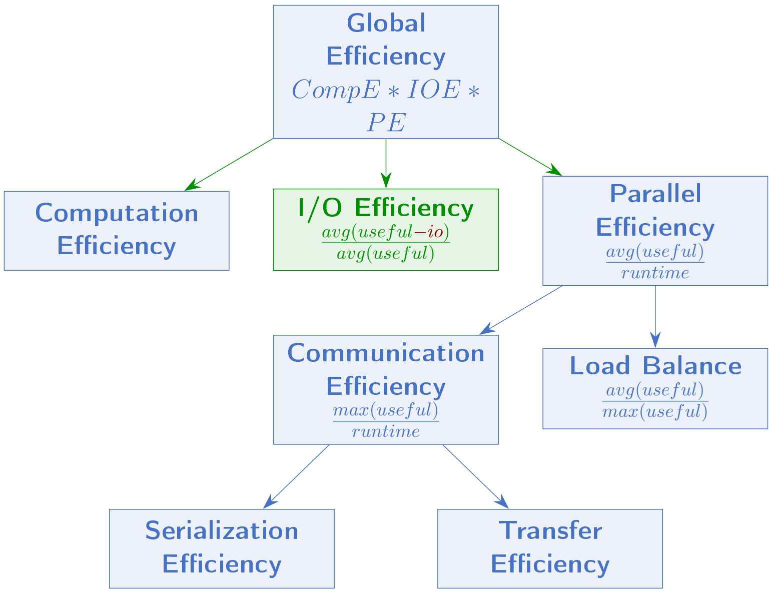
In the first version, shown in Figure 6, we introduce File I/O as a sub metric of Parallel Efficiency. An argument for this view is that File I/O in a parallel application influences the parallel behavior of the application. As an example, sequential output by a root rank can lead to a load imbalance solely caused by the I/O. With a better I/O system or by turning off the logging, the load balance could be improved. The resulting tree of metrics is visualized Figure 6.

Figure 6: I/O as a sub metric of Parallel Efficiency


Parallel Efficiency
$\frac{avg(useful-io)}{runtime}$
Communication Efficiency
$\frac{max(useful)}{runtime}$
Load Balance
$\frac{avg(useful)}{max(useful)}$
I/O Efficiency
$\frac{avg(useful-io)}{avg(useful)}$
Serialization Efficiency
Transfer Efficiency

An alternative, shown in Figure 7, is to interpret I/O efficiency as a global characteristic of the application. This view highlights the independence of the parallel execution from the influence of the I/O behavior of the application. The resulting tree of metrics is visualized Figure 7.

Figure 7: I/O as a sub metric of Global Efficiency



2.3 Vectorization metrics

Vectorization is one of the most powerful levers when considering performance optimization at the core level. Modern processors like Intel Xeon Skylake and next-generation ARM-based Fujitsu cores (SVE extension) feature 512 bits wide FPUs. Enabling vectorization can lead up to a 16x speedup when dealing with single (32bit) floating-point values and 8x with double (64bit) ones. That is why assessing vectorization efficiency is very important.

Vectorization metrics are usually computed at the loop level and, to a lesser extent, at function level when loops are inlined in small functions (C++ like behavior). It is still possible to build a global metric at the program level in order to provide a rough idea of the global behavior. Such global vectorization efficiency can be part of the computation efficiency.

As a complement we also compute a vectorization intensity metric to get an idea of how much instructions are actually vectorized.

Vectorization efficiency would be "How well is the code vectorized" and vectorization intensity "How much". For instance, if vectorization efficiency is 100% but only corresponds to 5% of total execution time, the information is meaningful.

We also introduced an OPC (Operations Per Cycle) metric that better reflects the vectorization efficiency when compared to IPC. Basically, for a given basic block if instructions, OPC = IPC * OPI (where OPI is Operations Per Instructions). It is not specific to vectorization but on x64 platforms it's clearly vectorization that provides data level parallelism. Whether we consider it at global or local level it permits getting a quick insight at the instruction level when it comes to vectorization efficiency. For example the CS3D code (cf. POP2_AR_077) heavily uses MKL. On 200 cores it has an average IPC of 1.12 while OPC is 5.77. The problem is not IPC, it is just that it is not meant to reflect operation (what users think about) but the number of instructions executed (including multi-operation instructions like vector ones).

Assessing vectorization requires detecting vector instructions. Vector instructions are characterized by their precision (*single* or *double*), their level of parallelism (*scalar*, i.e., one value, or *packed* , i.e., multiple values in a vector) and their width (e.g., 512bits).

The most common metric is the vectorization ratio. For each data precision (single/double) the amount of packed instructions over all vector instructions is computed (packed / (scalar + packed)). The goal is to estimate the number of packed instructions. This definition is used, for instance, in compilers. Typically, when a compiler states ”loop fully vectorized“, that does not mean it is exploiting the full architecture's vector width, but the considered instruction set. For instance, if the compiler is using AVX2 instructions on an AVX512 capable chip, then the loop would be fully vectorized but only using half of the actual vector's length. That is why we have to introduce vector width utilization. We call this metric vectorization efficiency.

Technically, there are two ways to deal with vector instructions. On the one hand, hardware counters can be used to count the number of arithmetic floating-point (FP) operations (single/double and scalar/packed for 32/64/128/256/512 operands). The two main limitations of this approach are:

- FP counters may not be available (dependent on processors/architectures/microarchitectures);
- it requires using eight counters, which usually requires an additional run if other counters need to be collected and also using multiplexing (more events than hardware slots)

On the other hand, it is possible to use tools like MAQAO [2] or Intel Advisor [3] implementing a static analysis (disassembling the application's binary). Using static analysis additionally provides loop

[2] https://www.maqao.org/
[3] https://software.intel.com/en-us/advisor

level granularity, integer vector operations detection, and also checking vector efficiency of vector memory operations, which cannot always be counted by hardware counters (as vector instruction).

We will first start by the method that can be implemented by any POP partner developing tools, that is to say the hardware counters approach.

For a typical x86 architecture with 512-bit vector registers, the definition of vectorization efficiency would be:

$$\text{Veff} = \frac{\begin{array}{c}100 * (32 * \text{SS} + 64 * \text{SD} + 128 * (\text{PS128} + \text{PD128}) + 256 * (\text{PS256} + \text{PD256}) + \\ 512 * (\text{PS512} + \text{PD512}))\end{array}}{512 * \text{total FP operations}}$$

SS = Scalar single-precision FP
SD = Scalar double-precision FP
PS = Packed single-precision FP
PD = Packed double-precision FP
For instance, PS128 means 128-bit packed single-precision FP instruction (4 * 32)

Every vector instruction is weighted (i.e., contribution) by its length in bits (numerator). Then, all instructions' contributions are compared to the peak vector performance (denominator).

Implementing this formula using hardware counters depends on the availability of these counters. Considering Intel products, the required counters are available starting from Skylake Xeon (Server) processors (were stopped since Nehalem).

The other method consists in mixing dynamic and static analysis (MAQAO [4] LProf + CQA). While providing the same feature, this method has the following advantages :

- avoids collecting vector events
- track memory operations while hardware counter tracks only vector FP operations
- works with in the case of AMD which counters implementation does not provided the vector width
- supports codes with integer vector operations which are not tracked by hardware counters

Similar to the I/O metrics, these considerations on vector metrics are currently in a draft state. Regarding the integration into the POP metric hierarchy, the vectorization will go into computation efficiency, as this is a characteristic that also applies to the serial execution.

2.4 Extensions used in practice

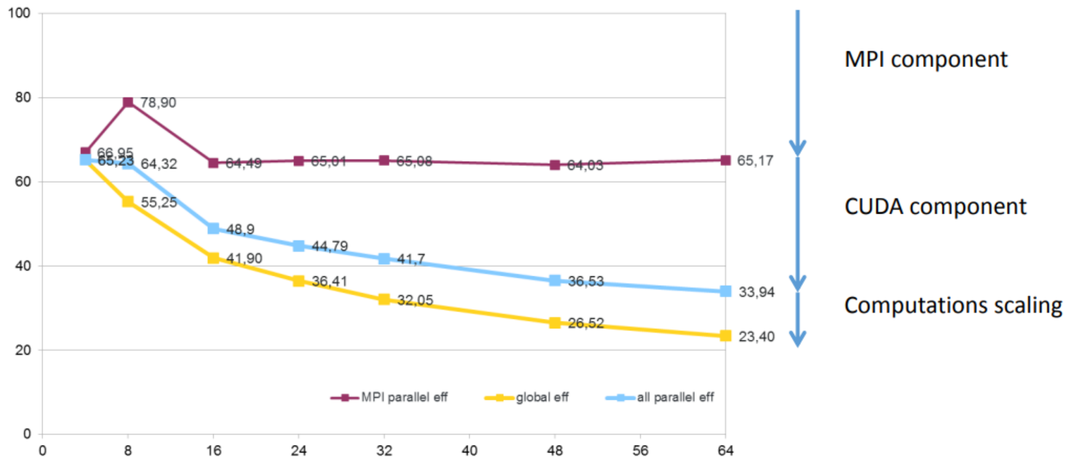
Our proposed extensions to the methodology have been used in the performance assessments from workpackage 5. Here we want to show the usefulness of our extensions in practice by highlighting some interesting performance assessments, where our metric extensions clearly helped to identify significant performance issues in hybrid application codes.

Multiplicative hybrid metrics on TsunamiHySEA

In the first assessment of the hybrid MPI+CUDA TsunamiHySEA (cf. POP2_AR_003) code hybrid metrics were not defined yet. In order to determine whether inefficiencies are caused by the MPI or the CUDA component the contribution of the MPI component to the *global efficiency* of the code was determined with the original POP MPI metrics and the remaining inefficiency was blamed to the CUDA component. The result of this breakdown is shown in Figure 8.

[4] https://www.maqao.org/

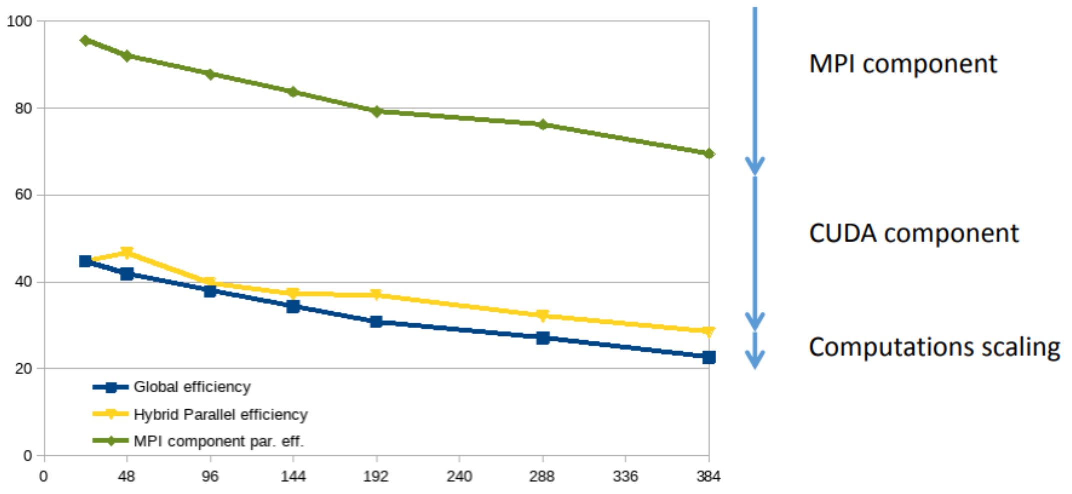
Figure 8: Breakdown of global efficiency (y-axis) for TsunamiHySEA (1st assessment, 1 MPI rank per GPU (x-axis)



For each run the code was executed with 1 GPU per MPI rank. One can recognize that the MPI parallel efficiency (red line) stays rather constant while the parallel efficiency considering both MPI and CUDA (blue line) decreases significantly with scale. Further analysis revealed that the scaling of the CUDA parallelization is limited because the overhead of memory copies and kernel launches increases at scale. On the MPI level this manifests as waiting time in an MPI_Allreduce operation due to the memory copies from the devices not being finished.

In fact this assessment motivated the development of the multiplicative hybrid metrics in order to verify that the performance issues are related to the CUDA parallelization. A second assessment was performed on TsunamiHySEA (cf. POP2_AR_085) with the same input data and the same hardware but with some code modifications by the developer based on the findings of the first assessment. The breakdown of the *global efficiency* based on the multiplicative hybrid metrics obtained in this audit is shown in Figure 9.

Figure 9: Breakdown of global efficiency (y-axis) for TsunamiHySEA (2nd assessment), 1 MPI rank per GPU (x-axis)



In this case both the MPI parallel efficiency (green line) as well as the hybrid parallel efficiency (yellow line) decrease significantly with scale. However, it is noticeable that the CUDA component contributes more to the loss of efficiency than the MPI component. This points further analysis to investigate the CUDA level.

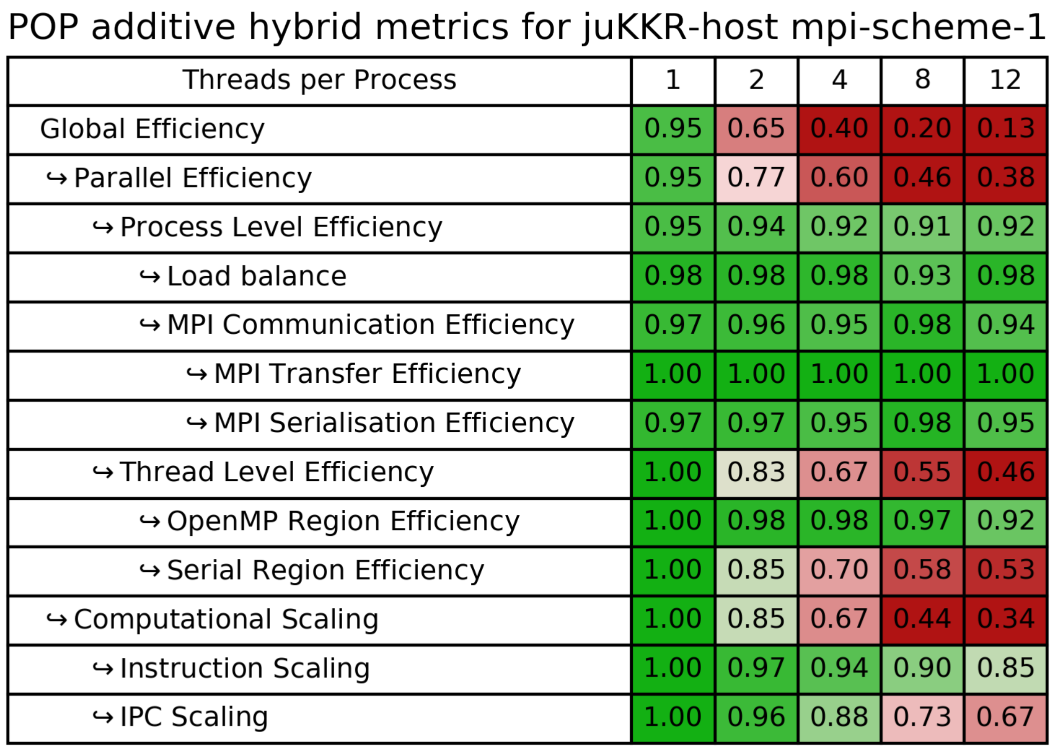
Further analysis revealed that point to point communication are now overlapped with CUDA kernels but still the same waiting time in the MPI_Allreduce operation occurs similar to the first audit of this code. The code developers removed this Allreduce operation and the prior delay after the execution of the CUDA kernels has disappeared.

So in this case the multiplicative hybrid metrics clearly helped the code developers to understand that there is a problem related to their CUDA parallelization, which could then be solved afterwards.

Additive hybrid metrics on JuKKR

For the analysis of the hybrid MPI+OpenMP application JuKKR (cf. POP_AR_079) we applied the additive hybrid metrics. The results are shown in Figure 10.

Figure 10: Additive hybrid metrics for JuKKR (16 MPI ranks, varying OpenMP threads)



POP additive hybrid metrics for juKKR-host mpi-scheme-1

Threads per Process	1	2	4	8	12
Global Efficiency	0.95	0.65	0.40	0.20	0.13
↪ Parallel Efficiency	0.95	0.77	0.60	0.46	0.38
↪ Process Level Efficiency	0.95	0.94	0.92	0.91	0.92
↪ Load balance	0.98	0.98	0.98	0.93	0.98
↪ MPI Communication Efficiency	0.97	0.96	0.95	0.98	0.94
↪ MPI Transfer Efficiency	1.00	1.00	1.00	1.00	1.00
↪ MPI Serialisation Efficiency	0.97	0.97	0.95	0.98	0.95
↪ Thread Level Efficiency	1.00	0.83	0.67	0.55	0.46
↪ OpenMP Region Efficiency	1.00	0.98	0.98	0.97	0.92
↪ Serial Region Efficiency	1.00	0.85	0.70	0.58	0.53
↪ Computational Scaling	1.00	0.85	0.67	0.44	0.34
↪ Instruction Scaling	1.00	0.97	0.94	0.90	0.85
↪ IPC Scaling	1.00	0.96	0.88	0.73	0.67

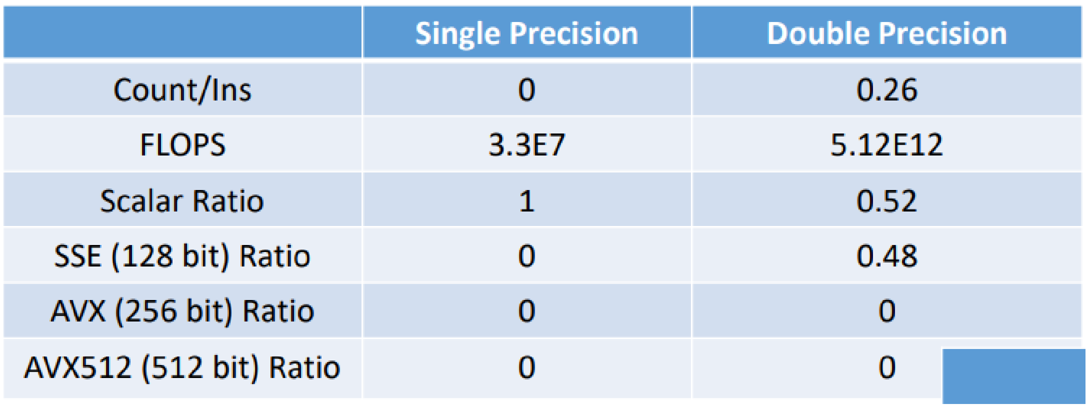
The additive hybrid metrics clearly show that there is no issue on the process level as most efficiencies are nearly optimal. However, on the thread level we recognize a significant decrease of the *Serial Region Efficiency* when the number of OpenMP threads used per MPI rank is increased. This child-metric tells us that there is a significant amount of the code that only runs in serial on each MPI rank but does not exploit the OpenMP parallelism.

After closer investigation we indeed found an inefficient parallelization using the threaded version of the Intel Math Kernel Library (MKL) in one of the hotspots of the application. We addressed this issue in a proof of concept (cf. POP2_POCR_026). We moved the thread parallelism out of the Intel MKL and implemented OpenMP parallelism on a higher level in the call-tree of this hotspot region. Our optimization yielded a speedup of roughly 6x in this proof of concept study.

Vectorization metrics on DISCO

As part of the performance assessment for the Disco code (cf. POP2_AR_123) the vectorization has been investigated. The results are shown in Figure 11.

Figure 11: Vectorization metrics for the Disco application



	Single Precision	Double Precision
Count/Ins	0	0.26
FLOPS	3.3E7	5.12E12
Scalar Ratio	1	0.52
SSE (128 bit) Ratio	0	0.48
AVX (256 bit) Ratio	0	0
AVX512 (512 bit) Ratio	0	0

Our vectorization metrics indicate that the code mostly performs double precision floating-point operations. Only 48% of those are vectorized using SSE (128 bit) instructions while the remaining 52% are scalar instructions. Here the vectorization metrics clearly indicate that there is huge potential to optimize the application by further vectorization of the code.

This has been done in a proof of concept (cf. POP2_POCR_021) activity. Several functions of the code have been identified which are suitable for further vectorization. By changing the order of three nested loops and also modifying the layout of the corresponding data structures used in these loops the execution time of these function was reduced by 70%.

3 Improvements made to the tools

Since the first phase of the POP CoE, the extensive use of performance tools from BSC and JSC in a wide range of parallel application execution assessment services undertaken by POP partners identified limitations and unexpected behavior that were important to address. Valuable feedback from POP partners also identified new functionalities to facilitate expanding and improving the delivery of POP services. In this context, additional tools available from UVSQ and RWTH providing complementary capabilities have also been included, along with a new reporting tool being developed by NAG. Our proposals to extend the POP metrics for hybrid use-cases, vectorization and file I/O also necessitated important development efforts in order to make these extensions usable in practice.

This section details all improvements made to tools related to POP services over the whole project period of POP2. We also briefly give an outlook on current work in progress.

3.1 Correctness Tools

The tools Archer and MUST are runtime correctness analysis tools, which observe the execution of a parallel program to detect erroneous behavior. The focus of Archer is multi-threaded applications, while the focus of MUST is MPI parallel applications. Current efforts target integrating the two tools to allow for analysis of hybrid MPI+OpenMP applications.

3.1.1 Archer

The Archer tool is used to detect data races in OpenMP applications. It builds on ThreadSanitizer 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. With the release of LLVM 10.0 in March 2020, Archer gets distributed to supported platforms when the latest clang is installed with OpenMP support. Moreover, an issue regarding data dependencies for OpenMP tasks was identified because these dependencies were synchronized globally. Efforts were made to improve the Archer tool such that it correctly synchronize task dependencies only for sibling tasks.

3.1.2 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.

As MUST is applied in more and more POP assessments, we identify a lack in support for specific functions introduced in MPI-3. If such unhandled MPI functions would provide crucial information for subsequent correctness analysis, like constructors for MPI opaque handles, we cannot apply MUST analysis for code using those functions. In this case, essential support for the identified MPI function gets implemented while moving forward with the POP assessment.

The 1.6 release of MUST includes increased coverage of MPI-3 features, but also provides basic thread-safety for the analysis of multi-threaded MPI applications. Moreover, configuring MUST with stack trace support, which helps MUST to generate more detailed and meaninful reports, has been simplified by providing an alternative stack trace implementation based on backward-cpp. Previously, installation of external software was required to enable stack trace support.

Release 1.8 further extends support for multi-threaded MPI applications with the integration of ThreadSanitizer into MUST to enable data race detection. Since only GNU- and LLVM-based compilers provide ThreadSanitizer instrumentation, such analysis is limited to applications compiled with either of those compilers. In addition, the type and memory allocation tracker TypeART has been integrated into MUST to improve the detection of local type mis-matches in MPI function calls. Other new features of MUST added in release 1.8 improve its usability by allowing users to filter MUST's output and an option to specify the output directory. Another milestone for release 1.8 was setting up continuous integration in order to deliver more robust releases and enhance MUST's compatibility with different MPI implementations and versions.

Work is in progress to integrate Archer and MUST analysis, to advance the detection of multi-threading issues in MPI applications. Current efforts also revolves around further extending the initial continuous integration for MUST.

3.2 Score-P-based tools

The Score-P instrumentation and measurement infrastructure generates runtime summary profiles (in CUBE format) and/or event traces (in OTF2 format) for parallel executions. Event traces can be interactively examined with trace visualization tools such as Vampir and automatically analyzed for execution inefficiencies with the Scalasca Trace Tools, which produce augmented (CUBE format) profiles containing additional metrics. CUBE tools are employed to post-process profiles, in particular, to derive additional metrics and construct a metric hierarchy. A CUBE GUI facilitates interactive exploration.

Within this POP2 work package, most effort focused on improving the use of the Scalasca nexus and CUBE GUI to facilitate assessments, along with targeted developments to Score-P driven by particular assessment issues. Scalasca and Score-P also benefited from contributions from other projects and partners from their developer communities.

3.2.1 Scalasca

The nexus provided by the Scalasca Trace Tools toolset for combined measurement collection and analysis of an application execution (SCAN) was extended to support sets of measurements. This multi-run capability can automate the generation of multiple measurements with varying application execution and measurement configuration settings. One usage supports measurements with different sets of hardware counters, specifically to address the limited numbers and combinations of counters, which can be recorded simultaneously in a single measurement. Various preset configurations are provided, including one specifically for POP standard assessments. Additionally, the SQUARE analysis report explorer was extended to post-process and integrate multi-run sets of experiments into an aggregate analysis report. Report post-processing has also been extended to construct metric hierarchies for CUDA, OpenCL and OpenACC overheads.

Work in progress will allow automated analysis of traces containing events for non-CPU locations, such as those used for GPU metrics and process memory.

3.2.2 Score-P

Several POP2 performance assessments have been applications employing C++ standard threads, which are POSIX threads originating from the C++ runtime library. Events from these threads (and others where the point of creation is not instrumented) are now handled and presented for (parent-less) orphan thread locations. While OpenMP has been the predominant threading model for assessed applications, a number of applications combine use of OpenMP with other threading models (typically C++ standard or POSIX threads) and these combinations are not yet supported by Score-P, restricting analysis to one model only.

Support for MPI measurement was extended to include recording RMA/one-sided communication events of MPI-3. Cartesian topologies from MPI communicators along with hardware and user-defined topologies are also recorded to support additional analysis presentation. Receives of messages matched by MPI probe routines (`MPI_Imrecv` and `MPI_Mrecv`) as used by Python/mpi4py applications in POP assessments have been added. Bytes read or written by MPI file I/O routines are included along with newly added POSIX file I/O events to support file I/O assessments. Work is underway to support the Fortran2008 MPI library interface module (`mpi_f08`) and make the Score-P MPI adapter thread-safe to be able to support the MPI_THREAD_MULTIPLE model where multiple threads of a process concurrently use MPI calls. Additionally, work is in progress to support inter-communicator events used by multi-executable (MPMD) applications, implement missing events/attributes such as those for `MPI_Comm_create`, `MPI_Alltoallv` and `MPI_Alltoallw` and relations, and incorporate aggregation/filtering of uncompleted `MPI_Iprobe`, `MPI_Test` and related test events.

Support has been added to support measurement on AMD Epyc processors, and work is in progress to support AMD GPUs and ARM A64FX systems (with their associated compilers and libraries), and to support OMPT events from OpenMP runtime systems.

Support for many of the compilers typically used on HPC computer systems has been improved to be able to instrument applications more reliably. This includes entirely new interfaces such as Kokkos, and the new version of the OpenCL interface.

Intel compiler instrumentation can now be controlled via a specification file (as previously for GCC compilers). The utility for scoring measurement based on their analysis report content has been extended with additional sorting modes and the ability to automatically generate a filter file which can be customized and used for subsequent instrumentation or measurements.

Identification of GPU devices is being improved, race conditions removed from the GPU event handlers, and avoid conflicts with NVTX region annotations. Wrapping of external libraries to capture their API events is being made more convenient.

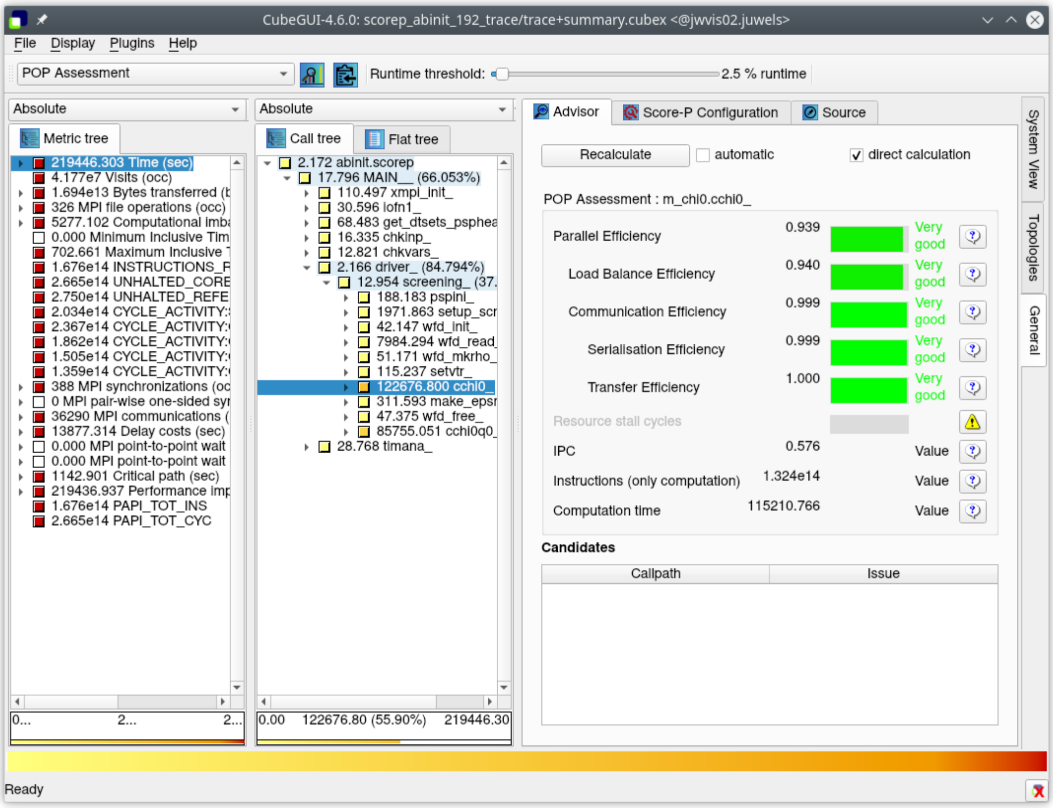
3.2.3 Cube

A key development for POP has been the incorporation of POP efficiency metrics, derived from the metrics contained in Scalasca and Score-P analysis reports, leading to the GUI Advisor plug-in presentation of the set of metrics associated with a selected call-path (or set of call-paths) as shown in Figure 12. This facilitates determination, exploration, and extraction of the POP efficiency metrics for a known or prospective assessment focus of analysis (FoA). Work is in progress to extract derived POP efficiency metrics for a specified callpath from one or a set of analysis reports via a command-line utility to enable automation of this currently interactive multi-step procedure.

Figure 13 shows variants of POP efficiency metric derivations for hybrid MPI+OpenMP analysis provided by different Advisor modes. Work in progress is extending these derivations for hybrid CPU+GPU analysis (such as MPI+CUDA), and more general combinations of parallelization paradigms (e.g. MPI+OpenMP+CUDA), and automated use of the appropriate Advisor mode based on analysis content.

Clockrate and computation granularity metrics, as well as prototype File I/O and Vectorization efficiencies, are implemented as derived metrics for evaluation, as shown in Figure 14, providing metric values for individual call-paths and processes/threads. These standard assessment metrics are expected to be incorporated in the Advisor panel presentation in due course.

While MPI File I/O is automatically collected and analysed, POSIX File I/O requires special additional instrumentation and measurement and is therefore often missing (particularly for codes written in Fortran). Derivations such as those shown for Vectorization rely on processor-specific hardware counters, that may not be available/complete on each computer system or require multi-


CubeGUI-4.6.0: scorep_abinit_192_trace/trace+summary.cubex <@jwvis02.juwels>
File Display Plugins Help
POP Assessment
Runtime threshold: 2.5 % runtime
Absolute
Metric tree
Call tree
Flat tree
Advisor
Score-P Configuration
Source
System View
Topologies
General
Recalculate
Ready

```
219446.303 Time (sec)
4.177e7 Visits (occ)
1.694e13 Bytes transferred (b
326 MPI file operations (occ)
5277.102 Computational imba
0.000 Minimum Inclusive Tim
702.661 Maximum Inclusive T
1.676e14 INSTRUCTIONS_R
2.665e14 UNHALTED_CORE
2.750e14 UNHALTED_REFE
2.034e14 CYCLE_ACTIVITY:
2.367e14 CYCLE_ACTIVITY:
1.862e14 CYCLE_ACTIVITY:
1.505e14 CYCLE_ACTIVITY:
1.359e14 CYCLE_ACTIVITY:
388 MPI synchronizations (oc
0 MPI pair-wise one-sided syr
36290 MPI communications (
13877.314 Delay costs (sec)
0.000 MPI point-to-point wait
0.000 MPI point-to-point wait
1142.901 Critical path (sec)
219436.937 Performance imp
1.676e14 PAPI_TOT_INS
2.665e14 PAPI_TOT_CYC
```

```
2.172 abinit.scorep
  17.796 MAIN__ (66.053%)
    110.497 xmpi_init_
    30.596 iofn1_
    68.483 get_dtsets_psphea
    16.335 chkinp_
    12.821 chkvars_
    2.166 driver_ (84.794%)
      12.954 screening_ (37.
        188.183 pspini_
        1971.863 setup_scr
        42.147 wfd_init_
        7984.294 wfd_read_
        51.171 wfd_mkrho_
        115.237 setvtr_
        122676.800 cchi0_
        311.593 make_epsr
        47.375 wfd_free_
        85755.051 cchi0q0_
    28.768 timana_
```

0... 2... 2...

0.00 122676.80 (55.90%) 219446.30

automatic [✓ direct calculation]

POP Assessment : m_chi0.cchi0_

Parallel Efficiency	0.939	Very good
Load Balance Efficiency	0.940	Very good
Communication Efficiency	0.999	Very good
Serialisation Efficiency	0.999	Very good
Transfer Efficiency	1.000	Very good
Resource stall cycles		
IPC	0.576	Value
Instructions (only computation)	1.324e14	Value
Computation time	115210.766	Value

Candidates

Callpath	Issue

Figure 12: Advisor panel within CUBE GUI showing assessment of parallel efficiency and associated computation metrics for selected callpath from combined trace and summary measurements. For this MPI application execution callpath communication efficiency is perfect, but IPC instruction intensity is poor.

ple measurements, therefore generalising this mechanism is work in progress.

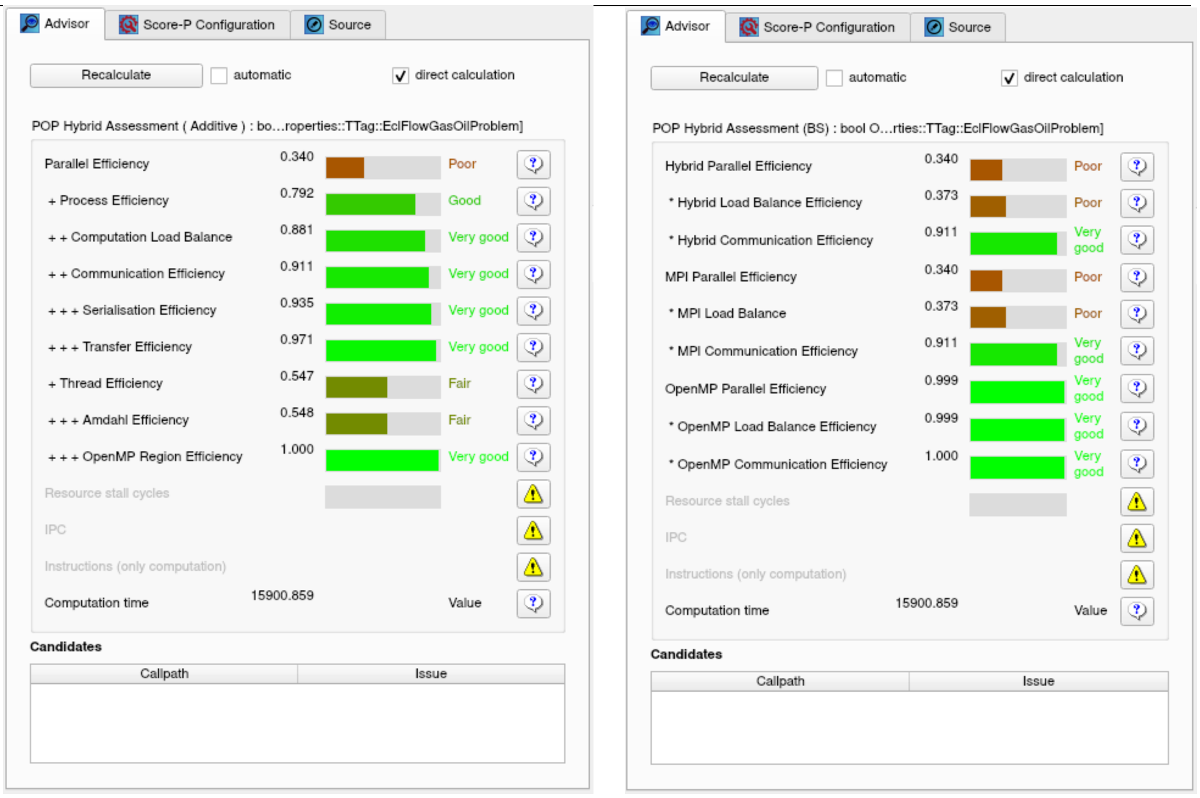
Additionally, the GUI now supports bookmarking to store and return to notes made during analysis exploration, and the underlying library is being hardened to support the asynchronous calculation of metric values for improved reliability and responsiveness. The presentation of asynchronously executed OpenMP tasks and kernels offloaded to GPUs has been made more consistent. An additional display providing an overview of the contents of a summary or trace analysis report has been prototyped for a future release.

3.3 Paraver-based tools

The BSC Performance Tools are extensively used in POP CoE. Some of the studies allow identifying the need to extend the currently supported scenarios as well as detecting unexpected behavior. The feedback from the POP partners is also very valuable to identify new functionalities or aspects that can be improved. In this section, we detail the improvements made to the BSC Performance Tools, including the current work in progress.

3.3.1 Extrae

Extrae is the instrumentation package for BSC Performance Tools, as such it concentrates most of the effort to extend the support to programming models and architectures. With respect to the



POP Hybrid Assessment (Additive) : bo…roperties::TTag::EclFlowGasOilProblem]

Parallel Efficiency	0.340	Poor
+ Process Efficiency	0.792	Good
+ + Computation Load Balance	0.881	Very good
+ + Communication Efficiency	0.911	Very good
+ + + Serialisation Efficiency	0.935	Very good
+ + + Transfer Efficiency	0.971	Very good
+ Thread Efficiency	0.547	Fair
+ + + Amdahl Efficiency	0.548	Fair
+ + + OpenMP Region Efficiency	1.000	Very good
Resource stall cycles		
IPC		
Instructions (only computation)		
Computation time	15900.859	Value

Candidates

Callpath	Issue

POP Hybrid Assessment (BS) : bool O…rties::TTag::EclFlowGasOilProblem]

Hybrid Parallel Efficiency	0.340	Poor
* Hybrid Load Balance Efficiency	0.373	Poor
* Hybrid Communication Efficiency	0.911	Very good
MPI Parallel Efficiency	0.340	Poor
* MPI Load Balance	0.373	Poor
* MPI Communication Efficiency	0.911	Very good
OpenMP Parallel Efficiency	0.999	Very good
* OpenMP Load Balance Efficiency	0.999	Very good
* OpenMP Communication Efficiency	1.000	Very good
Resource stall cycles		
IPC		
Instructions (only computation)		
Computation time	15900.859	Value

Candidates

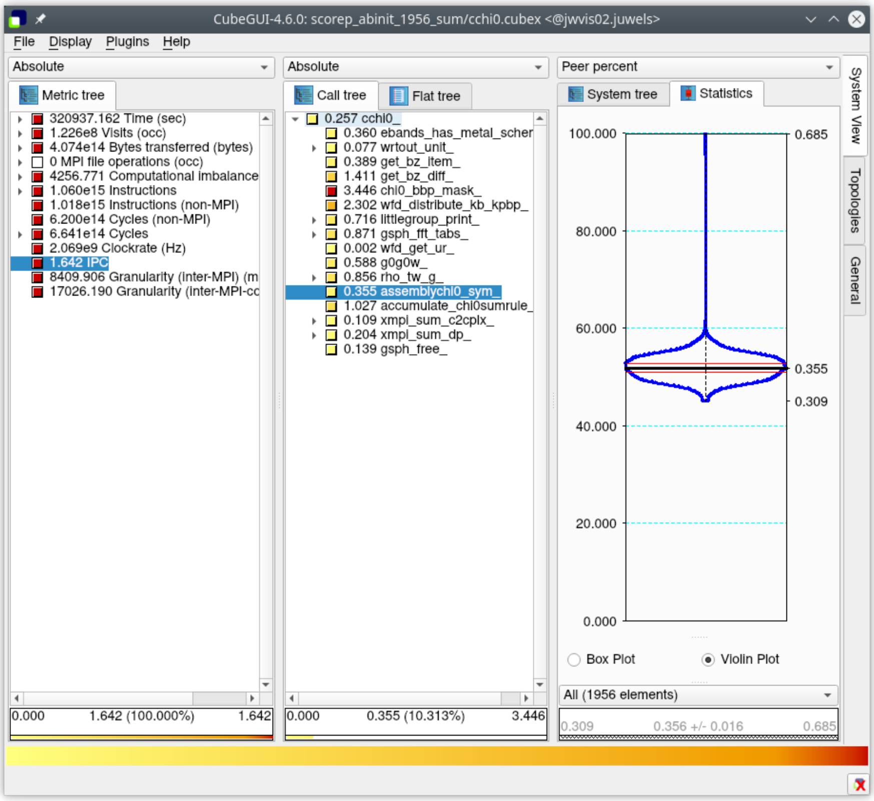
Callpath	Issue

Figure 13: CUBE Advisor panels showing hybrid efficiency variants for a selected focus of analysis, providing complementary perspectives of the same execution measurement. (left) Additive variant highlights low Thread Efficiency originating from low Amdahl Efficiency for code outside OpenMP parallel regions executed serially by master threads of each process, with MPI Process Efficiency considered to be good. OpenMP Load Balance Efficiency and Other OpenMP Inefficiency not yet included. (right) Multiplicative variant highlights very poor MPI Load Balance, while OpenMP Parallel Efficiency is perfect. MPI Serialization Efficiency and Transfer Efficiency not yet included.

architecures, the more relevant task within the scope of POP2 has been the porting of Extrae to the CTE-ARM cluster at BSC.

To support the POP studies, we need to update and complement the support to the different programming models that are instrumented. In that sense we extended the MPI calls intercepted by Extrae to include MPI_Mprobe, MPI_Improbe, MPI_Mrecv, and MPI_Imrecv that are used in the implementation of MPI4PY Python module. With respect to OpenMP, we added the instrumentation of manual locks, revised the support for the task loop construct, and fixed the state of the threads after a shutdown.

As bigger develoments for OpenMP support, we modified Extrae to represent OpenMP nested parallelism delimiting the inner level as a "black box" that we would try to characterize in a future refinement and we extended the Extrae *Burst Mode* to support OpenMP and MPI+OpenMP applications allowing to target such kind of applications at larger scales collecting summarized information. This last development is currently limited to the GNU environment, but it will be extended to Intel and OMPT in the near future. Figure 15 briefly compares the characterization provided for the parallel functions in detailed mode versus burst mode with Lulesh MPI+OpenMP execution that obtains a reduction on the data colected close to two orders of magnitude. We have also worked on updating the instrumentation of OpenMP runtime using OMPT (OpenMP Tools interface) that enables better and more portable measurement of OpenMP codes but as we have faced


CubeGUI-4.6.0: scorep_abinit_1956_sum/cchi0.cubex <@jwvis02.juwels>

File Display Plugins Help

Absolute — Metric tree
- 320937.162 Time (sec)
- 1.226e8 Visits (occ)
- 4.074e14 Bytes transferred (bytes)
- 0 MPI file operations (occ)
- 4256.771 Computational imbalance
- 1.060e15 Instructions
- 1.018e15 Instructions (non-MPI)
- 6.200e14 Cycles (non-MPI)
- 6.641e14 Cycles
- 2.069e9 Clockrate (Hz)
- 1.642 IPC
- 8409.906 Granularity (inter-MPI) (m
- 17026.190 Granularity (inter-MPI-cc

0.000 1.642 (100.000%) 1.642

Absolute — Call tree / Flat tree
- 0.257 cchi0_
  - 0.360 ebands_has_metal_scher
  - 0.077 wrtout_unit_
  - 0.389 get_bz_item_
  - 1.411 get_bz_diff_
  - 3.446 chi0_bbp_mask_
  - 2.302 wfd_distribute_kb_kpbp_
  - 0.716 littlegroup_print_
  - 0.871 gsph_fft_tabs_
  - 0.002 wfd_get_ur_
  - 0.588 g0g0w_
  - 0.856 rho_tw_g_
  - 0.355 assemblychi0_sym_
  - 1.027 accumulate_chi0sumrule_
  - 0.109 xmpi_sum_c2cplx_
  - 0.204 xmpi_sum_dp_
  - 0.139 gsph_free_

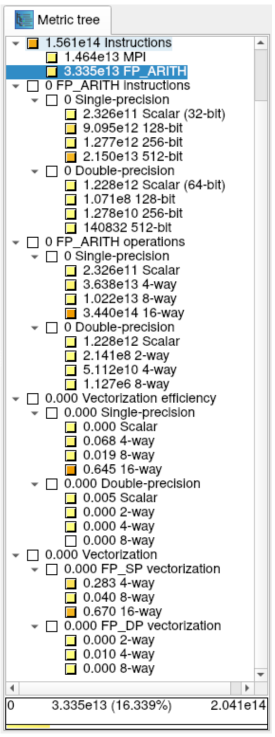
0.000 0.355 (10.313%) 3.446

Peer percent — System tree / Statistics

Box Plot [✓ Violin Plot]

All (1956 elements)

0.309 0.356 +/- 0.016 0.685


Metric tree
- 1.561e14 Instructions
  - 1.464e13 MPI
  - 3.335e13 FP_ARITH
- 0 FP_ARITH instructions
  - 0 Single-precision
    - 2.326e11 Scalar (32-bit)
    - 9.095e12 128-bit
    - 1.277e12 256-bit
    - 2.150e13 512-bit
  - 0 Double-precision
    - 1.228e12 Scalar (64-bit)
    - 1.071e8 128-bit
    - 1.278e10 256-bit
    - 140832 512-bit
- 0 FP_ARITH operations
  - 0 Single-precision
    - 2.326e11 Scalar
    - 3.638e13 4-way
    - 1.022e13 8-way
    - 3.440e14 16-way
  - 0 Double-precision
    - 1.228e12 Scalar
    - 2.141e8 2-way
    - 5.112e10 4-way
    - 1.127e6 8-way
- 0.000 Vectorization efficiency
  - 0.000 Single-precision
    - 0.000 Scalar
    - 0.068 4-way
    - 0.019 8-way
    - 0.645 16-way
  - 0.000 Double-precision
    - 0.005 Scalar
    - 0.000 2-way
    - 0.000 4-way
    - 0.000 8-way
- 0.000 Vectorization
  - 0.000 FP_SP vectorization
    - 0.283 4-way
    - 0.040 8-way
    - 0.670 16-way
  - 0.000 FP_DP vectorization
    - 0.000 2-way
    - 0.010 4-way
    - 0.000 8-way

0 3.335e13 (16.339%) 2.041e14

Figure 14: CUBE GUI showing additional derived metrics including Clockrate, IPC and two variants of Computation Granularity (distinguishing MPI collectives and all MPI operations), and (from a different measurement of the same code) hierarchies of Vectorization Efficiency and Vectorization Ratio metrics derived from counts of single- and double-precision floating-point arithmetic instructions on Intel 'Skylake' processors. IPC instruction intensity is poor for the selected callpath, though one process has much better IPC than the others. Double-precision floating-point is hardly used, whereas single-precision vectorization makes good use of 16-way AVX512 instructions. Routines with high vectorization are found to have very low IPC, whereas those with no floating-point operations have close to perfect IPC of 4 for this processor.

some limitations of the current interface, it is not yet included in the public version.

With respect to newer programming models and environments, we extended the Extrae library to support Python+CUDA applications and improved the CUDA support updating to the latest versions and correcting the information collected. We also validated we can instrument with Extrae Julia applications using the MPI.jl package.

General improvements include a new naming of PAPI native events that assigns the same identifier for the same counter over different executions (unlike PAPI's default identifiers), the extension of Extrae to include new types of counters (uncore counters and infiniband counters) and the instrumentation of dynamic memory intercepting the calloc function and adding the total amount of dynamic memory in-use across all program's allocation and free operations.

3.3.2 Paraver

Paraver is the visualization and analysis tool. The main goal of the Paraver improvements implemented in POP2 is to facilitate the usage for newcomers. In this sense, we have redesigned the hints

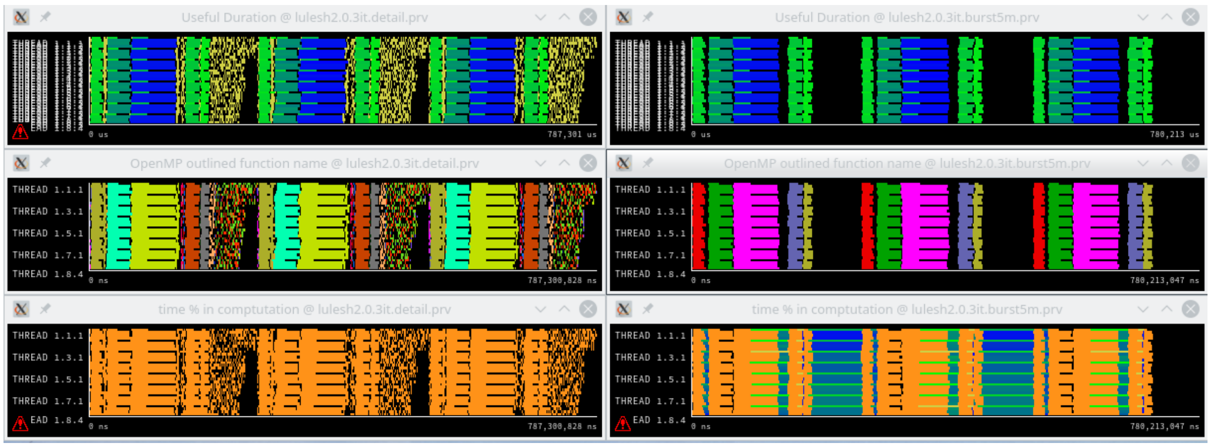

Useful Duration @ lulesh2.0.3it.detail.prv
Useful Duration @ lulesh2.0.3it.burst5m.prv
OpenMP outlined function name @ lulesh2.0.3it.detail.prv
OpenMP outlined function name @ lulesh2.0.3it.burst5m.prv
time % in comptutation @ lulesh2.0.3it.detail.prv
time % in comptutation @ lulesh2.0.3it.burst5m.prv

Figure 15: Comparing the traditional *Detailed mode* with the new OpenMP *Burst Mode* . Left images correspond to the detailed mode and the same metrics are shown on the right images in burst mode. Top figures compare the computing phases where we can see that the burst mode discards all the small computations colored in yellow in the left image. Middle view shows the information captured regarding parallel loops, also showing how the burst mode does not record the smaller loops in the trace. Bottom views express the time computing on the different phases, where the burst mode summarizes the activity for the discarded fine-granularity phases, identifying the imbalance as a gradient color from light-green (small) to dark-blue (large), with orange indicating phases that were instrumented in detail.

menu to add more configurations and also new workspaces, facilitating the access to the most typical timelines and tables. An example of the hints and workspaces option is shown in Figure 16. We have also improved the integration to run the clustering tool from the visualizer allowing us to specify the name of the output trace as well as to be able to activate the sampling. To facilitate the installation, we implemented a new functionality that allows to download and install the Paraver tutorials clicking a button of the tool.

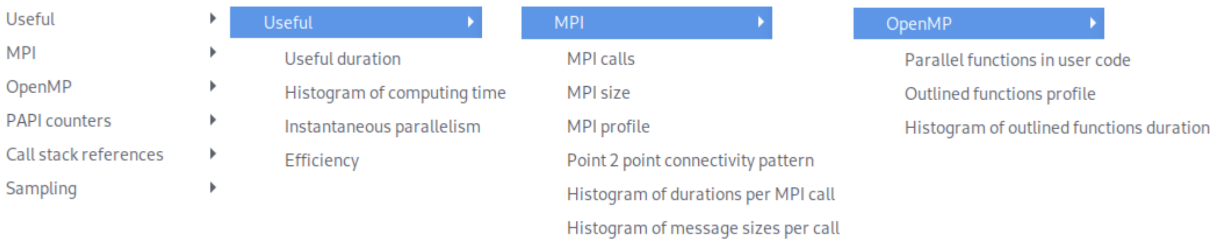

Useful
MPI
OpenMP
PAPI counters
Call stack references
Sampling
Useful
Useful duration
Histogram of computing time
Instantaneous parallelism
Efficiency
MPI
MPI calls
MPI size
MPI profile
Point 2 point connectivity pattern
Histogram of durations per MPI call
Histogram of message sizes per call
OpenMP
Parallel functions in user code
Outlined functions profile
Histogram of outlined functions duration

Figure 16: Example of the Paraver Hints menu for an MPI+OpenMP tracefile including the list of basic views for the workspaces that characterize the computations (Useful), the MPI calls and the OpenMP activity.

The POP studies also allow us to identify the need for new functionalities. In this aspect, we have added new semantic functions at composing level (accumulate, log N, and exp) as well as a new derived function (controlled: enumerate). To facilitate the programming of the tool we have implemented search with regular expressions for event selection, and the option to link properties when saving a configuration file in basic mode. With respect to the data visualization we have implemented a new feature that enables the user to configure the timeline colors, and a new functionality to reorder the columns of the tables.

Finally with respect to the tool execution, Paraver was parallelised with OpenMP on Linux and

Mac environments and it has been extended to run also in parallel in Windows environment.

3.3.3 Dimemas

Dimemas message passing simulator is mainly used for the efficiency metrics, despite some partners also use this tool to analyze the application network requirements. Dimemas has been extended to support the new MPI calls instrumented by Extrae (MPI_Mprobe, MPI_Improbe, MPI_Mrecv, and MPI_Imrecv). The POP2 traces also allowed us to detect and fix some bugs in the translation of MPI+OpenMP traces to Dimemas format and with race conditions.

The main developments in Dimemas have been concentrated in the support of accelerators and OpenMP. With respect to accelerators, Dimemas can simulate platforms with accelerators but it was limited to one stream per device and it has been extended to enable having multiple streams per device as it is the most frequent scenario in the applications. With respect to OpenMP, despite Dimemas does not model the execution of OpenMP, we have implemented the support to synchronize the threads in a MPI+OpenMP simulation, so that the parallel regions keep the threads sincronization.

3.3.4 Basic Analysis

The Basic Analysis module allows us to compute the efficiency metrics, and it was developed in 2013 (before the first POP project) to analyze pure MPI traces. It is written in Python, and it executes Paramedir (a non-graphical version of Paraver) and Dimemas, postprocessing their outputs.

Within POP2, we have redesigned the Basic Analysis module to increase the robustness of the tool and extend the programming models and configurations supported as we identified some partners were using the tool, for instance for MPI+OpenMP traces that were not adequately supported. We have added sanity checks to evaluate and report the weight of I/O and trace flushing as they impact on the quality of the efficiency metrics and we detected that some users of the tool do blind analysis without checking the content of the traces.

Figure 17 illustrates with the example of the NAS BT-IO the new information that characterizes the I/O activity and its efficiency that expresses its weight with respect to the time outside the parallel runtimes. When the percentage of time in I/O overpases a given boundary, a warning message is printed because the I/O impacts on the efficiency metric and it may improve or degrade for instance the load balance compared with the same execution without I/O.

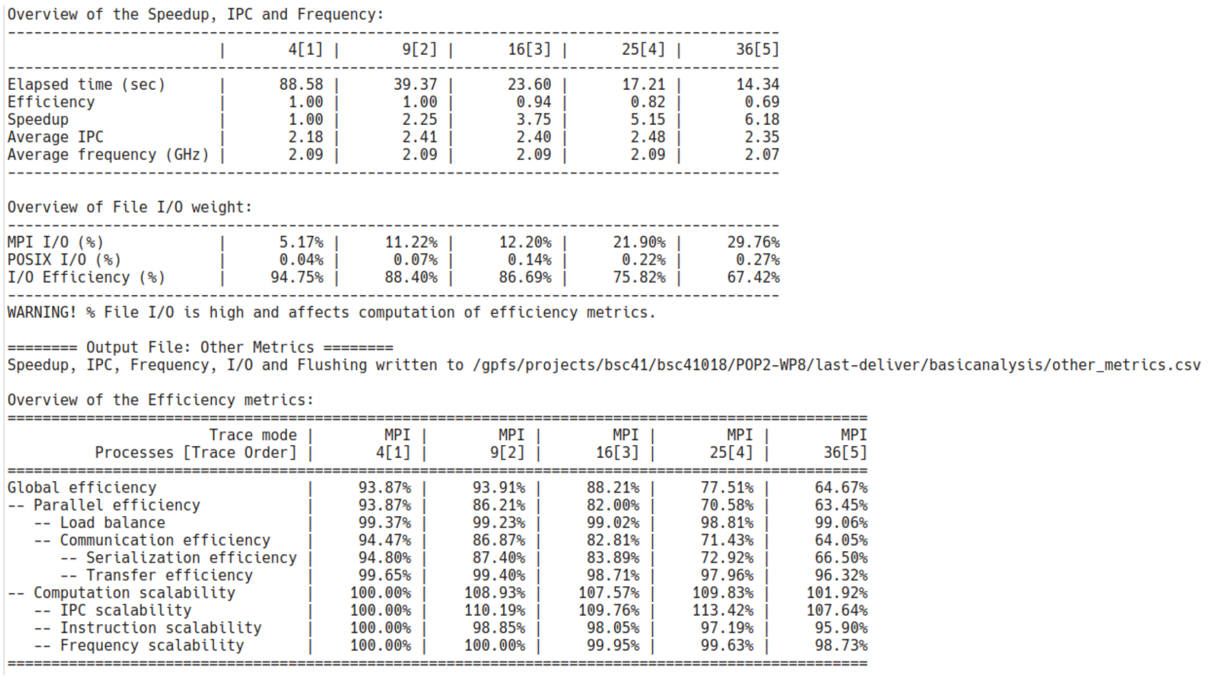
The tool detects the type of traces with respect to both the programming models (MPI, OpenMP, CUDA, ...), the level of information (Detailed or Burst mode) and the scaling approach (weak or strong scaling), applying the right configurations and determining if the scenario can be simulated with Dimemas or not. We have implemented the BSC proposal for the hybrid efficiencies that work for any aplication of MPI+X and also the BSC file I/O efficiency (at MPI-IO and POSIX-IO levels).

Finally we have also extended the outputs generated by the tool (plotting and tables).

3.4 PyPOP

PyPOP is a new high-level tool for generating POP metrics and reports. NAG is developing that in the second iteration of the POP project. Written in the Python programming language, it is designed to provide a unified interface for the calculation of the POP metrics using data from different performance analysis tools, generating high-quality scaling plot and metric table figures, and efficiently generating POP analysis reports incorporating these results.

PyPOP has been developed to extend the existing POP tool offering, adding new features to improve the power and accessibility of the POP tools and metrics. The primary reason for developing PyPOP is to streamline and automate many of the tasks involved in performing a POP assessment,



```
Overview of the Speedup, IPC and Frequency:
------------------------------------------------------------------------------------------
                        |        4[1] |        9[2] |       16[3] |       25[4] |       36[5]
------------------------------------------------------------------------------------------
Elapsed time (sec)      |       88.58 |       39.37 |       23.60 |       17.21 |       14.34
Efficiency              |        1.00 |        1.00 |        0.94 |        0.82 |        0.69
Speedup                 |        1.00 |        2.25 |        3.75 |        5.15 |        6.18
Average IPC             |        2.18 |        2.41 |        2.40 |        2.48 |        2.35
Average frequency (GHz) |        2.09 |        2.09 |        2.09 |        2.09 |        2.07
------------------------------------------------------------------------------------------

Overview of File I/O weight:
------------------------------------------------------------------------------------------
MPI I/O (%)             |       5.17% |      11.22% |      12.20% |      21.90% |      29.76%
POSIX I/O (%)           |       0.04% |       0.07% |       0.14% |       0.22% |       0.27%
I/O Efficiency (%)      |      94.75% |      88.40% |      86.69% |      75.82% |      67.42%
------------------------------------------------------------------------------------------
WARNING! % File I/O is high and affects computation of efficiency metrics.

======== Output File: Other Metrics ========
Speedup, IPC, Frequency, I/O and Flushing written to /gpfs/projects/bsc41/bsc41018/POP2-WP8/last-deliver/basicanalysis/other_metrics.csv

Overview of the Efficiency metrics:
====================================================================================================
                       Trace mode |         MPI |         MPI |         MPI |         MPI |         MPI
              Processes [Trace Order] |        4[1] |        9[2] |       16[3] |       25[4] |       36[5]
====================================================================================================
Global efficiency                 |      93.87% |      93.91% |      88.21% |      77.51% |      64.67%
-- Parallel efficiency            |      93.87% |      86.21% |      82.00% |      70.58% |      63.45%
   -- Load balance                |      99.37% |      99.23% |      99.02% |      98.81% |      99.06%
   -- Communication efficiency    |      94.47% |      86.87% |      82.81% |      71.43% |      64.05%
      -- Serialization efficiency |      94.80% |      87.40% |      83.89% |      72.92% |      66.50%
      -- Transfer efficiency      |      99.65% |      99.40% |      98.71% |      97.96% |      96.32%
-- Computation scalability        |     100.00% |     108.93% |     107.57% |     109.83% |     101.92%
   -- IPC scalability             |     100.00% |     110.19% |     109.76% |     113.42% |     107.64%
   -- Instruction scalability     |     100.00% |      98.85% |      98.05% |      97.19% |      95.90%
   -- Frequency scalability       |     100.00% |     100.00% |      99.95% |      99.63% |      98.73%
====================================================================================================
```

Figure 17: Example of the Basic Analysis output text reporting the weight of the I/O and the I/O effciencies.The tool warns the user when the percentage of I/O time is relevant as it is impacting on the efficiency metrics.

such as calculating metrics and formatting reports. This will allow us to reduce the time and cost involved in performing a POP assessment, which is one of the key goals of the POP2 project. The design of PyPOP is also intended to allow less-technical users to generate and explore the POP metrics by providing a simple interface that can accept multiple trace formats. In addition to streamlining and improving accessibility to existing POP tool functionality, PyPOP adds analysis functionality not currently available in any other POP tool, including per-region analysis of OpenMP performance, and calculation of the various proposed and experimental metric types currently under development.

PyPOP is open source under the BSD license and is available via GitHub[5].

3.4.1 Design Philosophy

PyPOP is implemented as a Python package that uses the Jupyter Notebooks as the primary method of user interaction. These notebooks may be accessed via any modern web browser or using a dedicated notebook viewer. Jupyter Notebooks are an example of what is known as “literate programming” where blocks of source code are interspersed with detailed text describing what is being calculated and discussing the results. This is particularly advantageous for complex technical work such as a POP assessment as it allows a single notebook to be a complete document describing the work that was done, including the inputs used, calculations made, and the analysis and discussion created by the POP analyst. This notebook can then be compiled into a final report document using included templates. As a result, the reproducibility of POP assessments produced with PyPOP is vastly improved as the report document also contains the code and parameters used in the analysis.

The backend code of PyPOP is designed to be easily extensible to support new tracing tool formats and new metric types. To achieve this PyPOP defines a Trace API and a Metric API, which

[5]https://github.com/numericalalgorithmsgroup/pypop

specifies the functionality required of any code that is written to support a new tool or metric. New classes can be written by inheriting from the provided Trace and Metric classes within PyPOP. These are then registered with PyPOP as plugins providing the relevant functionality. At the time of writing, two trace format plugins are supporting Extrae trace data and manual input of raw data into a spreadsheet-like interface. There are also a variety of Metric plugins supporting the core POP Metrics as well as the proposed hybrid MPI+OpenMP metrics and other experimental metrics for the OpenMP and PThreads paradigms.

PyPOP is based on widely used software in the scientific Python ecosystem, including the Pandas and Numpy numerical and statistical libraries, the IPython/Jupyter stack for the notebook, GUI and PDF conversion elements, and the Bokeh library to produce high-quality interactive tables and scaling plots. This reduces the barrier to entry for users who wish to perform advanced analyses or extend the functionality of PyPOP as it uses components which are likely to be familiar to most scientific users of Python.

3.4.2 Interface Design

Two key requirements drive PyPOP's interface design. First, the need for a standalone Jupyter notebook to function as both an interactive analysis environment and a static final report, and for non-Python programmers to be able to use PyPOP to generate reports without needing to write Python code. These two requirements are achieved by making use of graphical user interface (GUI) elements within the Jupyter notebook, to provide user interactivity during the analysis.

To allow non-Python programmers to create reports, an "Analysis Wizard" interface is provided. This allows the user to graphically select the trace files they wish to analyze and instruct PyPOP to calculate the required POP metrics and generate the metrics table and scaling plot figures for the user to inspect. If the user is happy with the generated metrics and figures, PyPOP can then be instructed to populate a further notebook containing the relevant code to generate these figures along with text cells containing the layout and structure of a POP report. By adding the relevant description and discussion in these text cells, the user can create a full POP report which can be converted to PDF format using the provided report generation utility. This workflow allows the user to generate a complete, reproducible notebook based report without needing to write any Python code.

The figures are generated by the Bokeh package, which outputs them in the form of HTML, which can be rendered by the web browser. As well as being portable across many platforms (Windows, Mac, Linux, mobile, etc.), this has the advantage that the generated plots may have an interactive element. This is supported by Bokeh and leveraged in PyPOP to provide additional information through tooltips. For example, when the user places the mouse cursor over a cell in the metrics table, a tooltip will appear, providing a description of the particular metric and what it describes, while a scaling plot tooltip would provide information about specific data points such as the absolute runtime and other key values. When converting to the PDF report, these figures are automatically converted to static images before the PDF is compiled.

Users who are familiar with the Python language may wish to start with the Analysis Wizard but are then free to use the full available functionality of PyPOP as they wish to perform more detailed custom analyses. Alternatively, example notebooks are provided which demonstrate advanced usage of PyPOP for a variety of scenarios, e.g., analysis of individual OpenMP regions, which users can adapt to their needs. Users can also access the raw data extracted from the trace files, as well as the calculated metric data and additional metadata describing the trace file and run conditions to perform completely custom analysis. The results of these analyses can then be rendered into the Notebook and PDF report using the PyPOP plotting functionality. For example, the metric table shown in Figure 10 was generated by PyPOP.

3.4.3 Current Development Status and Further Goals

Currently, the following features have been implemented in PyPOP:

- Jupyter notebook based literate programming interface
- High-quality interactive tables and plots
- Analysis "Wizard" for non-Python programmers
- PDF report creation from notebooks
- Support for POP MPI and proposed hybrid metrics
- Support for Extrae traces and manual statistics input

Further development goals include:

- Extending support to additional trace file formats
- Supporting new POP metrics as they are defined
- GUI improvements based on user feedback
- Improve testing and tool robustness
- Improved documentation
- Generation of the report as slides

3.5 MAQAO

MAQAO (Modular Assembly Quality Analyzer and Optimizer) is a performance analysis and optimization framework operating at binary level with a focus on core performance. Its main goal is to guide application developers through the optimization process with the generated reports and hints.

MAQAO mixes both dynamic and static analyses based on its ability to reconstruct high-level structures such as functions and loops from an application binary. Since MAQAO operates at the binary level, it is agnostic of the language used in the source code and does not require recompiling the application to perform analyses. MAQAO has also been designed to support multiple architectures concurrently. Currently, the Intel64 and Xeon Phi architectures are implemented. A basic support is also now provided for ARM.

The main modules of MAQAO are *LProf*, a sampling-based lightweight profiler offering results at both function and loop levels, *CQA*, a static analyzer assessing the quality of the code generated by the compiler. Both tools are integrated into the *ONE View* tool, which aggregates their results to provide users with one consolidated view.

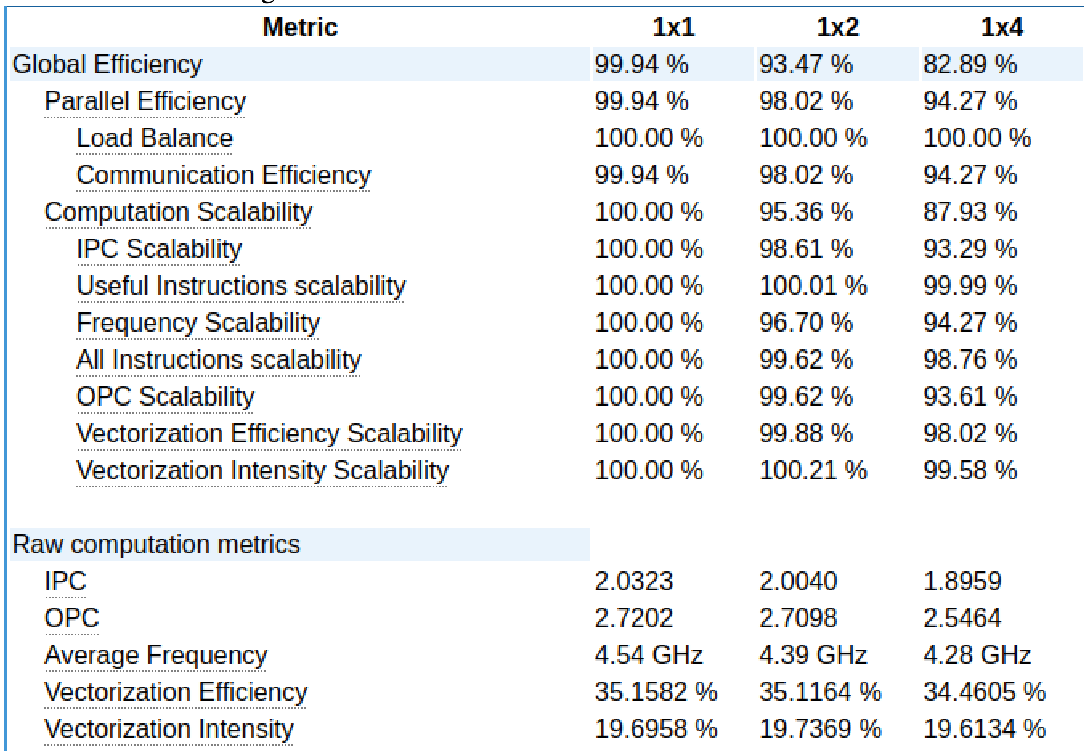
Until now, MAQAO provided its own way of displaying parallel and computation metrics. We have been working on integrating the POP metrics model into MAQAO. The result is a new view at the application level called *POP metrics* that provides the same layout common to all the tools. The provided metrics are:

- Parallel efficiency: we provide sub-metrics load balance and communication efficiency. Since MAQAO does not perform tracing, we do not provide the detailed sub-metrics of Communication efficiency (serialization and transfer).

- Computation scalability: IPC scalability, Useful instructions scalability, Average Frequency scalability but also Vectorization efficiency scalability and OPC scalability. Although we present more metrics in this computation scalability section, the computed metric is only based on IPC / Useful instructions and Frequency in order to remain consistent with the other tools (to be able to compare figures for example).

The following figure 18 shows the latest version (05/2022) if the *POP metrics* in the *ONE View* tool interface. We can see that we also provide raw computation metrics to keep an eye on the new OPC and Vectorization metrics. This will be important to build the computation efficiency part of the model.

Figure 18: POP metrics view in the ONE View tool



Metric	1x1	1x2	1x4
Global Efficiency	99.94 %	93.47 %	82.89 %
Parallel Efficiency	99.94 %	98.02 %	94.27 %
Load Balance	100.00 %	100.00 %	100.00 %
Communication Efficiency	99.94 %	98.02 %	94.27 %
Computation Scalability	100.00 %	95.36 %	87.93 %
IPC Scalability	100.00 %	98.61 %	93.29 %
Useful Instructions scalability	100.00 %	100.01 %	99.99 %
Frequency Scalability	100.00 %	96.70 %	94.27 %
All Instructions scalability	100.00 %	99.62 %	98.76 %
OPC Scalability	100.00 %	99.62 %	93.61 %
Vectorization Efficiency Scalability	100.00 %	99.88 %	98.02 %
Vectorization Intensity Scalability	100.00 %	100.21 %	99.58 %
Raw computation metrics			
IPC	2.0323	2.0040	1.8959
OPC	2.7202	2.7098	2.5464
Average Frequency	4.54 GHz	4.39 GHz	4.28 GHz
Vectorization Efficiency	35.1582 %	35.1164 %	34.4605 %
Vectorization Intensity	19.6958 %	19.7369 %	19.6134 %

Our primary contribution was to work on a global vectorization metric described in 2.3 and a more accurate metric at the instruction level (i.e. OPC as replacement for IPC).

3.6 Critical-path prototype

We developed a prototype tool that implements an on-the-fly critical path analysis, as described in Section 2.1.4, while observing the execution of a program. This tool enables the calculation and evaluation of our critical path-based hybrid metrics with applications coming from POP2 WP5 studies. The tool can also be used as a reference implementation for the critical path-based hybrid metrics that can be used by the partners to extend the existing tools with support for these hybrid metrics. In the following we want to briefly describe the implementation of this prototype tool.

In order to compute the critical path-based hybrid metrics we are only interested in the sum of weights along the critical path. Therefore, we can implement the critical path metric following the concept of a Lamport clock. Each execution unit has a local value of the metric. The weight is added to the local value for each step in the execution graph. Concrete synchronization in the execution graph can have different characteristics. The simplest form is point-to-point synchronization, like a pair of send and receive calls in MPI. Another form is barrier synchronization, where each execution unit needs to arrive before all can continue. Finally, OpenMP has several forms of channeled signal-wait synchronization, where the signaling execution unit and the waiting execution unit do not know all synchronization partners. An example is the synchronization of task execution with any of the task synchronization constructs like `barrier`, `taskwait`, `taskgroup`, or dependencies with other tasks.

Critical path in OpenMP The Archer (Section 3.1.1) runtime solves a similar challenge for OpenMP-aware data race detection, translating OpenMP synchronization into vector clock semantics, that the data race detection tool ThreadSanitizer can understand. Following the concept of synchronization clocks, initially introduced by FastTrack[6], all synchronization with signal semantic updates the thread-local clock towards the synchronization clock, and all synchronization with wait semantic updates the thread-local clock from the synchronization clock. We adapt and extend the Archer runtime to implement the Lamport clock updates for the OpenMP part of our analysis. In fact, our clock does not consist of a single clock value but keeps a separate clock value for CUE, COM, and COO—and actually a copy of each for the process local and global critical path. In addition to tracking the synchronization, we integrate the time measurement for time spent in the OpenMP runtime library into these OMPT callbacks.

Critical path in MPI For MPI, we implement the Lamport clock updates using communication piggybacking with additional communication calls on shadow communicators to avoid interference with application communication. For collective communication with barrier semantics in the application (e.g., `barrier`, `allreduce`, or `alltoall`) we use a maximum all-reduction on all participating threads' clocks. Similarly, we use a broadcast of the root's clocks for application calls like `bcast` or `scatter` and a reduction on all participating threads' clocks towards root's clock for application calls like `reduce` or `gather`.

[6]C. Flanagan and S. N. Freund, FastTrack: efficient and precise dynamic race detection, Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, 2009

4 Conclusions

In this document, we presented work performed by the POP methodology and tools working group in the context of WP8. We introduced various proposals for the extension of the POP methodology. For the analysis of hybrid MPI + OpenMP applications, we have proposed an additive, a multiplicative and a critical path-based set of metrics.

Based on experience from applying the metrics to concrete POP audits, we discussed the pros and cons of these metric sets. The multiplicative set of metrics focuses more on general concepts of parallel programming models and naturally applies to any hybrid application using MPI + an arbitrary model X. The additive set of metrics requires less abstract knowledge but is harder to adapt to new programming models. The critical path-based metrics also require similar adaption to new programming models but offer a breakdown of the basic POP model factors *load balance*, *serialization* and *transfer efficiency* separately for each programming model.

Moreover, we showed that the hybrid metric sets help the application developers to better understand and solve performance issues in practice. Which model to choose is a choice of personal preference similar to choosing the most suitable performance analysis tool. For specific use-cases we provided some recommendations which model to choose. In the future we will not add more hybrid models because the comparability of the metrics is one of the key strengths of the whole POP methodology, which gets harder the more different models are used.

Furthermore, we presented concrete proposals for metrics describing the influence of file I/O and vectorization to the efficiency of execution. We also described how these metrics integrate into existing POP metrics. The vectorization metrics were integrated into the computational scaling and have been proven to provide insights into optimization potential for real applications in practice. The I/O metrics were integrated into the existing hierarchy as complementary metrics. Due to the lack of use cases that we could find in the work package 5, we could not conduct thorough investigations on the impact of I/O metric to improve performance.

We also presented a tool called PyPOP to support the creation of POP reports with semi-automatically collecting the POP metrics from collected traces. This tool can help to make producing the report more productive. We also developed a prototype tool that allows very lightweight calculation of the redefined multiplicative hybrid POP metrics in an online fashion during runtime of an application. In the future our existing analysis tools may adapt this approach implemented in the prototype tool to also support the online calculation of POP metrics.

The effort spent in this work package 8 is crucial in order to enable the work done by the POP services in work packages 5 and 6. The POP methodology and the analysis tools needs to be kept up to date for all the different challenges faced by the POP services. Our proposed extensions to the POP methodology have also been implemented in our tools. At the same time programming models are also continuously evolving. For example, with the release of MPI 4.0 in the last year new features were added to the standard. 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.