How to create a POP performance audit
Version 1.1

Notices: *The research leading to these results has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No n°676553.*

Contents

1 Introduction

In many cases one goal of parallel programming is to generate certain results faster. Therefore measuring the performance of such an application is essential to check if this goal was reached. Many tools exist which help programmers to get detailed information of their codes performance. However, these tools often allow to measure a lot of different information and visualize it in different ways. Extracting useful insights out of this data is a very challenging task. The goal of a POP performance audit is to extract certain information out of this data and present it in a well-structured way to the developer. Having the same structure in all audits helps to compare different applications or different versions of one application. Therefore, POP performance audits fit perfectly into a software development cycle to check regularly if key performance aspects have changed. This document will explain the general structure of a POP performance audit as a best-practice guide. It helps you to apply our workflow to your parallel applications. Although we mainly focus on the tools developed by our project partners (Score-P, Scalasca, Cube, Extrae, Paraver, Dimemas) the workflow is independent of the actual tool used and can be done with any tool to a certain extent. This document will not give instructions how to use a certain tool, but this information can be found on our project website[1].

2 General structure

As mentioned earlier a goal of POP performance audits is to be comparable across different versions of the same application and also between applications. Therefore you should follow a structured approach to do a general health check of the application performance as we do it in our POP performance audits. The general structure of a POP performance audit consist of 9 sections, each dealing with different information. The report starts with a general overview and will dive into more specific topics. Next, you will find a brief overview of the sections in a POP performance audit. Detailed explanations on how to generate and analyze the data needed to fill these sections is provided in the rest of this document. The sections are:

1. **Background:** In this section you should write down all information relevant to reproduce your program run later on. This information might contain the version of your application, the dataset, the compiler, the library versions, the machine hardware and so on. If you investigate the report again some time later this information is essential to understand the analysis or compare it to a different report, e.g. after code optimization.

2. **Application Structure:** After generating a first profile or trace of your application run you should describe the application structure as observed by the tool. If you can identify different phases in your application run or detect several time consuming hotspots you should note those and write down how sever these findings were.

3. **(if appropriate) Region of Interest:** In many cases it is not reasonable to evaluate the whole application run all in once. If for example different phases have been identified in the application structure, it is better if you investigate the phases one after the other to get cleaner results per phase. It can also be useful if you only investigate the behavior of a subset of processes if the structure has shown abnormal behavior for some processes.

4. **Scalability Information:** Scalability is the first performance property we recommend you to investigate. Scalability Information is obtained by comparing different program

[1] www.pop-coe.eu

runs with an increasing number of threads and processes. If the performance increase is less than linear for a linear increase of hardware resources this is a first indication of a non-optimal parallel efficiency of the code.

5. **Application Efficiency:** The main part of this section is a table showing different efficiency metrics. Basically the performance of the application is summarized in a general efficiency number which gives information about how many percent of the overall time is spend for useful computation. Furthermore typical issues like communication overhead, load imbalance and synchronization are described. These metrics give hints if a performance problem is present and which kind of problem it is. Therefore, we recommend that you generate this handful of metrics for your ROIs in any report.

6. **Load Balance:** In this section more information is given about load balance. Besides the fact that there is load imbalance or not, which is already given as a metric in the previous section, it is interesting to investigate how the work is distributed across processes and threads and if it is uniform or changes during runtime.

7. **Serial Performance:** In this section the serial performance of the application is further investigated. Serial performance means regions, which do not have communication constructs inside. Here metrics like IPC can be used to give a hint if the hardware is used efficiently. If there is an indication that this is not the case, further test with hardware counts to measure the core or cache utilization can be shown here.

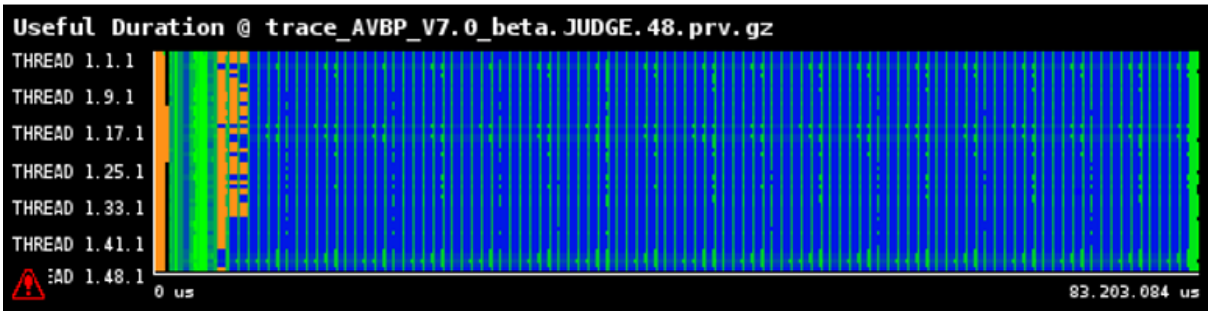
8. **Communications:** Here, you should provide information on communication overhead within the application. It should be analyzed which kind of communication causes the overhead and how the communication pattern looks like.

9. **Summary and Recommendations:** Finally, a summary of the key findings within the report is given. You should provide recommendations which performance issue should be investigated further or give an advice how to eliminate a certain problem. This part is useful to communicate next steps to others and also to remember later was essentially was the result of your performance study.

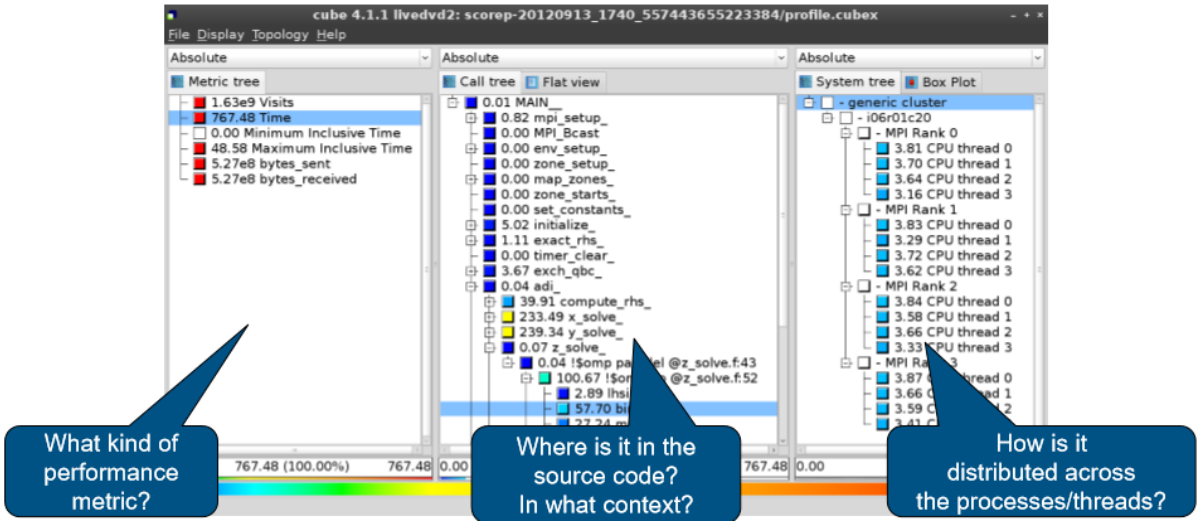
3 Application Structure and Region(s) of Interest

In the beginning of a performance audit we recommend to give an overview of the application. The goal of this section in a report is to present the overall context of the later in-depth analysis. Depending on the used performance tool, different information can be used to give an overview of the application.

One common way in performance tools to present data in a time related fashion is a timeline, as shown in figure 1(a). The application time is shown on the x-axis and processes and threads on the y-axis. The color value expresses in which region a thread or process is at which time. This allows to present a lot of details in a single view and it can often be used to get an overview of the application. In the presented example it can be observed, that some starting phase is executed at the beginning. Then the application performs quite similar iterations before at the end a shutdown phase can be seen. Such a timeline allows you to describe the chronological sequence of phases on a high level which occur during execution of the application.



(a) Paraver Timeline



(b) Cube Profile Explorer

Figure 1: Screenshots of analysis data in different tools.

A different view is presented in figure 1(b). Here a function profile is shown with the Cube GUI. This tool focuses on functions executed and the call path of the functions. Such a tool gives a good overview of the executed functions and how much time was spend in which functions in the source code. Furthermore, it gives an overview of processes and threads executing those functions. Such a view easily allows to identify compute intensive regions in the code, so called hot-spots.

After presenting an overview in one or the other way, a section about the region(s) of interest (ROI) is recommended. A ROI is the part where the rest of the audit will focus on. The reason for such an approach is to focus on the most relevant parts of the application, but reduce the amount of data to analyze as much as possible. ROIs might be one or more hot-spots consuming most of the execution time in a few functions or it might be a few iterations during execution. Also a spacial limitation, like focusing only on a few processes which show abnormal behavior in the overview analysis is sometimes useful.

The audit should present which ROIs were identified for further investigation and place them in the context of the general application overview. For example describing how much overall execution time is covered by the ROIs or telling the user if the chosen iterations are representative for the overall execution or are extreme cases of some bad iterations.

4 Scalability Analysis

After the region(s) of interest has/have been identified, the overall scalability of the application should be investigated if appropriate. The scalability describes how the runtime of the application changes if the number of cores used is changed. To do such an analysis the application and dataset must allow to run with a differing number of processes/threads. The scalability analysis gives a first hint on the parallel overhead in the application. The analysis should be done for the whole application and possibly for different regions of interest. To obtain the total speedup, a simple external timer can be used, but of course to analyze different regions of interest, you need to generate several profiles or traces with a different number of processes or threads.

The scalability can be shown in different ways, for example as runtime, speedup or efficiency.

Runtime The easiest way to present the runtime in a performance audit is by using a table. Fictitious example data for measurements with 128, 256 and 512 processes and for an application with three regions of interest, ROI 1, ROI 2 and ROI 3 is shown in Table 1.

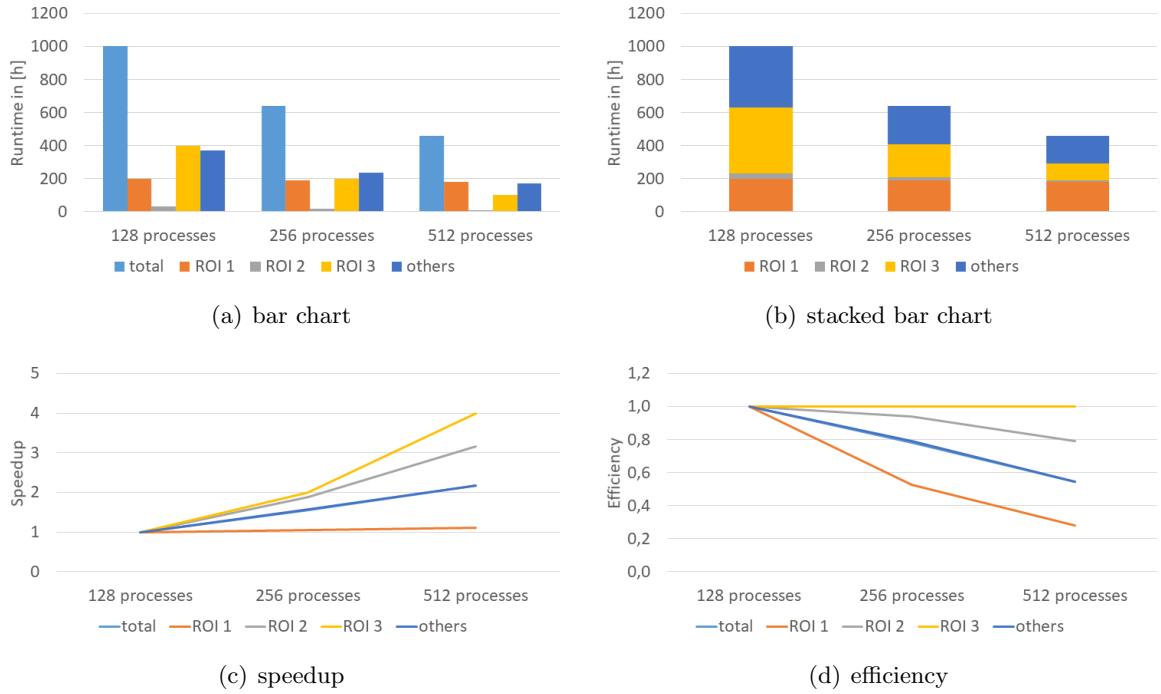
	128 processes	256 processes	512 processes
total	1000 h	640 h	460 h
ROI 1	200 h	190 h	180 h
ROI 2	30 h	16 h	9.5 h
ROI 3	400 h	200 h	100 h
others	100 h	90 h	85 h

Table 1: Runtime for an example application running with 128, 256 and 512 processes.

This is useful to present all the data exactly and if you look at the report later to compare it to a new version you still have all the numbers available, but for a larger number of regions and processes this gets confusing very easily. A better way to present the data is to have a chart, like a bar chart (Figure 2(a)) or a stacked bar chart (Figure 2(b)). Here you can easily identify how the runtime changes for all regions when the number of processes changes. A downside of these charts is that it gets hard to read the values for small regions, like ROI 2 in the example. However, since the regions of interest typically represent a large share of the execution time, these bar charts are appropriate in many cases. The stacked variant of the chart should be used if many different measurements are shown as it only uses one bar per measurement. The downside of the stacked variant is, that it is a bit harder to read the exact values and to compare differences in runtime for a region. The advantage is that it uses only one bar per measurement and thus uses less space. For a large number of measurements and ROIs it is often easier to read.

A very important point when presenting performance measurement results is to mention the used hardware resources for the experiments. As the data used here is purely fictitious, no hardware details are given. But, in a real scenario you should report enough data on the hardware and software environment to allow reproducing the experiment later on. If you have implemented some optimization and want to compare the performance to prior results, the same hardware must be used. Furthermore, the expected performance gain depends on the hardware usage. If e.g. 16 MPI processes are started on a single node, often the speedup is limited by shared resources like the memory bus or last level caches and a linear speedup cannot be expected in such cases. If a run with 16 MPI processes is done on 16 nodes, a higher speedup can be expected. Of course this is because 16 times more hardware is used. In plots where only MPI processes are shown, like the once shown here, you should mention the exact distribution

of processes to cores in the text of the report.



(a) bar chart

(b) stacked bar chart

(c) speedup

(d) efficiency

Figure 2: Different charts to present the performance for different regions of interest (ROI) and a different number of threads.

Speedup and Efficiency A different way to look at the data is to focus on the performance gain through parallelization. For this cases you should use a speedup plot. The speedup describes how many times an application gets faster compared to a reference run. It can be computed as $reference\ time \div actual\ time$. The reference run is often done serially but if the program cannot be executed serially also a higher number of processes and threads can be used. Figure 2(c) shows a speedup plot with 128 processes used as reference.

In such a speedup plot it is very easy to differentiate the scaling behavior of different regions. This can help to identify in which regions the potential for improvements in the parallelization is the largest. However, information on the actual runtime is lost in such plots. It does not necessarily make sense to focus on a region with a bad scaling behavior, if it only consumes a very small amount of the overall runtime. Therefore, you should always give a reference runtime to provide all useful information.

A different way to show the scaling behavior is an efficiency plot. Here the speedup is shown in relation to the linear speedup. An efficiency of 95% means, that 5% of parallel overhead were added by the parallelization between the reference run and the actual run. This plot can give a good overview on how well the hardware is utilized.

Overall in the section *Scalability Analysis* you should discuss the performance changes with an increasing core count for the application at all, and also for different regions of interest, if appropriate. This can give a hint which details are of relevance in later sections. For example ROI 3 in the given example consumes more runtime than other ROIs with 128 processes, but the scaling is linear. Therefore, if 128 processes is a typical setup, the serial performance of this region should be further investigated in detail. Looking at load balance or communication for this region is not very useful, as the scaling is already liner. ROI 1 in contrast does not scale

at all, if the application should be run with many processes in production the parallelization of this region needs to be investigated.

5 Efficiency Metrics

In the next section our recommendation is to presents efficiency metrics for the investigated application. Efficiency metrics give an overview of how well the parallelization of the applications works and how efficient the hardware is used. Each efficiency metric investigates one source of common inefficiency in a parallel program. Values for a metric are typically between 0 and 1 (0% and 100%) and describe how efficient the compute resources are used with respect to a certain metric. For example, a `Global Efficiency` metric of 80% means, that overall 80% of the compute resources are used for useful computation and 20% is used for some overhead introduced by the parallelization of the code. In general we consider efficiency metrics of 80% or higher as acceptable whereas values below this indicate an issue which should be investigated further.

The metrics are organized in a hierarchy which allows to drill down from the top level `Global Efficiency` to specific inefficiencies in a program. Presented in a table they allow to present a quite detailed overview of the parallel performance of an application in a very condensed form.

	128 processes	256 processes	512 processes
Global Efficiency	83%	72%	62%
↪ Parallel Efficiency	83%	78%	69%
↪ Load Balance Efficiency	91%	86%	77%
↪ Communication Efficiency	92%	91%	90%
↪ Serialization Efficiency	95%	95%	95%
↪ Transfer Efficiency	97%	96%	95%
↪ Computation Efficiency	100%	93%	89%
↪ IPC Scaling Efficiency	100%	97%	97%
↪ Instruction Scaling Efficiency	100%	95%	92%

Table 2: Efficiency Metrics for an example application running with 128, 256 and 512 processes.

Table 2 shows an example of such efficiency metrics for a fictive application. The metrics have been developed for parallel applications using MPI. These metrics can be calculated with different measurement tools. Here, we do not want to focus on one tool and present how this tool is used to generate a certain metric. However, exercises and solutions on how to create the metrics with the tools developed by POP partners can be found in the POP website under learning material[2].

5.1 Metrics explanation

Here we want to concentrate on the meaning of all metrics and which performance issues they indicate.

Global Efficiency This is the top level metric in the metric tree. It summarizes in one number how well the parallelization of an application works. A high value in the `Global Efficiency`, e.g. 90%, indicates that the parallelization is working well and no further analysis is needed

[2] https://pop-coe.eu/further-information/learning-material

here. For any optimization plans for this application the user should concentrate on the serial performance, details are presented in this document in section 6.2. The `Global Efficiency` is compound of `Parallel Efficiency` or `Computation Efficiency`. This means, if a bad `Global Efficiency` is reached, the reader can go down the tree to those two metrics. One or both of these metrics will have a low value as well and this indicates in which direction to search for the inefficiency in the ROI. In the example shown in table 2 it can be observed, e.g. for 256 processes, that parallel overhead is an issue for this code, since the `Global Efficiency` is only 72%. Furthermore, `Parallel Efficiency` is lower than the `Computational Efficiency`, so the most dominant performance issue should be found in one of the subcategories of `Parallel Efficiency`.

Parallel Efficiency The `Parallel Efficiency` summarizes all performance issues which are directly observable at synchronization or communication constructs. For a MPI program most of the time spend in a MPI routine is part of this class. Of course all time spend for synchronization and communication is overhead introduced through parallel programming. The `Parallel Efficiency` describes the share of time left for computation besides all these constructs. For the example in table 2 and 256 processes the value of 78% indicates that 22% of the overall time is used for communication and synchronization and 78% is left for computation. When the `Parallel Efficiency` metric indicates there is much time spend in MPI, this can have several reasons. Therefore, this metric can be subdivided into `Load Balance Efficiency` and `Communication Efficiency`.

Load Balance Efficiency This metric summarizes inefficiencies caused by load imbalance in the application. Figure 5.1 shows an example timeline for a program executing with four processes (P0-P4).

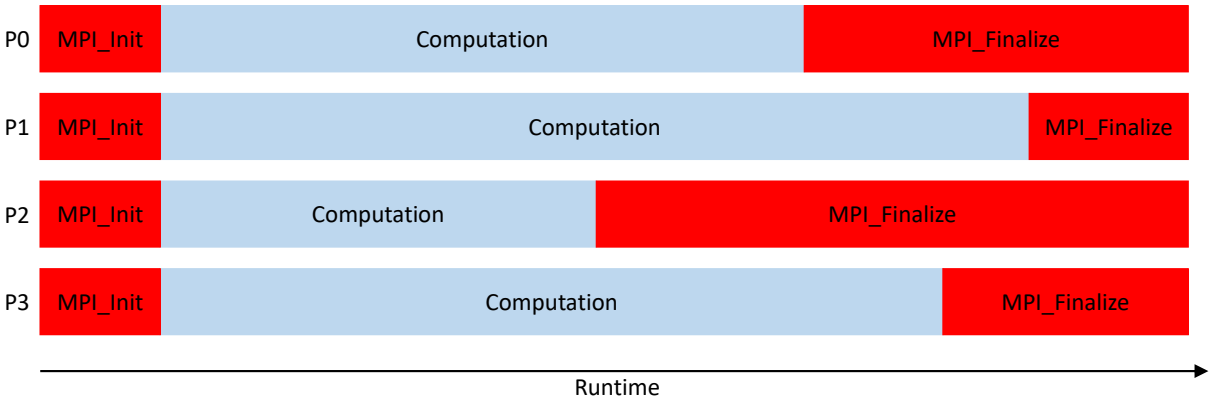


Figure 3: Timeline visualization of four processes executing a simple program with a major load imbalance between processes.

All processes call MPI_Init and MPI_Finalize. In between they spend time in a computation phase. The inefficiency in this program obviously is, that the computation phase takes longer for P1 than for the other processes, so they have to wait in MPI_Finalize. As mentioned before, all sorts of inefficiencies handled as part of the `Parallel Efficiency` lead to time lost in synchronization and communication routines, like MPI_Finalize in this example. In a profile a long time spend in MPI_Finalize could be observed. However, the reason for this issue is the different time spend in computation and this is where a user should start fixing the issue.

Therefore, if you report a low `Load Balance Efficiency` this gives more detailed information than a low `Parallel Efficiency`.

The metric can be computed by just comparing the time of all processes spend in useful computation. As mentioned before details on how to do this with Extrae/Paraver or Score-P/Scalasca can be found on the POP website. But, this kind of information can also be obtained by most other performance analysis tools.

If a low `Load Balance Efficiency` is found, you should further investigated this and report details in your audit in the section "Load Balance". Details on how this can be done are given below in section 6.1 of this guide.

Communication Efficiency This metric handles time spend in communication routines, like MPI_Send, MPI_Recv or any other data transfer routine. In contrast to the previous metric, here the reason for the handled overhead is that processes need to communicate and exchange data, whereas load imbalances typically are visible at synchronization points. Inefficiencies handled as part of `Communication Efficiency` have a different reason and need to be fixed in a different way. In general this metric can be further divided into two submetrics `Serialization Efficiency` and `Transfer Efficiency`. If this metric turns out to be low, which indicates a performance issue with communication, it should be analyzed in more detail in a separate section as described later in this document in section 6.3.

Serialization Efficiency As mentioned above, this metric is a submetric of `Communication Efficiency`.

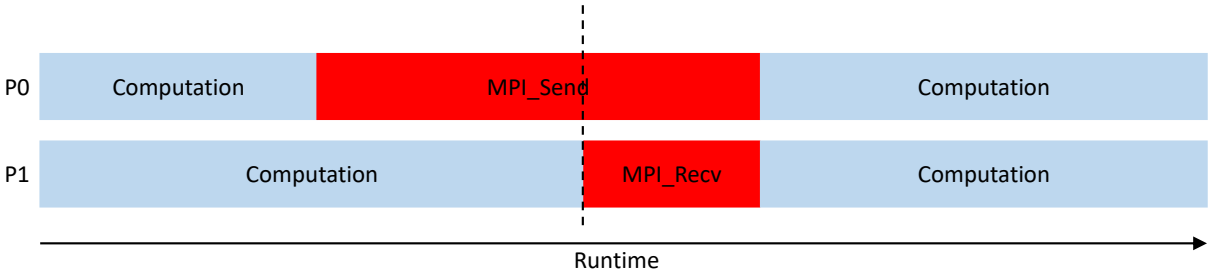


Figure 4: Timeline visualization of a message send with MPI. P0 needs to wait for P1 to arrive before the actual data transfer starts.

Most communication in MPI requires two or more processes to participate in a data transfer. An exception is so called one-sided communication which can be performed by one process through RDMA. All other calls require a so called hand-shake of the processes before the actual data transfer starts. In figure 4 a simple message send with MPI between two processes is shown. Here it can be observed, that P0 is ready to send data much earlier than P1 is ready to receive it. The actual time lost through data transfer is only the part right of the dashed line. The time in MPI_Send left of the dashed line, P0 is just waiting. This kind of overhead, which occurs when processes wait for other communication partners to arrive is handled in the `Serialization Efficiency`.

Figure 5 illustrates the difference between serialization and load imbalance. Here three processes start an MPI program and all have a compute phase of the same length, so we have a `Load Balance Efficiency` of 100% here. However, P2 waits for a message of P1 before starting to compute and P3 waits for P2. This leads to an execution where all compute regions are calculated after each other, meaning serialized.

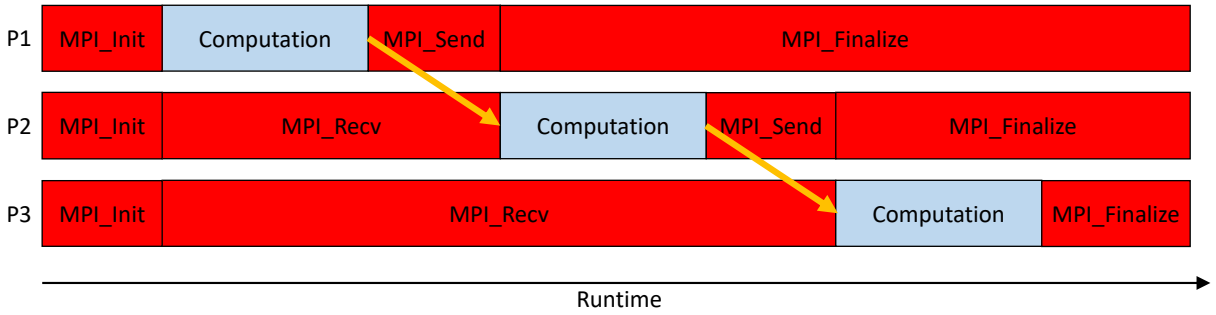


Figure 5: Three processes performing dependent computations which leads to serialization in the program execution.

Transfer Efficiency The part in figure 4 right of the dashed line where the actual data transfer happens is handled as `Transfer Efficiency`. This metric indicates how much time is spend not in actual data transfer. For example the `Transfer Efficiency` of 95% for 256 processes in the example shown in table 2 means that 5% of the overall time an application process is transferring data. Note, that there might be other data transfer happening in the background, for example if asynchronous MPI calls are used, but this is not seen as a performance problem, as long as all application processes are doing useful computation during this time. Therefore, asynchronous transfers do not need to be handled by any efficiency metric, but time spend waiting for such transfers, e.g. in MPI_Wait is part of the `Transfer Efficiency`.

Computation Efficiency All metrics mentioned so far handle inefficiencies which are directly observable as parallel overhead. In a performance profile or trace these inefficiencies show up as time spend in communication of synchronization constructs. `Computation Efficiency` in contrast, is related to inefficiencies in parallel programs which are not observable as direct parallel overhead. This means that no time is spend in additional routines, like MPI routines, but the computation phases in between are prolonged. To get an idea if the computation was prolonged, a reference is needed. Therefore, `Computation Efficiency` uses the execution with some number of processes as basis and then presents the efficiency for a different number of processes in relation to this reference run. In the example shown in table 2 the reference is with 128 processes. So, here all values are 100%. The value of 89% for 512 processes indicates that 11% of overhead was added to the computation phases of the application. This on a high level can have two reasons as explained below.

Instruction Scaling Efficiency The first reason why the computation parts take overall a longer execution time is, that parallelization might involve extra computation. For example the work decomposition must be computed or some setup is done redundantly by all processes. This is of course overhead of the parallelization, but it is harder to identify than time spend in communication routines. To compute this inefficiency hardware performance counters can be used to measure the total number of instructions executed by all processes outside of any synchronization or communication construct. This number would stay constant for a perfect parallelization with linear speedup, of course. The `Instruction Scaling Efficiency` presents how much overhead is added through additional instructions executed relative to the reference run.

IPC Scaling Efficiency A second reason for longer execution time of computation regions is that the hardware might execute code slower when more processes are used. This is typically

because of shared hardware resources, like a shared cache, memory or network. The effect is that a core executes less instructions per cycle (IPC) because it is waiting for the cache, memory or network. The IPC can be computed when hardware counters are measured for executed instructions and for cycles used.

5.2 Metrics Usage

In every performance audit you should mention these metrics if possible. A table, like the example shown in table 2, gives a lot of details about the parallel execution and efficiency of an application in one single view. Even if everything is performing well and all efficiencies are close to 100%, it is good to report these number for later reference.

Of course, the `Computation Efficiency` and its submetrics only make sense if a reference run is available. If only one configuration is tested these metrics will always be 100% and can be omitted. If no hardware performance counters are available on the target system `Instruction Scaling` and `IPC Scaling` have to be omitted as well, as they require hardware counters. In such cases you can only report the `Communication Efficiency` based on time measurements.

After presenting this table in an audit you should interpret the presented data. As mentioned before, a rough guideline is that an efficiency below 80% needs further investigation. The report should highlight which metrics show a low efficiency and how they develop with an increasing process count. This will set the stage for the deeper analysis presented in the later sections of the report.

For the example in table 2 we highlight, that the `Parallel Efficiency` seems to be the most important performance bottleneck. The `Computation Efficiency` with 89% in the worst case is still okay. Looking into more details reveals that load imbalance seems to be more severe than actual communication in this case. This means that a deeper analysis for the load balancing should be presented in the later section about load balance in the POP report. The section about communication can be rather brief. Overall the performance for 128 processes is still okay for the overall application. If 512 processes is the target to run on with this application, more tuning should be performed, starting with the load balancing.

6 Deeper Analysis

After giving an overview of the performance of the application by looking into the efficiency metrics, several sections handle different inefficiencies in more detail. In contrast to the metrics in the previous section which we recommend to present in all performance audits for comparability, the following sections should handle only inefficiencies which are present in the application and try to give more insights. The efficiency metrics determine which deeper analysis section(s) are in the focus. For example, if the `Load Balance Efficiency` was 99%, the section on load balance might just contain the sentence: "As seen in table 1, the load balance in this application is nearly perfect." The following three sections require some expert knowledge on the topics investigated and different action for all applications. Here, we will give some hints and advises based on our experience in the POP project. If this is not sufficient you might want to apply for a POP performance audit by a POP partner on the POP website[3].

[3] https://pop-coe.eu/request-service-form

6.1 Load Balance

As mentioned in section 2 in the first deeper analysis you should investigate load balance issues of the application. Of course we cannot give a recipe how load imbalance is avoided in all parallel applications in general as this is highly problem dependent, but some points which you might investigate are:

- Locate where the load imbalance happens, this might be a certain region in the source code or a phase during execution.
- Check if there is one processes extremely slow and the rest is well balanced or if all processes have different execution times. A reason might be, that one process does extra computation or IO.
- Investigate if the load balance changes over time or if it is a fix amount in every iteration.
- Check if processes perform a different amount of instructions or if they have a different IPC, both can lead to a different execution time.

After giving more details on the load balance performance issue in this section, you should, if possible, give recommendations on what can be done to tackle these issues. Some general hints on load balancing, depending on the issues found are:

- If there are regions which are executed only by one process, investigate if they can be parallelized.
- If processes perform a different number of instructions, the work distribution algorithm of the application should be investigated. Maybe a different distribution is possible which distributes the instructions more evenly.
- If the load balance gets worse over time, check if a rebalancing step after a certain time is useful.
- If a different IPC is achieved by different processes, check if the cache and network usage differs. Maybe this should be taken into account in the work distribution.
- Check if a dynamic work distribution is possible and useful, this can help to handle load imbalances at runtime.
- If a dynamic approach is already used, check if the size of work packages is too large and needs to be adjusted.

6.2 Serial Performance

In this section you should handle issues related to the serial performance of the investigated application. This means parts where every thread or process performs computation, it does not mean that an application is executed serially with only one process. What is mainly investigated here is, if the compute hardware is used in an efficient way. A first indication can be given by the number of instructions executed per cycle (IPC) on a core. This value was already calculated to determine the `IPC Scaling Efficiency` metric, so it should be present at this point already. In this section IPC can be investigated in more detail, for example for different regions in the code. To understand if an IPC value is good or bad, it is important to know what is the limit of the architecture. A modern x86 based Intel server chip, which is the dominant architecture in HPC at the moment, is able to finish up to 4 instructions per cycle because of superscalarity in

the chip. This means an IPC of 4 would be the theoretical peak performance. However, this is typically not achieved in an application. Our experience in POP has shown, that an IPC above 1 is good. If an IPC is lower than 1 for a region, further investigation and improvements might be useful. If you calculate an IPC for a POP performance audit, it is important to exclude any communication and synchronization constructs. The reason is, that these constructs often use spin waiting to synchronize and this executes a lot of instructions polling a lock. This can be done very fast, since the lock can be kept in L1 cache and this artificially increases the IPC value. And of course long waiting time in a lock is not a sign of efficient recourse utilization.

If for a certain region a low IPC value is observed, this means that the core is waiting between executing instructions. The most common reason is that it is waiting for data from main memory. To further investigate this, the following investigations can be performed with hardware performance counters:

- Use counters to measure how much data is transferred over the memory bus per second. Typically counters to measure the memory instructions exist or counters to measure the last level cache misses. Multiplying this with the cache line size and dividing it by the time in seconds delivers the bandwidth reached in the algorithm in GB/s. If this value is close to the peak value of the system, the memory bandwidth is a bottleneck in the application for this region. Sometimes blocking can help to have more reuse of data in the cache and this bottleneck can be circumvented. Sometimes it is a property of the algorithm and has to be accepted.

- Comparing L1 cache accesses and L1 cache misses allows to determine how many items of a cache line are used. Ideally, data is aligned in memory in such a way that all the data on a cache line is used in the algorithm. For example if 8 byte double values are loaded and the cache line size is 64 byte, having a cache miss for 8 cache accesses is a sign that all values on the cache line are used. If every access is a miss, this indicates that only one value per cache line is used. In such cases it might be possible to achieve a better performance with a different data layout in the algorithm. Changing arrays-of-structures into structures-of-arrays is a typical hint in many such cases. Although this is more important on GPGPUs where alignment is much more important, it can also be beneficial on CPUs to have a better reuse of data in the cache.

- On servers with two or more sockets every processor typically has its own memory attached, these architectures are called NUMA architectures. The application can access any memory in the system with the same load and store instructions. But, accessing the local memory is faster for a processor than accessing the memory of a different socket. To find out if a lot of remote memory accesses are present during execution, hardware counters can be used to measure local memory accesses and remote memory accesses. This allows to compute the rate of remote memory accesses. If this value is high, for example 50% of all accesses go to remote memory, this is likely a performance problem. The programmer should optimize the data distribution during initialization or migrate the data later on between sockets.

If a bad value for the `IPC Scaling Efficiency` has been determined, the experiments mentioned above should be done for multiple configurations. Comparing results might help to understand why the IPC is going down for an increasing number of processes. One reason might be that some data structures increase in size with the number of processes and this can lead to a different cache behavior, influencing the IPC value.

6.3 Communication

The next section handles the communication done between processes during execution. Here, deeper analysis results should be presented if a bad `Communication Efficiency` was determined before. This section can present information on:

- Which kind of MPI communication is taking how much time.
- Where in the code or when during execution do these calls occur.
- How frequent are the MPI calls.
- What is the typical message size.
- Can some kind of pattern be observed, like one process communicating with all others pair-wise.
- If `Serialization Efficiency` was an issue, which processes have to wait for which others?
- Does the communication change over time.

In this section you should try to give gives a hint where the inefficiency is present in the code. In general it is very hard to propose a general optimization here, because the communication is part of the algorithm and algorithmic optimization requires in depth knowledge of the application and therefore needs to be handled individually.

If the `Transfer Efficiency` is an issue for an application, a general hint is to look for potential to overlap communication with computation phases. In some cases you can observe potential to do this by looking in the code of a communication intensive routine.

7 Summary

The last section in an audit is a summary of all results. Here you should giva a brief overview of the investigated ROIs and the main findings. Furthermore, if possible recommendations for further actions should be listed. This might be a deeper analysis of a performance problem which was detected. But it might also be a recommendation for a tuning action to perform or investigate for the application code. If a specific performance issue is tackled by the tuning action the report should also try to name the benefits which can be reached. This can often be estimated based on the efficiency metrics. For example, if the `Load Balance Efficiency` was 70%, a tuning action distributing the load in a better way should target to increase the performance by 30%, but it will not reach more than that, because only 30% of the time was wasted due to load imbalance.

8 Tools Recommendations

As mentioned earlier, this guideline is meant to be independent of the tools used and should mainly help to interpret and present performance results in general. However, if you are not familiar with a specific performance tool you prefer, of course you need to choose one to do such an audit. During the POP project we mainly used the performance tools developed by POP partners, which are Score-P/Cube/Scalasca on the one hand and Extrae/Dimemas/Paraver on

the other hand. These tools are freely available for download and we encourage you to also use them to do a performance analysis on your own.

Here are some pros and cons about both tool packages you might want to consider when you choose on of the packages for analysis, in our experience both sets complement each other well. You can find guides for all tools on the POP project website[4].

Score-P/Cube/Scalasca These tools are most useful if you can recompile and relink your application. Then Score-P can automatically add instrumentation to the code and allow a detailed analysis on a call path level. Performance results will be presented for individual functions. This is often useful to see which parts of the source code need further investigation. If you do not have access to the source code, we would recommend not to use these tools.

One big advantage of these tools is, that they allow to do a profile first. A profile typically produces much less data. So, if your application will likely produce a large amount of data, e.g. because you need many processes and longer runs to have certain performance problems, you should try these tools and start with a profile first.

The main visualizer here is the Cube GUI. Here you get an overview of certain metrics and their distribution on functions and resources. A downside is, that you do not get a timeline overview of the application behavior. Such a view typically is the easiest way to look at performance data for beginners as they are quite intuitive to understand. If the commercial Vampir tool is also available for you, this can be used to visualize the Score-P traces.

Extrae/Dimemas/Paraver Extrae does not require recompilation of your application, you can just preload a measurement library. If you only have access to a binary, we recommend to use these tools.

A downside often seen with this set of tools is that a large amount of data is produced during tracing. For a long run with many processes this can exceed the capacity of your disk storage and these very large traces are harder to analyze. If you only have certain performance problems with very large configurations, this can also be handled by these tools, but it might be easier to start with Score-P/Cube/Scalasca in such cases.

The Paraver analyzer is very powerful but for a beginner it is sometimes hard to handle. But, if you did the first steps, e.g. by going through some of the prepared tutorials, the timeline view gives a nice compact overview of the application behavior over time.

[4] https://pop-coe.eu/further-information/learning-material