Promoting HPC Best Practices with the POP Methodology

Fouzhan Hosseini
The Numerical Algorithms Group
Manchester, UK
fouzhan.hosseini@nag.co.uk

Craig Lucas
The Numerical Algorithms Group
Manchester, UK
craig.lucas@nag.co.uk

ABSTRACT

The performance of HPC applications depends on a wide range of factors, including algorithms, programming models, library and language implementations, and hardware. To make the problem even more complicated, many applications inherit different layers of legacy code, written and optimized for a different era of computing technologies. Due to this complexity, the task of understanding performance bottlenecks of HPC applications and making improvements often ends up being a daunting trial-and-error process. Problematically, this process often starts without having a quantitative understanding of the actual behavior of the HPC code.

The Performance Optimisation and Productivity (POP) Centre of Excellence, funded by the EU under the Horizon 2020 Research and Innovation Programme, attempts to establish a quantitative methodology for the assessment of parallel codes. This methodology is based on a set of hierarchical metrics, where the metrics at the bottom of the hierarchy represent common causes of poor performance. These metrics provide a standard, objective way to characterize different aspects of the performance of parallel codes and therefore provide the necessary foundation for establishing a more systematic approach for performance optimization of HPC applications. In consequence, the POP methodology facilitates training new HPC performance analysts. In this paper, we will illustrate these advantages by describing two real-world examples where we used the POP methodology to help HPC users understand performance bottlenecks of their code.

KEYWORDS

Parallel performance analysis, HPC performance optimization, POP metrics

1 INTRODUCTION

High-Performance Computing (HPC) is an essential tool for science and industry. It is used to solve diverse problems such as weather forecasting, material design, drug discovery, climate modeling and predictions, etc. Most HPC facilities represent a major capital investment and run at a high level of utilization. Improving the efficiency of application software running on these facilities means less time to solution and more capacity to solve larger, more challenging problems.

Today's HPC facilities allow using hundreds to hundreds of thousands of cores to perform extensive calculations and process large amounts of data. Efficient use of these facilities has been proven to be extremely challenging. While HPC applications are often designed with performance in mind, many suffer from poor performance; here, unexpected behavior is more likely to happen, and an excellent design choice for a given problem size and a particular hardware might lead to poor performance for a different configuration.

There has been significant research and engineering effort to build tools that support performance optimization of HPC applications by collecting and analyzing their runtime behavior; for examples, see [5]. However, profiling or tracing HPC applications often results in large amounts of performance data that are difficult to interpret beyond simple observations. There is often a lack of a quantitative understanding of the actual behavior of parallel applications, and the performance optimization task, in turn, becomes a trial-and-error and ad-hoc process.

Performance Optimisation and Productivity (POP) is a Centre of Excellence (CoE) in HPC funded by the EU under the Horizon 2020 Research and Innovation Programme. The mission of POP [6] is to provide performance optimization and productivity services for HPC applications in all domains. POP attempts to achieve this while establishing general principles and systematic methods for parallel performance optimization.

POP has defined a scalable performance analysis methodology based on a set of hierarchical metrics [7], where each metric represents a common cause of inefficiency in parallel applications. These metrics are calculated using basic runtime statistics and provide a standard, objective way to characterize different aspects of performance of parallel codes, such as communication overhead and load imbalance. These metrics allow comprehensive comparison of the performance of a parallel application across different platforms or with different configurations (e.g. different numbers of threads/processes). This allows for a better understanding of program efficiency, quick diagnosis of performance problems, and identification of target kernels for code refactoring.

In Section 2 of this paper, we provide an overview of POP metrics for Message Passing Interface (MPI) applications and their calculation. Section 3 and 4 review performance assessments of two parallel applications using POP methodology. These examples are chosen from different domains: molecular dynamics simulation, Section 3, and computational fluid dynamics, Section 4. Section 5 concludes the paper.

https://doi.org/10.22369/issn.2153-4136/12/2/13

2 POP METRICS FOR PARALLEL PERFORMANCE ANALYSIS

In this section, we review the MPI performance metrics used and promoted by the POP CoE [7]. These metrics measure relative impact of parallel inefficiency factors on overall performance and provide a quantitative understanding of parallel application behavior.

The hierarchy of POP metrics for pure MPI applications, in fact applications written using any message-passing model, is shown in Fig. 1. The *Global Efficiency* at the top of the hierarchy indicates how well a parallel application scales. At this level, inefficiencies are typically due to two factors:

(1) overhead imposed by parallelism, represented with *Parallel Efficiency*, and
(2) poor scaling of computation with increasing number of processors, represented with *Computational scaling*.

The *Global Efficiency* is defined as product of the *Parallel Efficiency* and the *Computational scaling*. Going further down the hierarchy, both of these metrics are defined as the product of their own sub-metrics.

For MPI applications, the *Parallel Efficiency* reports inefficiencies due to either uneven distribution of computational work or overhead of data communication and synchronization between processes. These are measured with the *Load Balance* and the *Communication Efficiency*, respectively.

These two metrics are calculated using basic statistics from a program execution, including the total runtime and the computation time per process. Here, the computation time refers to the time that useful instructions are being executed, e.g. it excludes the CPU time in the MPI library. The *Load Balance* is defined as the ratio of the average computation time of all processes to the maximum computation time across all processes. The *Communication Efficiency* is defined as the ratio of maximum computation time to the total runtime.

The *Communication Efficiency* includes two metrics: *Transfer Efficiency* and *Serialization Efficiency*. The former indicates performance loss due to actual data transfer time. The latter reveals communication inefficiencies due to idle time within communication, i.e. when no data is transferred. This happens when processes wait at communication or synchronization points for other processes to arrive. To calculate these two metrics, we need to calculate the total runtime of the application on a system with an ideal communication network, i.e. what the runtime would be if data transfer were instantaneous. The *Transfer efficiency* is the ratio of the runtime on an ideal network to the runtime on the real system, and the *Serialization Efficiency* is the ratio of the maximum computation time to the total runtime on an ideal network.

Going up in the metrics hierarchy, the *Computational Scaling* shows how well the computation load scales with increased parallelism. It is calculated with respect to a reference execution case using total computation time, i.e. the time spent executing useful instructions summed over all processes. For example, when analyzing strong scaling behavior, it is calculated as the ratio of the total computation time for a reference case such as one processor (or one node) to the total computation time as number of processors (or nodes) is increased.

Multiple issues can lead to a poor *Computational Scaling* value, and they can be investigated using hardware performance counter data via interfaces such as PAPI counters [4]. In the POP hierarchy of metrics, the *Computational Scaling* is composed of three metrics:

- *Instruction Scaling*: compares the total number of instructions executed for different numbers of threads/processes. Decreasing values of this metric indicate that total computation load increases with employing more processes.
- *Instruction Per Cycle (IPC) Scaling*: compares how many instructions per cycle are executed for different numbers of threads/processes. Decreasing values indicate that rate of computation has slowed down. Decreasing cache hit rate and exhaustion of memory bandwidth are typical causes.
- *Frequency Scaling*: compares the processor frequency for different numbers of threads/processes. Decreasing values indicate that with increasing load, some cores operate with lower frequency.

Basic runtime statistics which are needed to calculate the POP metrics can be collected using almost any performance analysis tool. However, automatic calculation of the POP metrics is supported in the tools developed by Barcelona Supercomputer Center (BSC) [2] and the Jülich Supercomputing Centre (JSC) [9]. The former family includes Extrae for collecting performance data, Dimemas for simulating behavior of MPI applications under different network conditions, and Paraver and Basic Analysis for post-mortem trace analysis, including calculation of the POP metrics. The latter includes Scalasca and Cube for parallel performance analysis; Scalasca uses Score-p [10] for instrumenting parallel applications and collecting performance data.

By definition, the POP efficiency metrics can take values between 0 and 1, with higher numbers representing better performance. As a rule of thumb, values above 0.8 are considered acceptable, whereas lower values indicate performance issues that need to be explored in detail.

3 EXAMPLE 1 - MOLECULAR DYNAMICS SIMULATION

In this section, we describe the use of the POP metrics in assessing parallel performance of a molecular dynamics simulation (MDS) code. We call this code E1-MDS. E1-MDS uses MPI and consists of a legacy core written in Fortran with a layer of modern C++ on top. We did not have access to the source code. Performance data was collected by code developers using Extrae [2], running the application on their in-house server machine with a dual Intel Xeon Gold 6248 CPU (40 cores per socket).

Extrae uses instrumentation mechanisms[1] to collect performance data at known application points (e.g at MPI function calls) and collects trace data of the application runtime behavior. All performance data can be gathered in one file for post-mortem analysis. We were given trace data for the application running on 2, 10, 20, 30, and 40 cores, solving the same problem. Given these trace files, we used Basic Analysis [2] to calculate the POP metrics.

[1] Sampling mechanisms are supported as well.

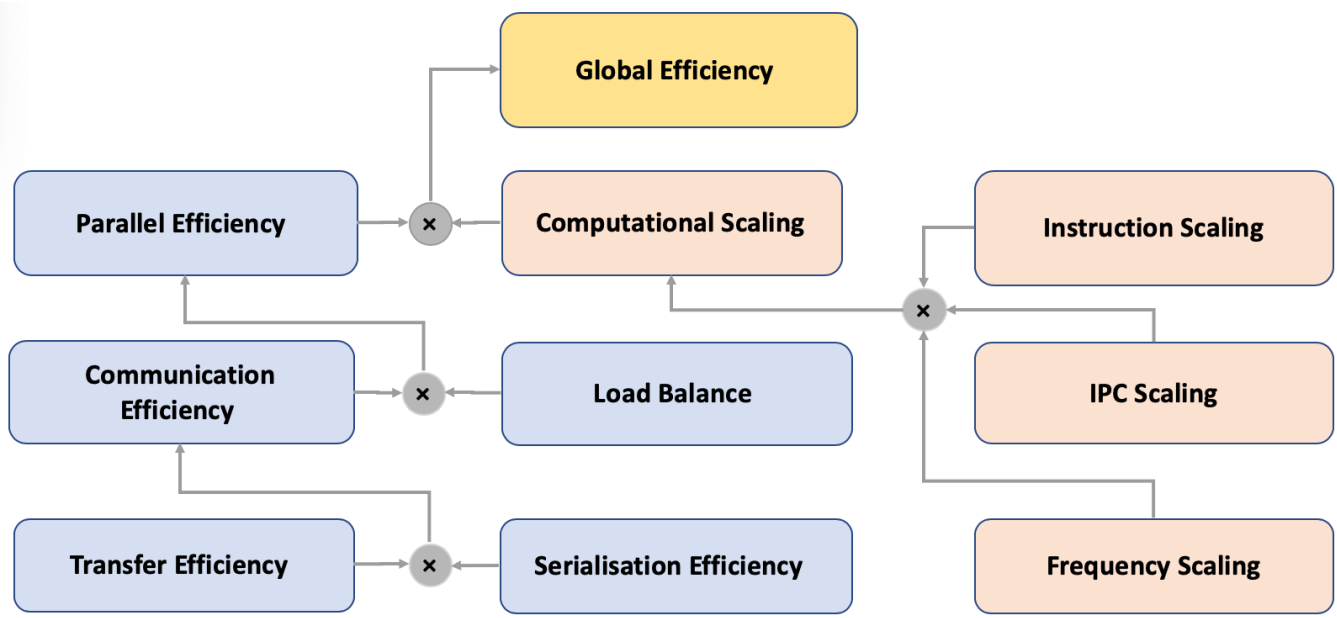

Figure 1: POP MPI Parallel Efficiency Metrics.

E1-MDS has three stages: initialization, main body of the simulation, and finalization. The time spend in initialization and finalization phases is negligible in comparison with the main body. Therefore, the performance assessment was focused on the second stage, i.e. the main body was the Region of Interest (ROI) for this assessment.

Figure 2 and 3, respectively, show the scalability plot for E1-MDS and the POP metrics calculated for ROI using 2, 10, 20, 30, and 40 cores. As shown in Fig. 2, the speedup drops below 80% of the ideal, i.e. linear speedup, on 10 cores, and it does not scale well beyond that. This is also evident in the *Global Efficiency* metric; it drops to 73% on 10 cores and gets as low as 36% on 40 cores.

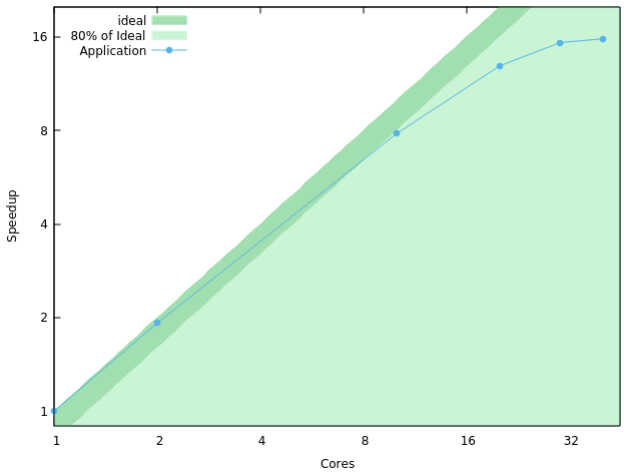

Figure 2: E1-MDS Scalability plot.

Figure 3 shows that the POP metrics decrease as the number of cores is increased. The values of these metrics reveal which factors contribute more in the loss of performance. The *Computational Scaling* drops below 80% on 20 cores, with the *Instruction Scaling* being the fastest dropping factor. The *Parallel efficiency* also drops below 80% on 30 and 40 cores, with the load imbalance being the major contributing factor. Therefore, according to the POP metrics, poor instruction scaling and load imbalance are the two main factors that limit scalability of the application.

Number of cores	2	10	20	30	40
Global Efficiency	0.95	0.73	0.60	0.47	0.36
Parallel Efficiency	0.95	0.89	0.81	0.75	0.68
Load balance	0.95	0.92	0.85	0.81	0.80
Communication Efficiency	0.99	0.97	0.95	0.92	0.85
Serialisation	0.99	0.99	0.99	0.98	0.94
Transfer Efficiency	0.99	0.98	0.96	0.94	0.91
Computational Scaling	1.00	0.82	0.74	0.63	0.53
Instruction Scaling	1.00	0.87	0.83	0.79	0.76
IPC Scaling	1.00	0.99	0.95	0.90	0.83
Frequency Scaling	1.00	0.95	0.94	0.88	0.83

Figure 3: E1-MDS, the POP metrics for ROI: poor instruction scaling and load imbalance limit scalability of the application.

The findings of the POP metrics were confirmed with further analysis of the trace data using Paraver [2]. The next step was to report our findings to the developers of E1-MDS. They could rather

quickly put their fingers on regions of the code that caused load imbalance. This, however, was not the case for the *Instruction Scaling*. It took some digging in the algorithms and the code to confirm that poor *Instruction Scaling* was due to duplicated computation. This is one of the strengths of the POP metrics. They can provide an insight into a code of which developers are ignorant.

This example showed how the POP metrics can help us to quickly diagnose the causes of poor parallel performance. This allows for a better understanding of program efficiency and the identification of target kernels for code refactoring. In case of E1-MDS, algorithmic changes are needed to make the code scalable on higher numbers of cores; however, using hybrid parallelism, i.e. OpenMP + MPI, can be a quick way to get better performance on the existing hardware with minimum code refactoring. Running the code with fewer MPI processes and using OpenMP to exploit extra free cores will improve instruction scaling and load imbalance.

4 EXAMPLE 2 - COMPUTATIONAL FLUID DYNAMICS

Our second example is a computational fluid dynamics code. It is an incompressible flow solver, and we refer to it as E2-CFD. E2-CFD uses MPI for parallelism, it is written in modern C++, and it depends on a couple of libraries for numerical computation. We had access to the source code. Performance data was collected using Scalasca [9], running E2-CFD on MareNostrum-IV [3] using 1, 2, 4, 8 and 16 nodes, where each node has 48 cores. Scalasca supports calculation of the POP metrics.

E2-CFD scales well on a couple hundred cores, and the speedup drops below 80% of ideal on 768 cores (16 nodes). The POP metrics for ROI are shown in Fig. 4. As can be seen, the *Global Efficiency* only drops below 80% on 768 cores with the *Communication Efficiency* and especially the *serialization* being the major contributing factors. The *IPC Scaling* improves on higher number of cores, likely due to better cache access. The *Instruction Scaling* also drops by about 8% on 768 cores but it is still above 90% and in the acceptable range. In short, the POP metrics suggest that for code optimization we need to find the regions of the code that cause low *Serialization Efficiency*. Serialization typically happens due to at least one process arriving early/late at a synchronization point.

Number of cores	48	96	192	384	768
Global Efficiency	0.91	0.93	0.97	0.86	0.69
Parallel Efficiency	0.91	0.87	0.87	0.75	0.58
Load balance	0.99	0.98	0.97	0.94	0.91
Communication Efficiency	0.92	0.89	0.89	0.8	0.64
Serialisation	0.93	0.91	0.93	0.87	0.73
Transfer efficiency	0.98	0.98	0.96	0.92	0.88
Computational Scaling	1.00	1.07	1.11	1.15	1.19
Instruction Scaling	1.00	0.99	0.97	0.95	0.92
IPC Scaling	1.00	1.08	1.16	1.27	1.44
Frequency Scaling	1.00	1.00	0.99	0.96	0.90

Figure 4: E2-CFD, the POP metrics for ROI: serialization is the main factor that limits scalability

To identify causes of poor *Serialization Efficiency*, we used delay cost analysis [1], which is available in Scalasca. The delay cost metric highlights the root causes of serialization by attributing processes' waiting time to the routines causing it [8].

This further analysis identified that low *Serialization Efficiency* was mainly related to a library function call, and it was caused by regions of computational load imbalance between MPI synchronization points and growing waiting time, especially in MPI collective calls.

In this example, POP metrics provide a quick insight on the causes of parallel performance loss. While we used other tools for further analysis and to locate problematic regions of code, the choice of this tool was guided by the POP metrics.

5 CONCLUSION

Attempts to optimize performance of HPC applications start with collecting performance data. This could result in large amounts of performance data that are difficult to interpret beyond simple observations. The problem is often a lack of a quantitative understanding of the actual behavior of HPC applications. To address this, POP CoE [6] has defined a set of hierarchical metrics [7], where each metric represents a common cause of inefficiency in parallel applications.

In this paper, we described the use of the POP methodology with two real-word examples. In both cases, POP metrics quickly and correctly highlighted causes of parallel inefficiency and provided the knowledge necessary to decide the best course of action to improve efficiency of the parallel applications. Both examples are production codes used in their respective communities. They belong to different domains of science and technology and run on different scales. This is the other advantage of the POP metrics; they work across domains and scales. The POP metrics establish a systematic and efficient approach for parallel performance evaluation, help HPC users to better understand performance bottlenecks of their codes, and facilitate training new HPC performance analysts.

REFERENCES

[1] D. BOHME, M. GEIMER, L. ARNOLD, F. Voigtlaender, and F. Wolf. 2016. Identifying the root causes of wait states in large-scale parallel applications. *ACM Trans. On Parallel Computing* 3, 2 (2016).

[2] BSC-tools [n. d.]. Perforamnce Parallel Tools Developed at BSC. https://tools.bsc.es.

[3] MareNostrum [n. d.]. https://www.bsc.es/marenostrum/marenostrum.

[4] PAPI [n. d.]. Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.

[5] POP [n. d.]. Parallel Performance Tools. https://pop-coe.eu/partners/tools.

[6] POP [n. d.]. The POP CoE. https://pop-coe.eu/.

[7] POP-Metrics [n. d.]. POP Standard Metrics for Parallel Performance Analysis. https://pop-coe.eu/node/69.

[8] Scalasca [n. d.]. Performance properties. https://apps.fz-juelich.de/scalasca/releases/scalasca/2.5/help/scalasca_patterns-2.5.html#delay

[9] Scalasca [n. d.]. A Software Tool for Performance Optimization of Parallel Programs. https://www.scalasca.org.

[10] Score-p [n. d.]. Scalable Performance Measurement Infrastructure for Parallel Codes. https://www.vi-hps.org/projects/score-p/.