Module1 / Application1
Performance Assessment Report

Document Information

Reference Number	POP_AR_Sample
Author	(BSC)
Contributor(s)	
Date	02.09.2016

Notices:
The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No "676553".

Content

1. Background

Applicants Name:	Sample Applicant
Application Name:	Module1, future submodule of Application1
Programming Language:	Fortran
Programming Model:	MPI, OpenMP
Input data:	Test case
Performance study:	Initial Audit with focus on scalability

The application was monitored on Intel Sandy Bridge based system. We received and evaluated four traces with 120 MPI processes each and one, two, four, and eight threads per MPI process, respectively. The traces were used to study the effects of increasing the number of threads per node, i.e. all measurements used 60 nodes running two, four, eight and sixteen threads per node, respectively. All traces were collected with Extrae 3.3.0 using detailed trace mode with no sampling and recording of hardware counters in three sets changing every 0.5 seconds.

2. Application Structure

Figure 1 depicts the timeline of the execution using 120 MPI processes with one thread each. The code executes three main phases *Init*, *Calc*, and *Last*. In the measured standalone version, *Init* makes up for about 80% of the runtime. However, this time is overrepresented in the standalone version; within a normal usage scenario, the *Calc* phase is dominating and, thus, the focus of the audit. The *Last* phase uses less than 0.1% of the time and can be neglected.

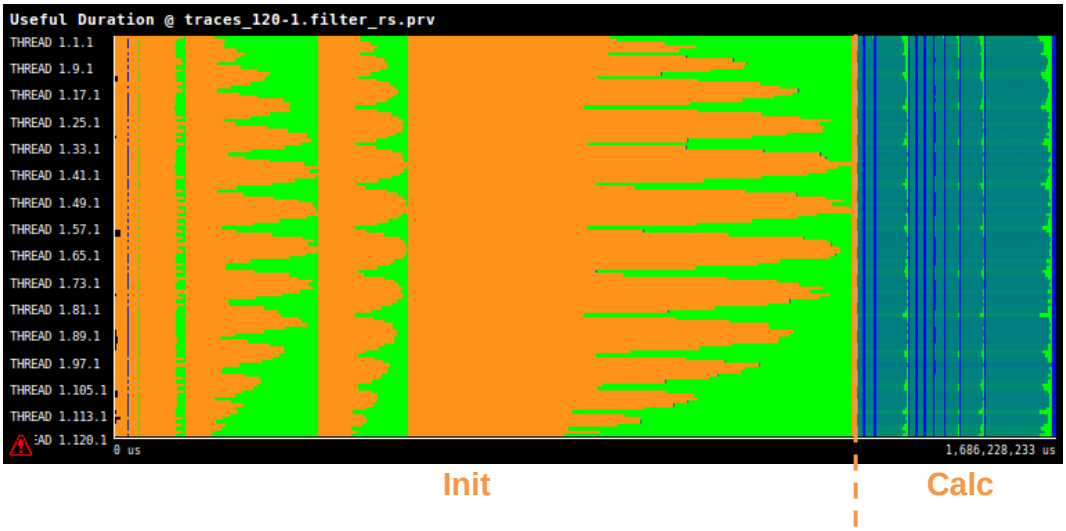
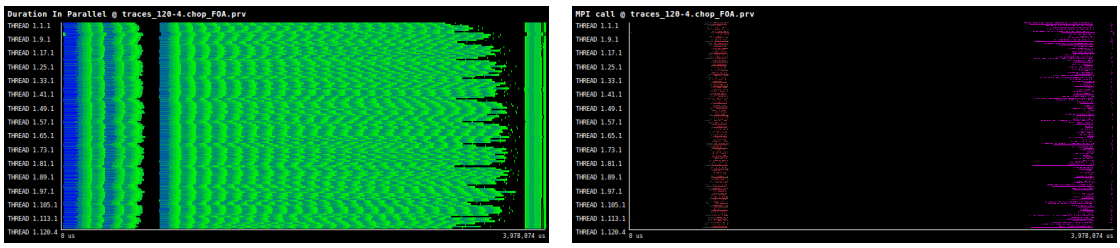


Figure 1. Application structure in a timeline view using 120 MPI processes.

3. FOA (Focus of Analysis)

The *Calc* phase includes ten main phases of varying length. Thereby, the relative timing is identical for each phase, i.e. all phases are virtually the same but are differently stretched along time. We selected the fourth of these phases as focus of analysis (FOA). Figure 2 presents the distribution of computation phases (left) and MPI communication (right) for the selected phase. Thereby, the colour gradient from green to blue represents the duration of the individual compute phases.



Useful duration **MPI calls**

Figure 2. Focus of Analysis (FOA) using 120 MPI processes with four threads each.

The FOA itself consists of three compute phases: A first phase followed by calls to *MPI_Win_create*, *MPI_Get*, *MPI_Win_fence*, and *MPI_Win_free*. The central main compute phase followed by a large *MPI_Allreduce*, which is a global synchronization and collects the load imbalance of the previous phase. At the end, a short compute phase followed by another *MPI_Allreduce*.

4. Scalability

Figure 3 highlights the scalability of the computation phases. It shows the execution structure of the compute phases of the FOA on the left side; whereas the time iskept constant for all four measurements. By the change of the gradient from green to dark blue (marking relatively longer phases), it can be seen that specifically the second and third long phase grow relative in time, i.e. they do not scale as well as the other phases.

The right side depicts the speed up of the FOA in comparison to the smallest run with one thread per process. In a perfectly linear strong scaling execution we expect that each time the number of threads doubles, the total execution time per iteration reduces by half (red line on the Speedup chart at the right side of Figure 3). The overall scaling of the FOA is fair with a Speedup of 5.5 out of 8. In addition, the trend of the curve hints to even smaller speedups for more threads per process. However, this machine is limited to 16 concurrent threads per node anyway. In addition, it must be noted that a decrease in scaling is expected since the number of active cores per node are increased with more threads per process and some resources, e.g. memory bandwidth, are shared among all cores of the node.

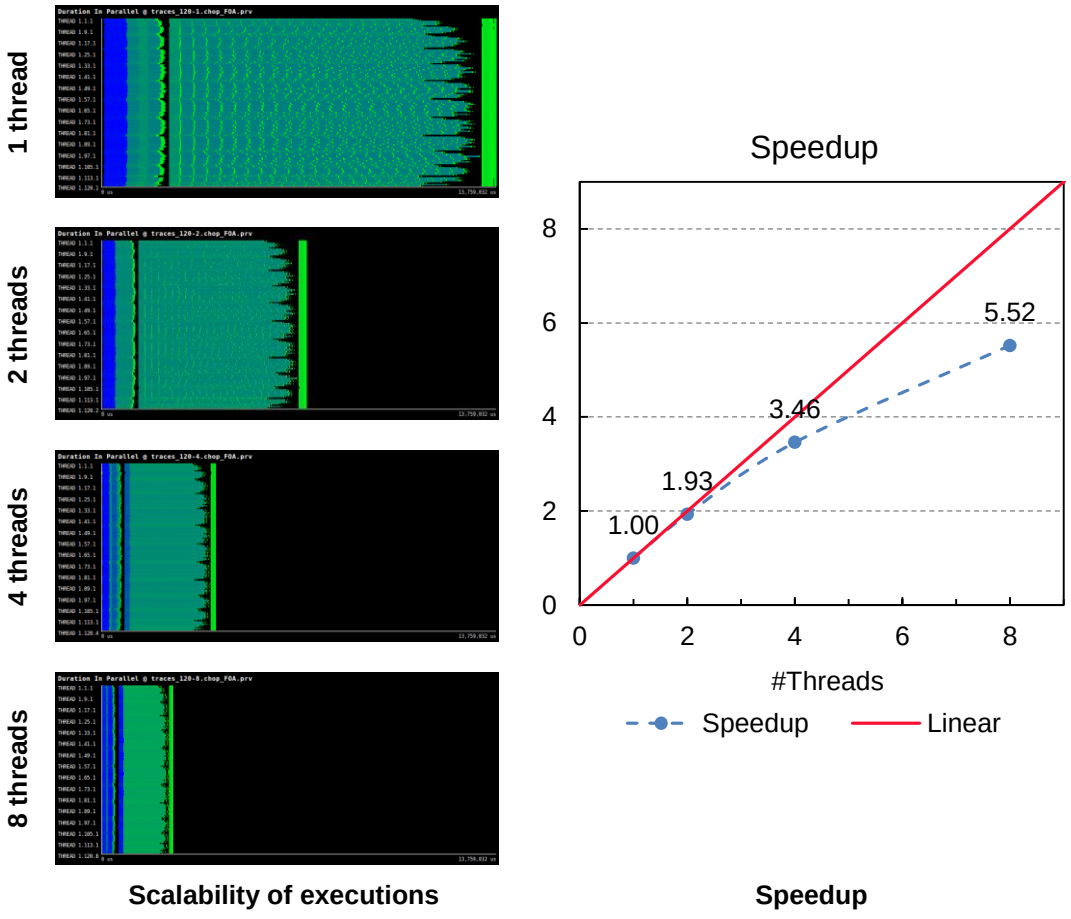


Figure 3. Scalability of FOA. Timeline of computational regions and speedup chart.

5. Efficiency

Table 1 and Table 2 show metrics for fundamental factors and efficiencies from the FOA of the executions using one to eight threads per MPI process. Values are in percentages with higher values being better.

The observed global efficiency of the application decreases steadily from 91% with one thread per process to 63% with eight threads per process. The decreasing global efficiency is mainly caused by decreasing computation scalability, i.e. an increasing amount of time (accumulated over all processes and threads) is spent in computation. While there is some replication of the workload (instructions scalability), the main reason for the decrease is the decreasing computing efficiency (IPC scalability), i.e. the number of instructions per cycle decreases. This is discussed in more detail in Section 7.

The second contributing factor is the decreasing communication efficiency (see Section 8). This is mainly due to the fact that the number of MPI processes and the volume of communication stay the same when increasing the number of threads. However, the load balance is slightly improved with more threads per nodes and achieves good values.

	1 thread	2 threads	4 threads	8 threads
Parallel Efficiency	**90.69%**	**89.69%**	**87.41%**	**84.18%**
↳ Load Balance	93.00%	93.18%	93.68%	94.65%
↳ Comm. Efficiency	97.52%	96.25%	93.31%	88.93%
↳ Serialization	99.00%	99.83%	99.80%	99.34%
↳ Transfer	98.50%	96.42%	93.49%	89.52%
Computation Scalability*	**100.00%**	**97.33%**	**89.72%**	**74.37%**
Global Efficiency	**90.69%**	**87.30%**	**78.42%**	**62.61%**

Table 1. Time efficiencies for the FOA.

	1 thread	2 threads	4 threads	8 threads
IPC Scalability*	100.00%	97.94%	93.42%	83.33%
Instructions Scalability*	100.00%	99.39%	98.97%	97.88%

Table 2. Other efficiencies for the FOA.

** Reference values are useful computation, IPC and total instructions based on three nodes.*

6. Load Balance

The observed measurements show a good load balance that is slightly increasing with the number of threads per process (93% to 94.7%). The small load imbalance is mainly due to different amount of work (number of instructions). The nested d-shape of the load distribution (see Figure 4) hints to a decomposition where the most load is in the centre of the domain and the centre of each partition. The progression of the load balance leads to the fact that at some point, the fast threads are already starting in the next phase. We tried to highlight this with the orange lines in the centre of the timeline. This continues until all processes and threads are synchronized with the large MPI_Allgather (see also Figure 8).

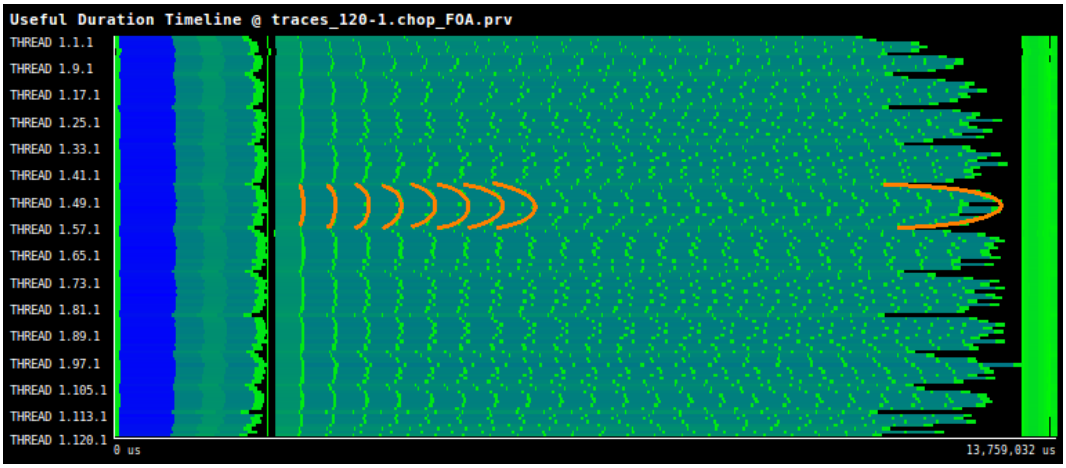


Figure 4. Load balance of the FOA with highlighting of the structure of sub-phases.

7. Computing Performance

The observed computing performance of the application averages between 1.57 instructions per cycle (IPC) with one thread per process and 1.31 with eight threads per process. In general, the application achieves an average computing performance for this machine. However, the computing performance varies a lot between the different compute phases. To further distinguish the individual compute phases we applied clustering, which groups compute phases with similar performance behaviour. Figure 5 shows the compute phases detected by clustering. It includes five main clusters (light green, yellow, red, dark green, and violet).

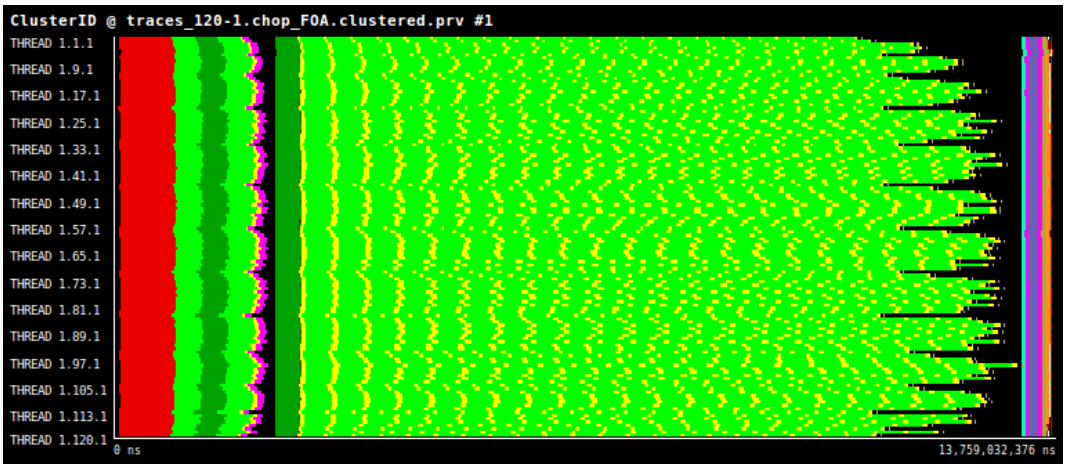


Figure 5. Clustering of compute phases in the FOA.

When comparing the scaling of the individual clusters it can be seen, that Cluster 4 (dark green) scales particularly bad. Figure 7 shows a comparison of the timeline for one and eights threads per process on the left and the total runtime increase (total amount of time spent in the cluster accumulated over all processes and threads) on the right. Both highlight the total runtime increase of over 350% in Cluster 4.

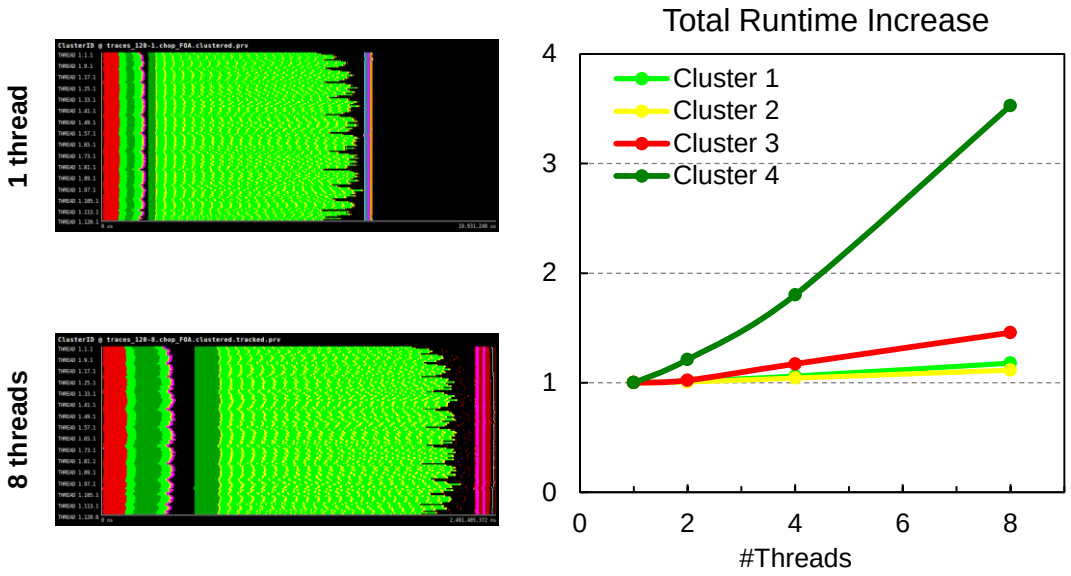


Figure 6. Scalability of FOA. Timeline of clusters and total runtime increase.

Furthermore, we used the above mentioned clustering to track the evolution of the compute performance of the individual clusters when scaling to higher core counts, i.e. more threads per process. Figure 7 shows this evolution for the above clusters. It highlights that the runtime increase in Clusters 4 is mainly due to a drastic decrease of compute performance (IPC).

Out of the various performance counters the resource stalls caused by the re-order buffer correlate the strongest with the achieved IPC in Cluster 3 and 4. Since the ratio of load instructions remains more or less the same when scaling, the increase in stall cycles due to the re-order buffer is due to an increased time to load data from memory. This is most likely caused by a saturation of the memory bandwidth or the memory controller since with increasing the number of threads per process the number of active cores per node is increased from two to 16.

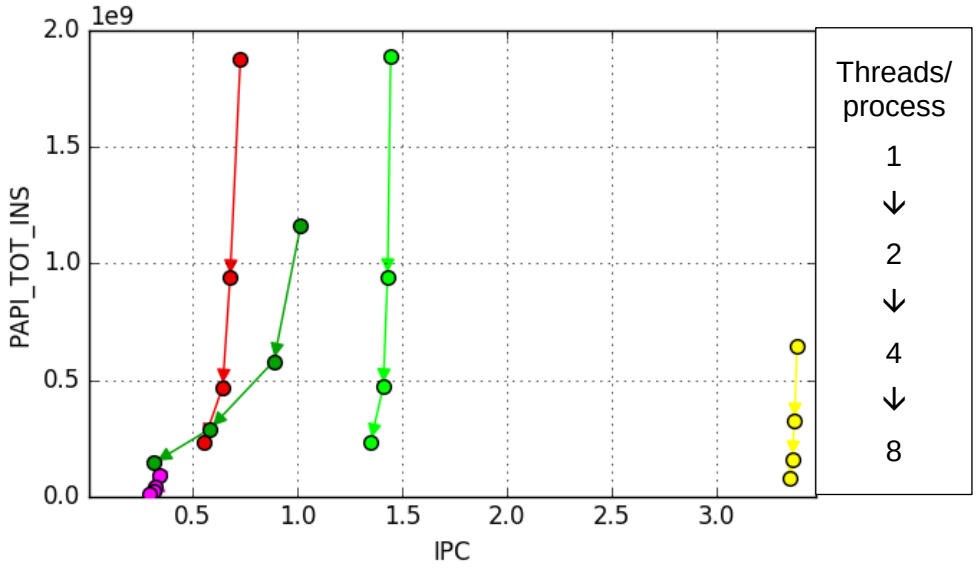


Figure 7. Evolution of compute performance for individual clusters.

8. Communications

There are two main MPI communication phases in each main compute phase: first, a set of calls to *MPI_Win_create*, *MPI_Get*, *MPI_Win_fence*, and *MPI_Win_free* and a large *MPI_Allreduce*; at the end, follows another, much shorter *MPI_Allreduce (see* Figure 8*)*. The *MPI_Allreduce* is only taking up the load imbalance of the previous computation phase and does not introduce any significant overhead. However, the first communication phase is not scaling at all since the total communication volume remains basically the same and the number of MPI process, too, i.e. the communication does not benefit from more threads. As a result the runtime share of this communication phase increases from 0.8 to 4.8 percent.

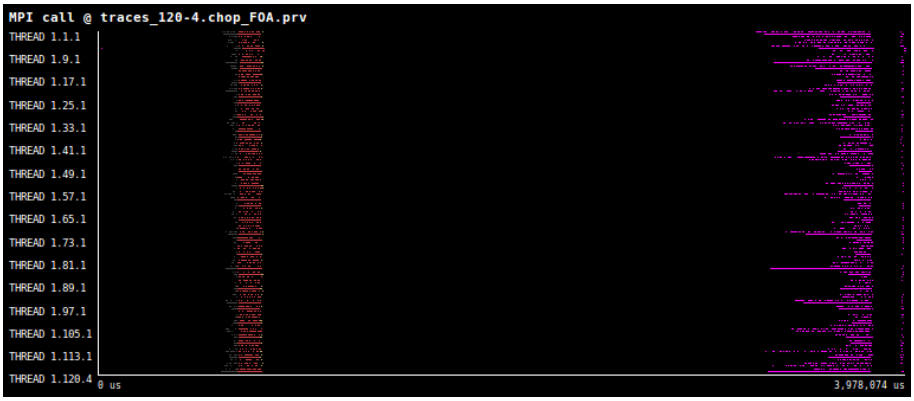


Figure 8. MPI communication patterns in the FOA.

9. Threading

The OpenMP parallelization is working very efficient. As far as the traces indicate, threads are spawned once and then run the entire time. In total, only 0.95% of the runtime are spent in the OpenMP runtime (fork/join or scheduling). Figure 9 highlights the main parallel regions that also coincide with the clusters above: *sparsematrix_mp_sequential_access_matrix_fast2* (cyan, Cluster 3 and Cluster 4), *sparsematrix_mp_sparsemm_new* (dark blue, Cluster 1), *setzer* (light green, Cluster 5), and *chebyshev_mp_chebyshev_clean* (red, Cluster 2). Whereas the first parallel region matches with the two clusters that do not scale very well.

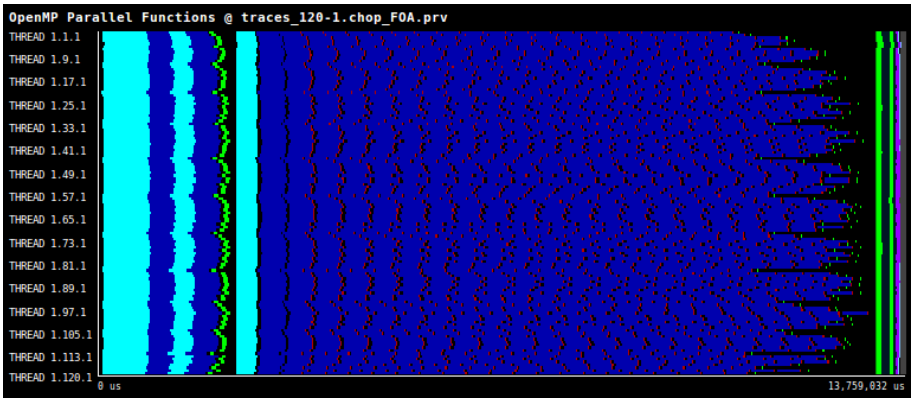


Figure 9. OpenMP parallel regions in the FOA.

10. Accelerators

This section does not apply for this audit.

11. I/O

This section does not apply for this audit.

12. Summary and Suggestions

In this audit we analysed the performance of a submodule of Application1 with a focus on scalability with increasing OpenMP threads. We analysed traces based on runs using 120 processes with one to eight threads per process. Thereby, the number of processes per node (two) remains constant. Overall, the application achieves a good parallel efficiency, workload distribution and efficiency in the OpenMP runtime. However, the overall scaling is only fair.

We found the main reasons for a declining overall efficiency are, first, the decreasing computing efficiency and, second, the non-scaling communication. In addition, load balance and computing efficiency in general might be considered for optimization.

- The primary obstacle for a better scaling is the fair computation scalability. While the workload is partitioned well and only marginally duplicated (instructions scalability), the computing scalability is only fair. This is most likely due to the saturation of the memory bandwidth or the memory controller since with increasing the number of threads per process since the resources are shared by more active cores per node.
- The second restriction in scaling is the decreasing communication efficiency. The efficiency decreases since the time spent in the first communication phase with one-sided communication does not decrease at all. This communication phase does not scale at all because communication volume and MPI process both remain constant.
- Additionally, the load balance (while quite high) might be further invested since it seems directly related to an increase in workload in the centre of the domain as well as the centre of each partition (double d-shape).
- Furthermore, the general computing efficiency (exempt scaling) is quite low and might be considered for improvement. The average IPC of 1.57 to 1.31 is only fair. This is mainly due to a high number of load/store instructions (about 25% - 40%).
- Finally, we highly recommend a further study that compares the results of this report (based on an increase of threads) with measurements that increase the number of MPI processes in total and per node to achieve comparable results. This way, we can evaluate which ratio of MPI processes to OpenMP threads delivers best results. A further study can be done either as a second POP Audit or a POP Performance Plan that includes a more detailed analysis and optimization guidance. In discussion with the user we decided to conduct this further study.