iPic3D performance assessment report

Document Information

Reference Number	POP_AR_85
Author	Michael Knobloch (JSC)
Contributor(s)	Brian Wylie (JSC), Ilya Zhukov (JSC)
Date	December 15, 2017

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.*

©2015 POP Consortium Partners. All rights reserved.

Contents

1 Background 3

2 Application structure and Focus of Analysis 3

3 Scalability 5

4 Load Balance 5

5 Serial performance 7

6 Communication and I/O 7

7 Efficiency 8

8 Summary of observations 9

1 Background

Applicants Name: Roman Iakymchuk
Application Name: iPic3D
Programming Language: C++
Programming Model: MPI + OpenMP, one version using MPI_THREAD_MULTIPLE and OpenMP tasks
Source Code Available: yes
Application Description: iPic3D is an implicit Particle-in-Cell code for Space Weather applications. iPIC3D simulates the interaction of Solar Wind and Solar Storms with the Earth's Magnetosphere. The magnetosphere is a large system with many complex physical processes, requiring realistic domain sizes and billions of computational particles. In the PIC model, plasma particles from the solar wind are mimicked by computational particles. At each computational cycle, the velocity and location of each particle are updated by solving the equation of motion, the current and charge density are interpolated to the mesh grid, and Maxwell's equations are solved.
Input Data: The input used was the "8-testMagnetosphere3Dsmall-particle.inp" data set, running on 8 processes with 8 OpenMP threads each.
Performance Study: Performance check (audit)
Machine Description: JURECA cluster at JSC (1872 compute nodes: 2xIntel Xeon E5-2680 v3 Haswell CPUs per node (12 cores, 2.5 GHz, Intel Hyperthreading Technology).
Used Environment: Intel compiler 2017, Intel MPI 2017, CMake, HDF5
Analysis Tools: Score-P, Cube[1], Scalasca, Vampir[2], PAPI

Due to the high measurement overhead of a fully instrumented run of iPic3D extensive compile-time filtering was necessary. In this case we filtered all routines that are not on a call-path to either MPI routines or OpenMP constructs.

A novelty in this audit is the comparison of two versions of iPic3D, the traditional one and a new version that uses OpenMP tasks in some kernels and implements a new communication scheme based on MPI_THREAD_MULTIPLE. Measurement of OpenMP tasks added significant memory requirements for the measurement system, so we were only able to measure at most 3 iterations. Trace analysis of this version is not possible as the tools used don't support MPI_THREAD_MULTIPLE (and thus no traces were collected at all). So this report focuses mainly on the traditional version and highlights the differences for the OpenMP kernels in section 2 and the new communication scheme in Section 6.

2 Application structure and Focus of Analysis

A graphical representation of the traditional application execution is shown in Figure 1. It clearly shows the typical structure of a scientific application: initialization ($\sim 7\ s$), 20 compute loop iterations, and finalization.

iPic3D is dominated by two computational kernels, the `ParticlesMover` consuming 63 % of the total runtime and `CalculateMoments`, which is responsible for 18 % of the runtime. In both kernels the majority of time is spent in OpenMP parallel for loops. The OpenMP for loop in the `ParticlesMover` kernel is in the `mover_PC_AoS` subroutine and consumes 75 % of the kernel runtime and 47 % of the total runtime. The `sumMoments_AoS` function in the

[1] https://pop-coe.eu/sites/default/files/pop_files/cube_display_quickref.pdf
[2] https://pop-coe.eu/sites/default/files/pop_files/vampir_display_quickref.pdf

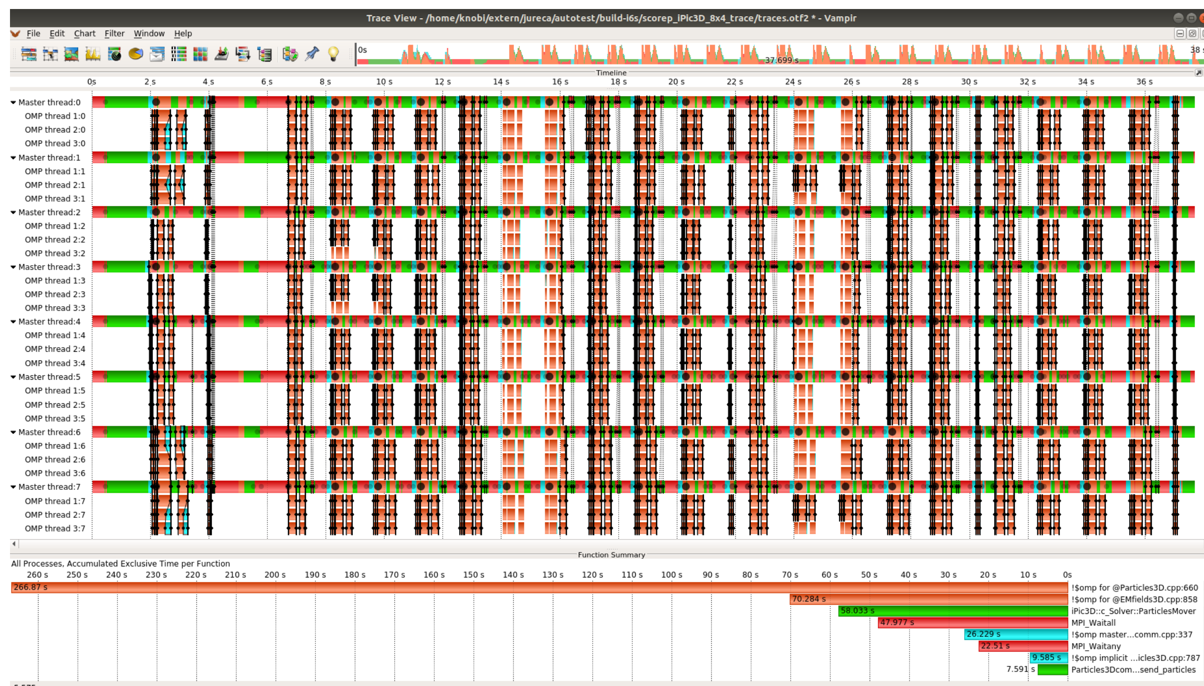

Figure 1: Execution timeline of iPic3D with 20 iterations on JURECA. Timeline chart of the 8 processes at top, and function summary chart at bottom.

`CalculateMoments` kernel contains an OpenMP for loop that consumes 68 % of the kernel runtime and 12 % of the total runtime. While the `ParticlesMover` routine itself takes significant time (10 % of the total runtime), the `CalculateMoments` contains an OpenMP master region (in the `pad_particle_capacities` function) which requires about a quarter of the runtime of this kernel (or 4.4 % of the total runime). The main MPI communication is done in `CalculateB` and the `recommunicate_particles_until_done` routine in the `ParticlesMover` kernel. MPI communication is responsible for 12.6 % of the runtime. At the end of each iteration the output is written via MPI I/O in the `WriteOutput` routine, which requires 6.7 % of the total runtime. We focus the analysis on these parts of the code.

Figure 2 shows one iteration of iPic3D in detail. While the function summary shows the same profile, we already see some imbalances in the OpenMP regions (in orange) and especially in the `ParticlesMover` kernel on the right half of the picture. We discuss these load balance issues in detail in Section 4.

In the new version with OpenMP tasks, the tasks are used as outlined in Listing 1. A task is generated for each iteration of an already parallelized loop. This might be beneficial if the runtime of different iterations varies significantly, as in that case the distribution of loop iterations on the hardware threads might be sub-optimal. For this test case at least the usage of OpenMP tasks decreased performance significantly. We tried 2 alternative versions of the main kernel, the OpenMP loop of the `ParticlesMover` for comparison– one with only the parallel for loop (no tasks) and one with only tasks (no OpenMP for). The fastest version was the plain OpenMP for version with a runtime of 27.4 s, followed by the one with parallel loop and tasks with a runtime of 31.3 s. The version using only OpenMP tasks performed poorly with a total runtime of 98 s. There are more kernels where OpenMP tasks are used in the same way, removing those would probably be even more beneficial.

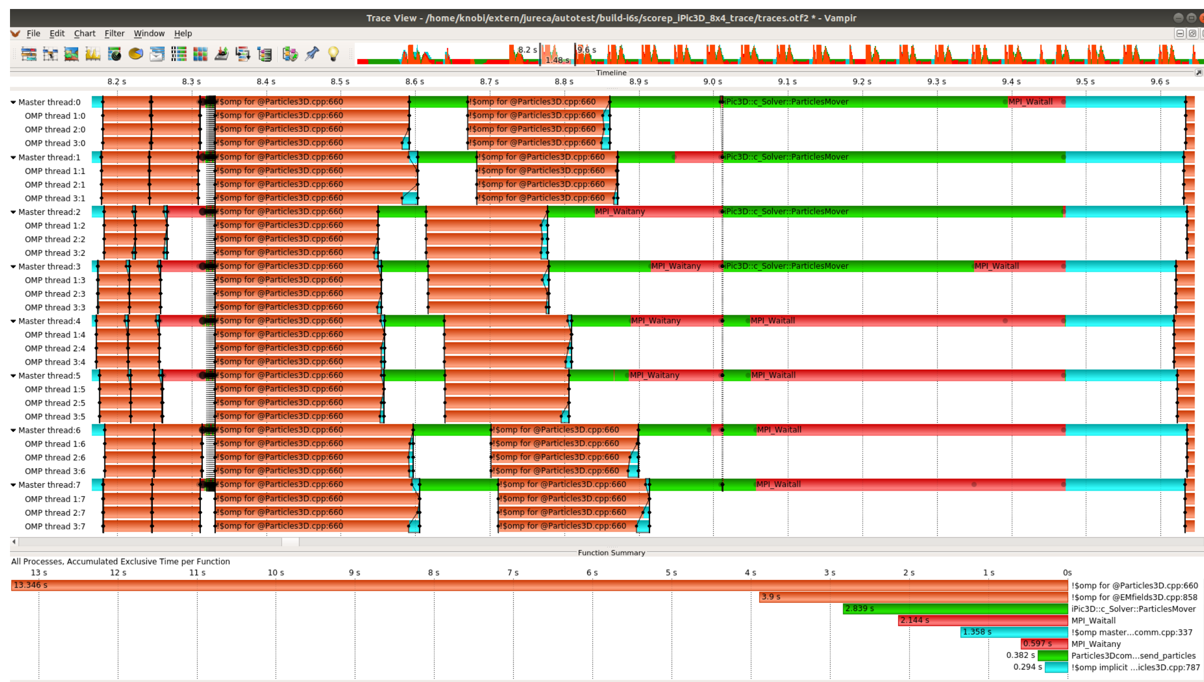

Figure 2: Execution timeline of iPic3D for the 2nd iteration on JURECA. Timeline chart of the 8 processes at top, and function summary chart at bottom.

```
#pragma omp parallel
#pragma omp for
for (...) {
#pragma omp task
{
/*loop iteration*/
}
}
```

Listing 1: Structure of OpenMP task usage in iPic3D

3 Scalability

A scalability analysis was not part of this audit, but should be performed in another POP service (see section 8).

4 Load Balance

The iPic3D analysis showed two significant load balance issues in the two main computational kernels. In the main kernel, the `ParticlesMover`, the OpenMP part is quite balanced with about 20 % variation. The rest of the routine (which has a similar aggregated runtime (including idle threads) to the OpenMP part) is very badly balanced. The first four processes run 3 times as long as the other four processes, see Figure 3. This in turn leads to a high MPI waiting time on those processes, see Section 6.

The other issue occurs in the OpenMP parallel for loop in the `CalculateMoments` kernel. As Figure 4 shows, the first and the last two processes take 40 % more time than the 4 processes

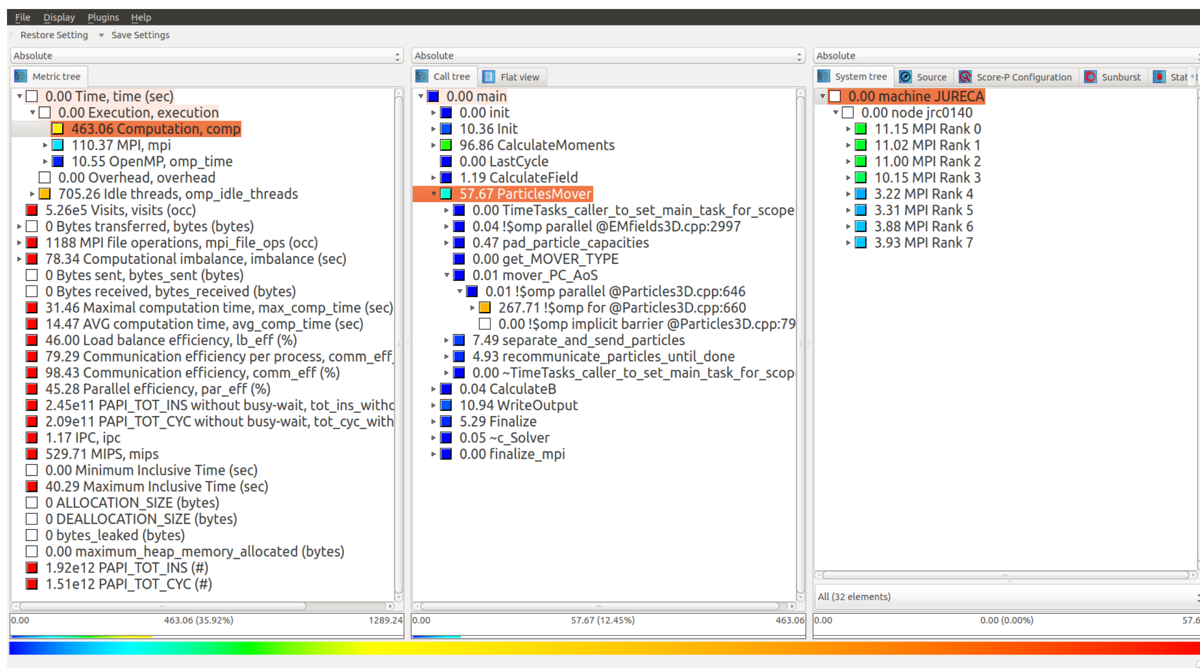

Figure 3: Cube screenshot showing the load balance issue in the `ParticlesMover` routine. Processes 0 to 3 take significantly longer than processes 4 - 7.

in between.

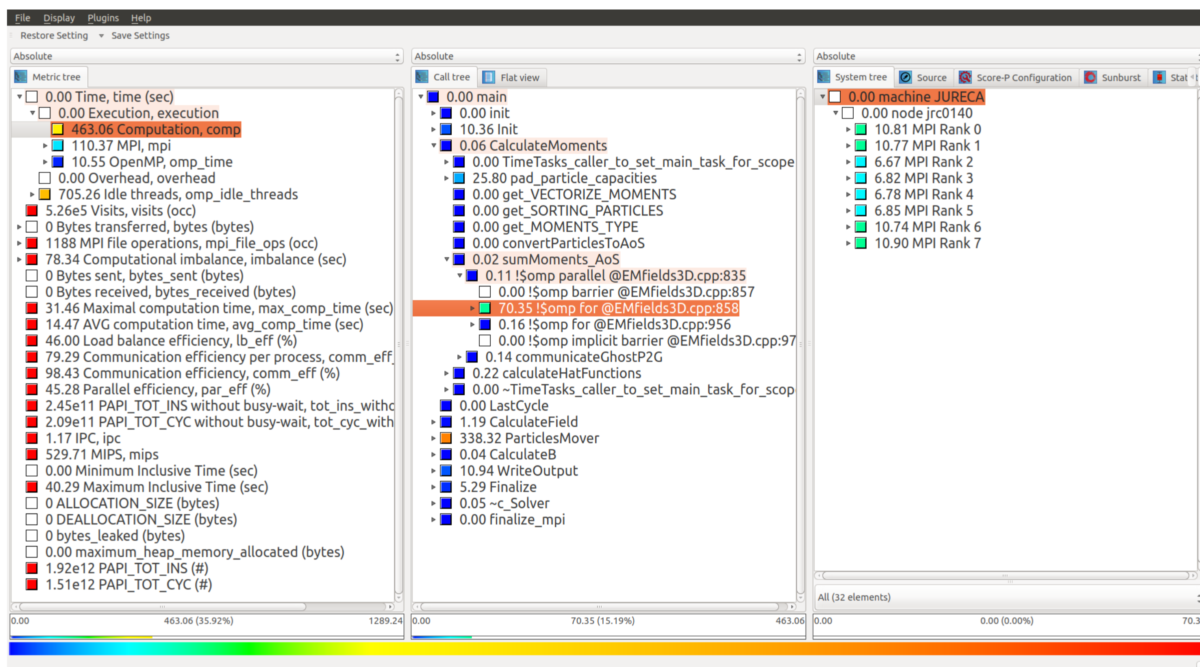

Figure 4: Cube screenshot showing the load balance issue in the `CalculateMoments` kernel. Processes 0+1 and 6+7 take significantly longer than processes 2 - 5.

5 Serial performance

In the regarded execution of iPic3D, two hardware counters were measured to investigate serial computational performance:

- `PAPI_TOT_INS`: Total instructions completed
- `PAPI_TOT_CYC`: Total processor cycles

A common metric to determine the efficient usage of compute resources is IPC, the instructions per cycle. We can easily calculate that as `PAPI_TOT_INS/PAPI_TOT_CYC`. Figure 5 shows a profile augmented with the IPC metric.

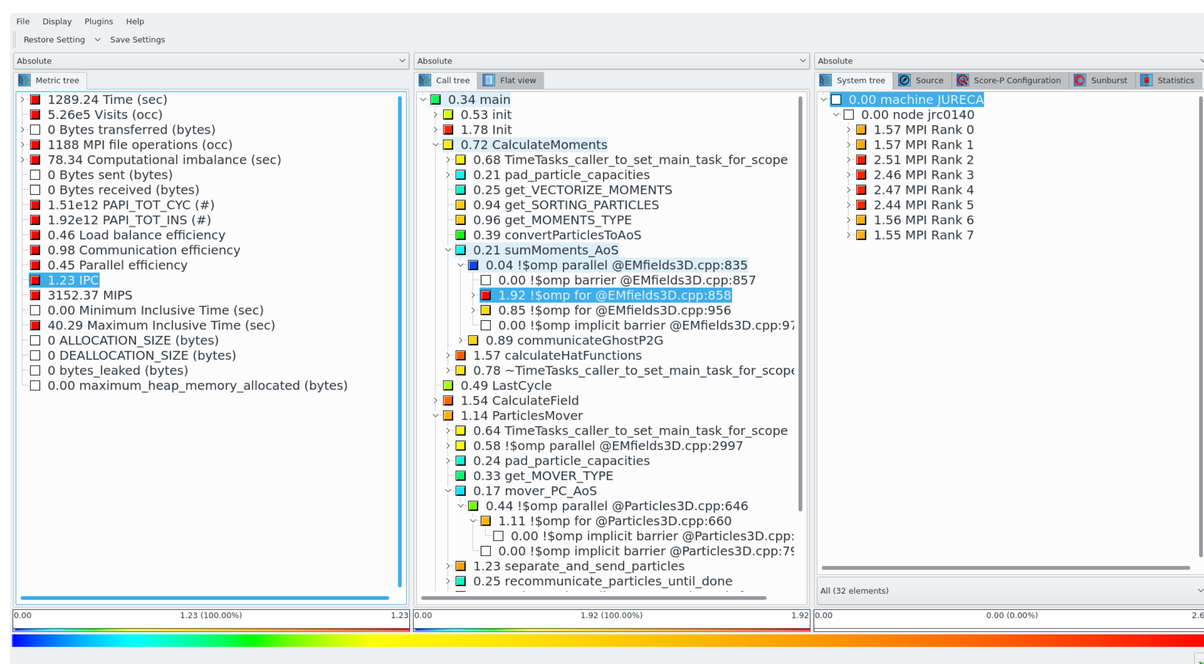

Figure 5: Profile report of iPic3D enhanced with the IPC metric. It shows a discrepancy in the main OpenMP loop of the `CalculateMoments` kernel where four processes have a significantly higher IPC ratio than the other four processes.

Looking at the four most computational intensive parts we see IPC ratio of 1.1 for both the `ParticlesMover` and the embodied OpenMP parallel for loop and a poor IPC ratio of 0.2 for the OpenMP master region in the `CalculateMoments` kernel. The most interesting observation is for the OpenMP loop in the `CalculateMoments` kernel which shows a load balance issue (see Section 4). The processes that take less time (processes 2-5) have a significantly higher IPC ratio of 2.5 than the other four processes, which have an IPC ratio of 1.5. This discrepancy should be further investigated.

6 Communication and I/O

MPI communication in iPic3D is mainly non-blocking point-to-point communication. The proportion of time spent in MPI communication calls varies from 4 % on rank 0 to 20 % on rank 5. Nearly all of the communication time is waiting time in `MPI_Waitall` and `MPI_Waitany`.

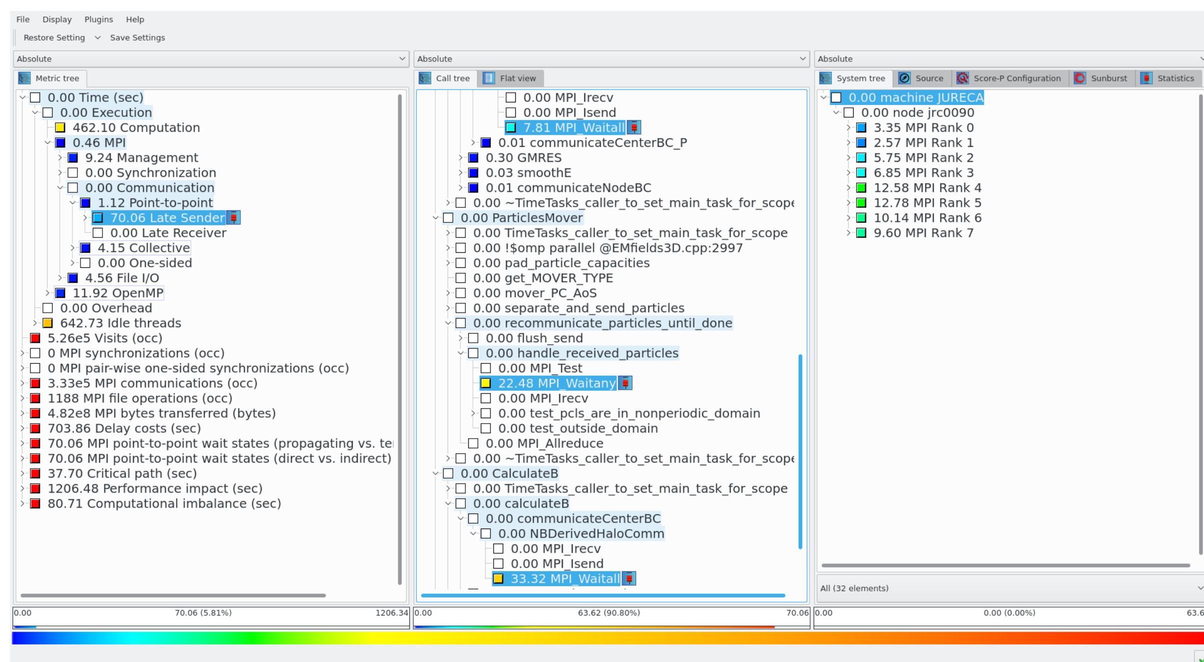

Figure 6: Cube screenshot showing the distribution of Late Sender time.

The *Late Sender*[3] metric is one of the performance metrics determined by the Scalasca trace analyzer. It quantifies the time a process receiving a message is waiting for the sending process to finish its work and start sending the message. Figure 6 shows the distribution of the *Late Sender* metric across the processes. As Figure 2 already shows we see a higher waiting time on processes 4 - 7, resulting from the load imbalance in the `ParticlesMover` routine (see Section 4).

It is difficult to evaluate the effects of the new communication scheme based on `MPI_THREAD_MULTIPLE`. As in the original case, most time is spent in the various `MPI_Wait` routines and changes in runtime and load-balancing naturally influence these values. It should be noted that this new scheme only supports up to 8 threads in powers of two, i.e. one, two, four, and eight threads. Due to the complexity of the implementation it is very difficult to increase that number.

iPic3D uses MPI I/O as a parallel I/O paradigm. The `WriteOutput` is responsible for about 6.7 % of the total runtime in this testcase. However, 66 % of that time is spent in `MPI_File_set_view`, which should become less significant for larger, more realistic test-cases.

7 Efficiency

In the course of the POP audits we defined a set of so-called efficiency metrics to characterize the application behavior[4]. These metrics' values are always between 0 and 100% (or, equivalently, a value between 0 and 1), the higher the number the better. *Load balance* is the ratio of average computation time to maximal computation time. *Communication efficiency* is the ratio of maximal computation to maximal execution time. *Parallel efficiency* is the ratio of average

[3] https://apps.fz-juelich.de/scalasca/releases/scalasca/2.3/help/scalasca_patterns-2.3.html#mpi_latesender

[4] https://pop-coe.eu/node/69

computation time to the maximal executing time, which is also the product of *Load balance* and *Communication efficiency*.

While iPic3D shows a very good Communication efficiency of 98 %, the Load balance efficiency of 46 % is poor, resulting in a Parallel efficiency of just 45 %.

It has to be noted that the calculation of the efficiency metrics is currently imperfect for hybrid MPI/OpenMP codes. However, they still give a reasonable foundation for further investigation and subsequent comparison.

8 Summary of observations

The audit of this relatively small testcase of iPic3D on JURECA showed two major load balance issues in the two main computational kernels. We recommend to continue the analysis process and focus on the following points in subsequent performance audits or performance plans:

- Investigate load-balance issues in `ParticlesMover` and `CalculateMoments`.
- Investigate discrepancy in IPC ratio in `CalculateMoments`.
- Scalability analysis to investigate iPic3D scaling on a larger number of MPI processes and OpenMP threads
  - Weak scaling, i.e. increase input data with number of processes
  - Strong scaling, i.e. keep input data constant
- Investigate serial performance of computational kernels in more detail, e.g. with a thorough hardware counter analysis.
- Investigate influence of I/O on larger problems.