CAMP CPU profiling

Victor Correal, Xavier Yepes-Arbós

August 2024

Contents

1 Introduction & Methodology

This document contains the performance analysis of the CAMP[1] CPU dwarf application. CAMP is a novel framework permitting run-time configuration of chemical mechanisms for mixed gas- and aerosol-phase chemical systems. This application is based on time-stepping, meaning that the same computation is repeated one after the other until a certain number of time-steps is achieved according to the forecast length requested in the simulation.

This CPU profiling aims to be used as a reference for the already implemented GPU version of CAMP[1] and in a future heterogeneous CPU-GPU implementation. The main goal is to check the computation performance of this application in terms of IPC efficiency, cache memory efficiency, and vectorization. This profiling analysis is conducted in the CPU partition of the pre-operational Marenostrum 5 supercomputer of BSC, Table 1 contains the hardware specification of this machine.

	Marenosturm 5
CPU model	2x Intel Xeon Platinum 8480+ with 112 cores (224 threads with hyperthreading) running max. at 3.80 GHz
NUMA nodes	2 (56 cores per NUMA node)
Memory per node	256 GiB
L1d Cache	5.3 MiB (112 instances)
L2 Cache	224 MiB (112 instances)
L3 Cache	210 MiB (2 instances)
Filesystem	GPFS Parallel filesystem 960GB NVMe local storage
Network	ConnectX-7 NDR200 InfiniBand (shared by two nodes, 100Gb/s bandwidth per node)

Table 1: Marenostrum 5 CPU partition specifications

We will use Extrae and Paraver, from the BSC toolset[2], as the main tools to develop this performance analysis. Extrae is a library that intercepts different parallel programming model calls and generates trace files with events that can be displayed afterward. After generating the trace, we can open them with Paraver. Fig. 1 shows an overview of the workflow of these two tools.

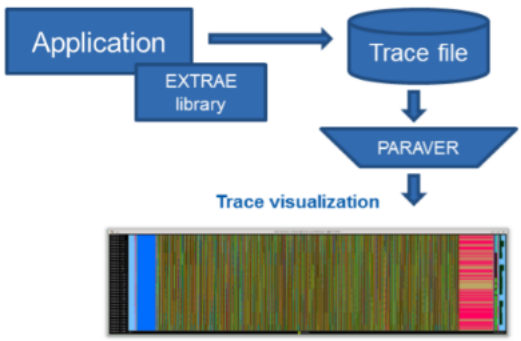

Figure 1: Overview of Extrae and Paraver (BSC Tools)

The CAMP dwarf is parallelized using a distributed-memory approach using the MPI library, without any multithreading shared-memory approach. It is completely parallel, which means that the domain is decomposed at the beginning of the execution into many subdomains, called cells, and all the computation happens without any dependencies, synchronization, or communication between tasks.

[1] Chemistry Across Multiple Phases
[2] https://tools.bsc.es/

2 Profiling results

The following sub-sections will cover a strong scaling study and the performance analysis using Extrae traces with performance counters. This sections will evaluate the computational performance of this dwarf. We will use one node of the CPU partition of the Maresnotrum 5 supercomputer.

2.1 Scalability study

For this strong scalability study, we will measure the elapsed times in the computation phase of the dwarf with the BSC tools. In this time-step-based application, the initialization phase must be discarded from the performance evaluation, as the model usually runs for a lot of timesteps, making this initialization time negligible.

The dwarf was compiled using the Intel compiler, with the third level of optimizations (`-O3`) and the Intel MPI library. We used one node of Marenostrum 5, increasing the number of nodes in 5 steps (10, 30, 60, 80 and 112 tasks) and set the simulation parameters to 100,000 cells and 5 timesteps. Fig. 2 shows the scalability plot alongside with the initialization time in the right vertical axis.

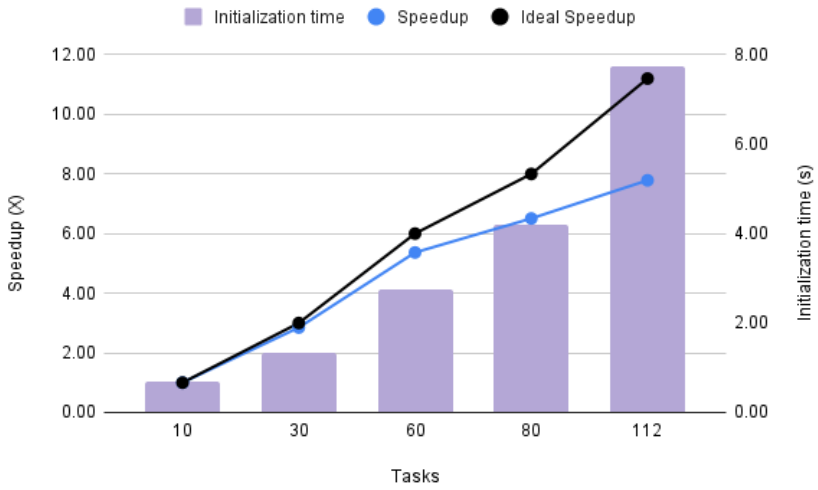

Figure 2: Strong scaling of the computation part, discarding initialization. The right axis shows the initialization time in seconds.

The scalability curve looks acceptable, despite the two last executions with 80 and 112 MPI tasks being far from the ideal scalability line. This lack of strong scalability can appear due to different computational factors but this also can appear if the problem size is too small for this amount of tasks.

Following the scalability analysis, we used the abstract performance model from the BSC tools, which reports different efficiency metrics based on Extrae traces, that help to understand the overall application performance. Section 4.1 contains a brief explanation of this performance model and the metrics. Fig. 3 shows the results for the CAMP dwarf with the same MPI tasks as the strong scaling curve of Fig. 2 (i.e. 10, 30, 60, 80 and 112 MPI tasks).

Figure 3: Results of the performance model for executions with 10, 30, 60, 80 and 112 MPI tasks

We can notice that the global efficiency drops because of a lack of frequency and IPC scalability in the computation scalability metrics. The other metrics are good quite good. Following Section 2.2 will analyze in depth the computational performance to understand the reason for this lack of scalability reported by the performance abstract model.

2.2 Performance analysis

For this analysis, as explained in the Introduction (Section 1), we used the BSC tools to get traces of the execution of CAMP with Extrae and analyze them with Paraver. The simulation parameters for all the executions are set to 100,000 cells and 5 time-steps, which are the same as the strong scaling analysis. To get the full potential of the BSC tools, we need to recompile the dwarf with debug symbols (`-g`) and instrument the subroutines calls (`-finstrument-functions -rdynamic`).

The first step in the performance analysis was to recognize the application structure in terms of communication and top-consuming user functions, to center the subsequent analysis with performance counters.

2.2.1 MPI Communication

The MPI communication involved is very simple because this application is fully parallel, without any communication in the middle of the computation. Fig. 4 shows a Paraver timeline with the MPI calls and two zooms in the initialization and finalization phases. This timeline represents MPI calls in different colors for each task (y-axis) at each moment (x-axis).

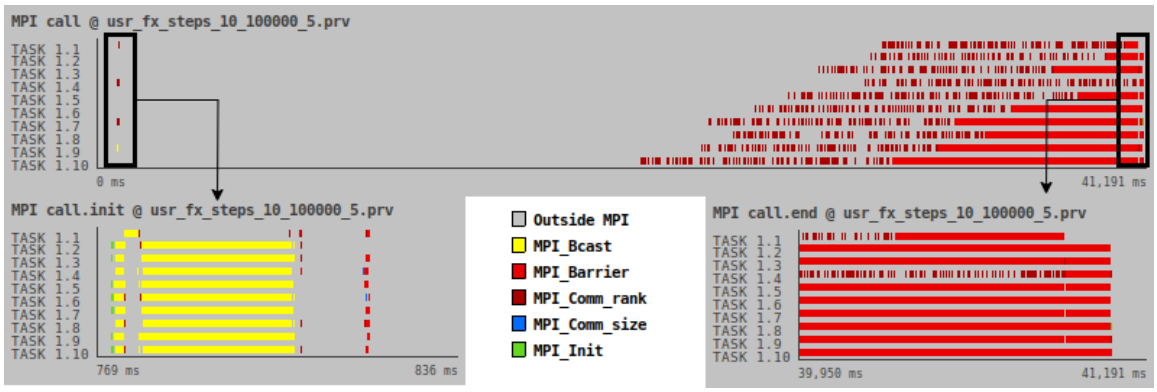

Figure 4: MPI communications involved in CAMP dwarf. 100,000 cells execution and 5 timesteps. Two additional timelines are zooming the initialization and finalization phases.

We can distinguish the initialization phase just before the first MPI communication (bottom-left zoom), then the subdomain decomposition with MPI broadcast calls followed by a long computation phase (top), and then the finalization with an MPI barrier(bottom-right zoom). As expected, all the computation happens without any MPI communication.

It is noticeable that the MPI_Comm_rank is called multiple times in the execution, accounting for a total of 10,012 times per MPI task. The impact on the execution time is negligible but these unnecessary MPI calls, depending on the case, can prevent the compiler from inserting some optimizations and have a very easy solution to avoid (just store the task ID in a variable).

2.2.2 User functions

The CAMP dwarf iterates over the request timesteps calling the *integrate* subroutine. So, our first approach was to trace the *integrate* user function to check the time of each timestep. Fig. 5 shows a Paraver timeline with 10 tasks and 5 timesteps, we can see for each task (y-axis) and each moment (x-axis) of the execution the start and finish of each timestep, marked with the green flag.

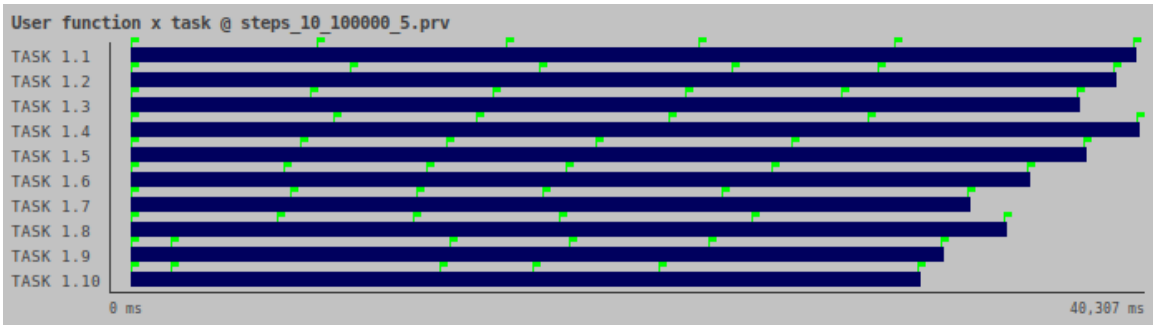

Figure 5: CAMP dwarf timestep overview, marked by the *integrate* subroutine. The start/finish of each timestep is marked with a small green flag at each task.

We can see that the timestep times are irregular as not all the steps took the same time. For example, tasks 9 and 10 have a small first step compared with the others but the second step is longer than the second and third steps of the other tasks. The workload of the computations seems irregular as well, as each task took a different time to complete all the computations; this can be an effect of the irregularity of the timesteps or a bad workload distribution.

Next, we increase the user functions to trace with Extrae. The goal is to identify the most time-consuming subroutines and get performance information for each subroutine later. Fig. 6 shows the timeline of the six most-consuming user functions for a 10 MPI tasks execution and 5 timesteps. In this timeline view, the color code represents each user function for each MPI task (y-axis) at each moment (x-axis).

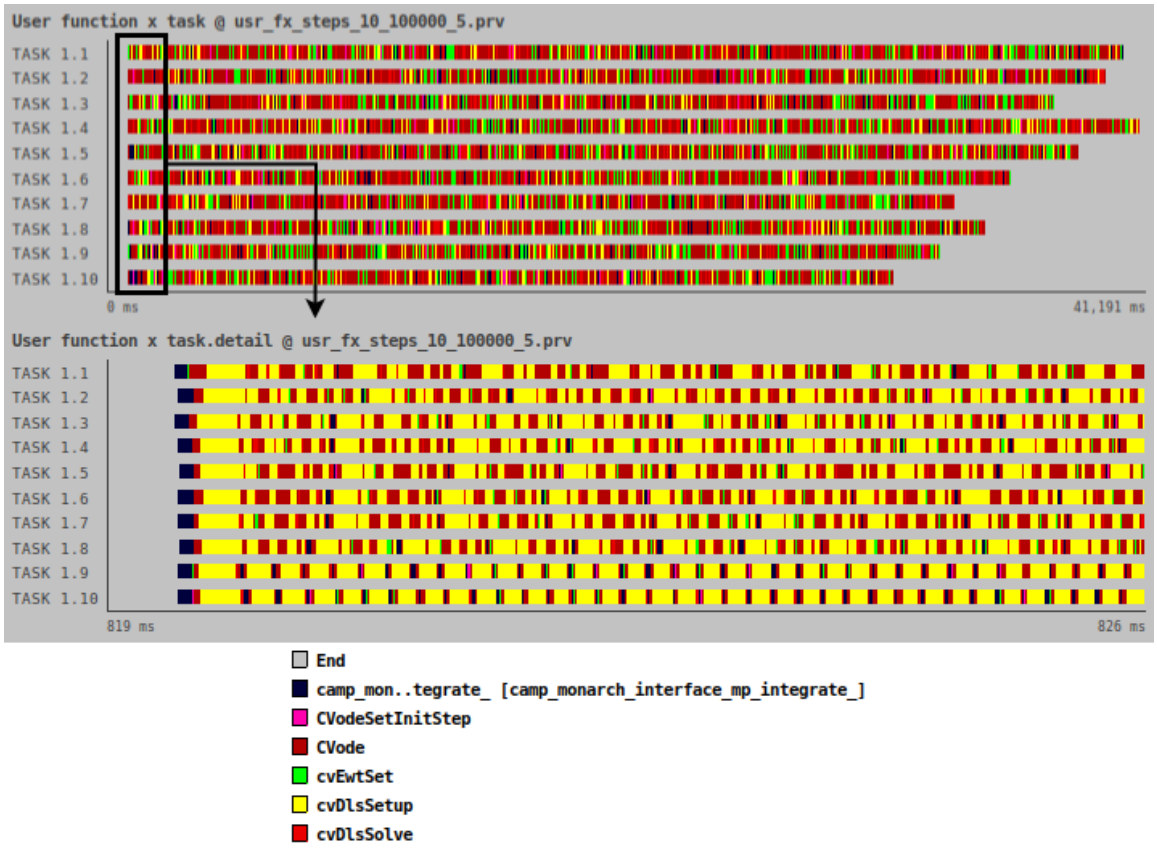

Figure 6: Overview of the most time-consuming user functions (different colors) of CAMP for each task (y-axis) at each moment (x-axis). The bottom part is a detailed view of the first 5 milliseconds of execution.

To get insights into the user functions time usage, Paraver can generate histograms from the timeline views and generate statistics. Fig. 7 shows the percentage of time spent in each of the subroutines from the trace of Fig. 6 for each MPI task and the statistic aggregation at the bottom. The histogram also contains a gradient scale representing as blue the maximum values and light-green the minimums.

User functions histogram @ usr_fx_steps_10_100000_5.prv

	CVode	cvDlsSetup	End	camp_mon..tegrate_	cvDlsSolve	cvEwtSet	CVodeSetInitStep
TASK 1.1	45.30 %	40.78 %	4.18 %	5.72 %	3.07 %	0.95 %	0.01 %
TASK 1.2	42.67 %	41.93 %	5.91 %	5.41 %	3.12 %	0.95 %	0.01 %
TASK 1.3	40.95 %	39.04 %	10.87 %	5.27 %	2.96 %	0.91 %	0.01 %
TASK 1.4	44.00 %	43.72 %	2.70 %	5.42 %	3.20 %	0.95 %	0.01 %
TASK 1.5	43.19 %	38.73 %	8.62 %	5.69 %	2.86 %	0.89 %	0.01 %
TASK 1.6	40.58 %	35.53 %	15.01 %	5.46 %	2.60 %	0.81 %	0.01 %
TASK 1.7	36.12 %	34.72 %	20.45 %	5.42 %	2.51 %	0.77 %	0.01 %
TASK 1.8	37.92 %	35.75 %	17.50 %	5.53 %	2.53 %	0.76 %	0.01 %
TASK 1.9	37.75 %	31.75 %	21.95 %	5.49 %	2.31 %	0.73 %	0.01 %
TASK 1.10	34.71 %	30.76 %	26.36 %	5.34 %	2.15 %	0.67 %	0.01 %
Total	403.20 %	372.71 %	133.57 %	54.76 %	27.31 %	8.39 %	0.07 %
Average	40.32 %	37.27 %	13.36 %	5.48 %	2.73 %	0.84 %	0.01 %
Maximum	45.30 %	43.72 %	26.36 %	5.72 %	3.20 %	0.95 %	0.01 %
Minimum	34.71 %	30.76 %	2.70 %	5.27 %	2.15 %	0.67 %	0.01 %
StDev	3.38 %	4.07 %	7.71 %	0.13 %	0.34 %	0.10 %	0.00 %
Avg/Max	0.89	0.85	0.51	0.96	0.85	0.88	0.88

Figure 7: User functions histogram using the information from Fig. 6 and computing the percentage of time of each subroutine. The end represents everything that is not the user function.

Near 80% of the time is spent in the *CVode* and *cvDlsSetup* subroutines. So the subsequent sections need to ensure that the performance in these two subroutines is optimal in terms of computational efficiency. We also noted that the number of calls to *cvDlsSetup*, *cvDlsSolve*, and the *CVode* is different depending on the MPI task. This difference in the number of subroutines calls points to a convergent algorithm, which explains why some tasks have more calls to this user function than others and explains the imbalance shown on Fig. 5.

2.2.3 Cycles & Instructions

Following the performance analysis now we analyze the IPC of the dwarf, which is a good indicator of the overall performance of an application. We configured Extrae to capture the PAPI performance counters for instructions (`PAPI_TOT_INS`) and cycles (`PAPI_TOT_CYC`). We have discarded, using Paraver, the IPC metric in the non-computation phases of the application (like communication or synchronization); we call this useful IPC.

Fig. 8 shows the useful IPC metric for 112 MPI task executions at each moment, the IPC metric is drawn using a color gradient (from light green to blue). The average value is around 4.00, which is a good IPC metric for any application. We can also see some very small green values in the middle of the executions. Other traces with other task counts have the same value of IPC but are not displayed for simplicity.

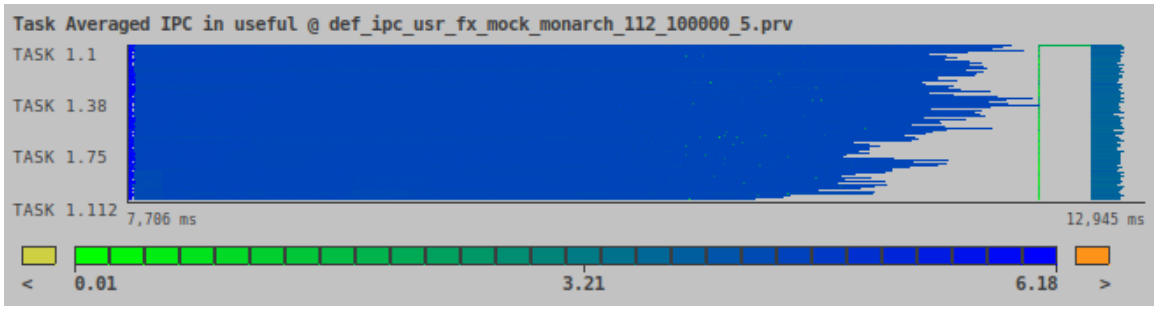

Figure 8: Useful IPC timeline of the computation and finalization phase. 112 MPI tasks executions with 100,000 cells and 5 timesteps. IPC is represented with a color gradient from light-green to blue.

With Paraver, we quantified the average IPC values for the top-consuming user functions. For *CVode* the average value across all the MPI tasks is 3.80 and for *cvDlsSetup* is 3.73 which is a very good IPC figure.

As shown on Fig. 5, we detected a load imbalance issue, as the finalization of the computation phase is different for each MPI task. With Paraver we have generated a histogram with the useful instructions per MPI task, as this load imbalance issue usually comes from an imbalance in the number of completed instructions. Fig. 9 contains the histogram for 112 MPI tasks, obtained from the same trace as Fig. 8. The histogram uses a color scale (from the lowest value as light green to blue as the highest value) to represent the number of instructions for each burst of the histogram (x-axis).

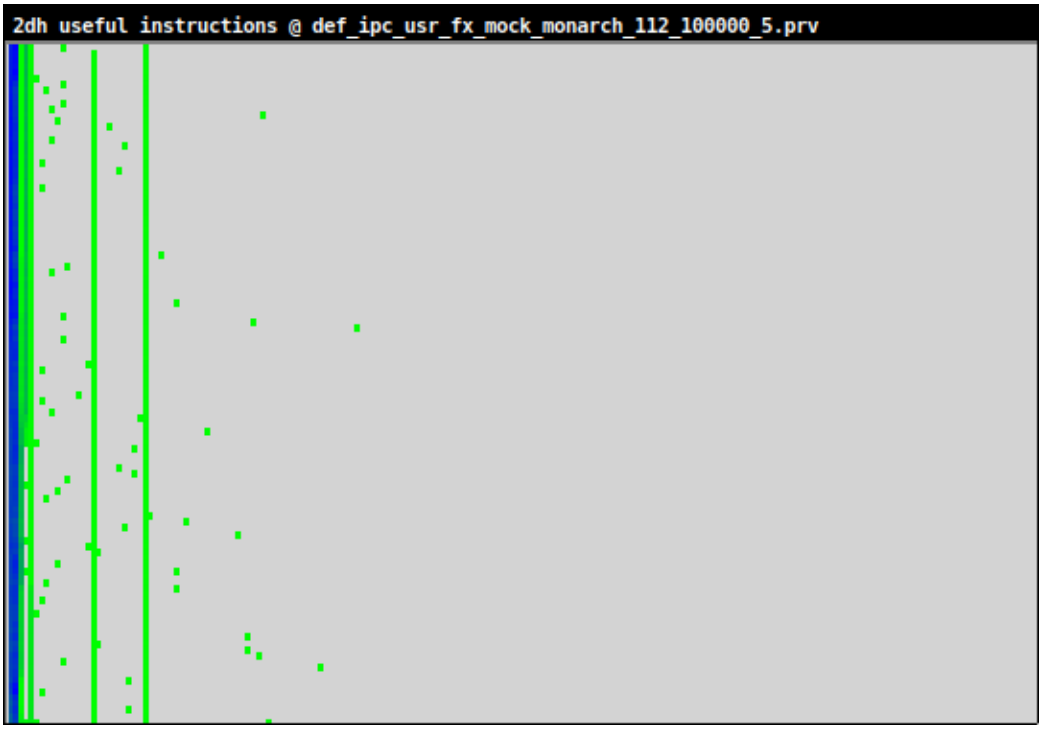

Figure 9: Histogram of useful instructions per MPI task for 10, 80 and 112 execution.

At the histogram, we can see that different tasks execute different numbers of useful instructions in some cases but there are also a lot of equally distributed instructions, as we can see three completely vertical lines. To quantify this small imbalance we used Paraver to find the absolute number of useful instructions completed by each user function, shown on Fig. 10. The average divided by the Maximum (Avg/Max) value measures the degree of imbalance, being 1 a completely balanced metric.

2DH - Useful instructions - User functions @ def_ipc_usr_fx_mock_monarch_112_100000_5.prv

	CVode	cvDlsSetup	cvDlsSolve	cvEwtSet	CVodeSetInitStep
Total	2,134,936,006,406	1,802,428,791,456	102,577,088,360	37,086,575,509	349,822,604
Average	19,061,928,628.62	16,093,114,209.43	915,866,860.36	331,130,138.47	3,123,416.11
Maximum	22,693,247,765	19,748,468,808	1,086,404,084	388,815,016	3,269,534
Minimum	15,001,670,083	11,961,084,366	620,673,312	231,460,330	3,075,061
StDev	1,829,955,057.95	1,936,709,007.68	118,167,866.61	40,500,370.74	27,587.50
Avg/Max	0.84	0.81	0.84	0.85	0.96

Figure 10: Useful instructions completed by each user function with statistics. The degree of imbalance (Avg/Max).

2.2.4 Data access operations

To check the performance of the memory cache hierarchy we are going to follow the same approach as Section 2.2.3. It is noticeable, that the CPU nodes of Marenostrum 5 have a shared last level of cache cache among the cores of the same NUMA node (Fig. 11).

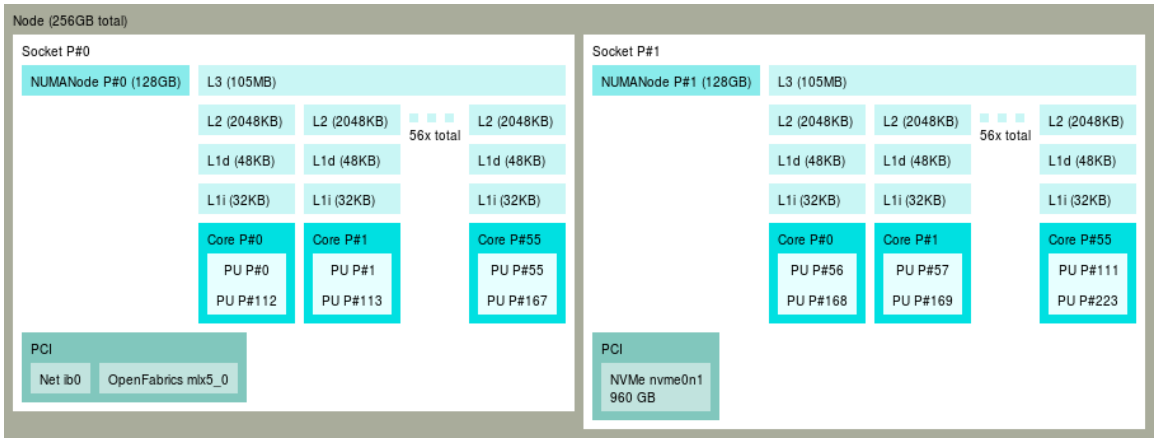

Figure 11: MN5 memory topology of one node in the CPU partition

We execute the dwarf with the same setup (i.e. 100,000 cells and 5 timesteps with 10, 30, 60, 80 and 112 MPI tasks) using Extrae to capture the data cache misses for first and second level (`PAPI_L1_DCM`, `PAPI_L2_DCM`) and the total cache misses for the last-level (`PAPI_L3_TCM`). We will also collect the number of instructions (`PAPI_TOT_INS`) to compute the cache miss ratio per 1,000 instructions.

Fig. 12 show the average cache misses per 1,000 useful instructions in the computation phase using 112 MPI task for the three cache levels. We can see that at all cache levels, the miss ratio is very low in general, the highest value is around 70 chance misses per 1,000 useful instructions for the first level of cache.

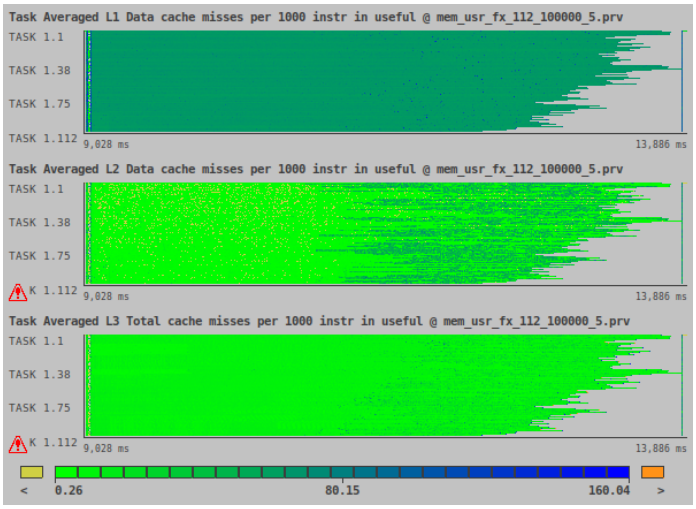

Figure 12: Timeline for cache misses per 1,000 useful instructions.
Execution with 112 MPI tasks, 10,000 cells and 5 timesteps

To complete the study, we extract the cache misses per 1,000 useful instructions for each of the most time-consuming user functions. Fig. 13 contains several plots for the three cache levels, we can see the cache miss ratio is very good as well for the most time-consuming user functions.

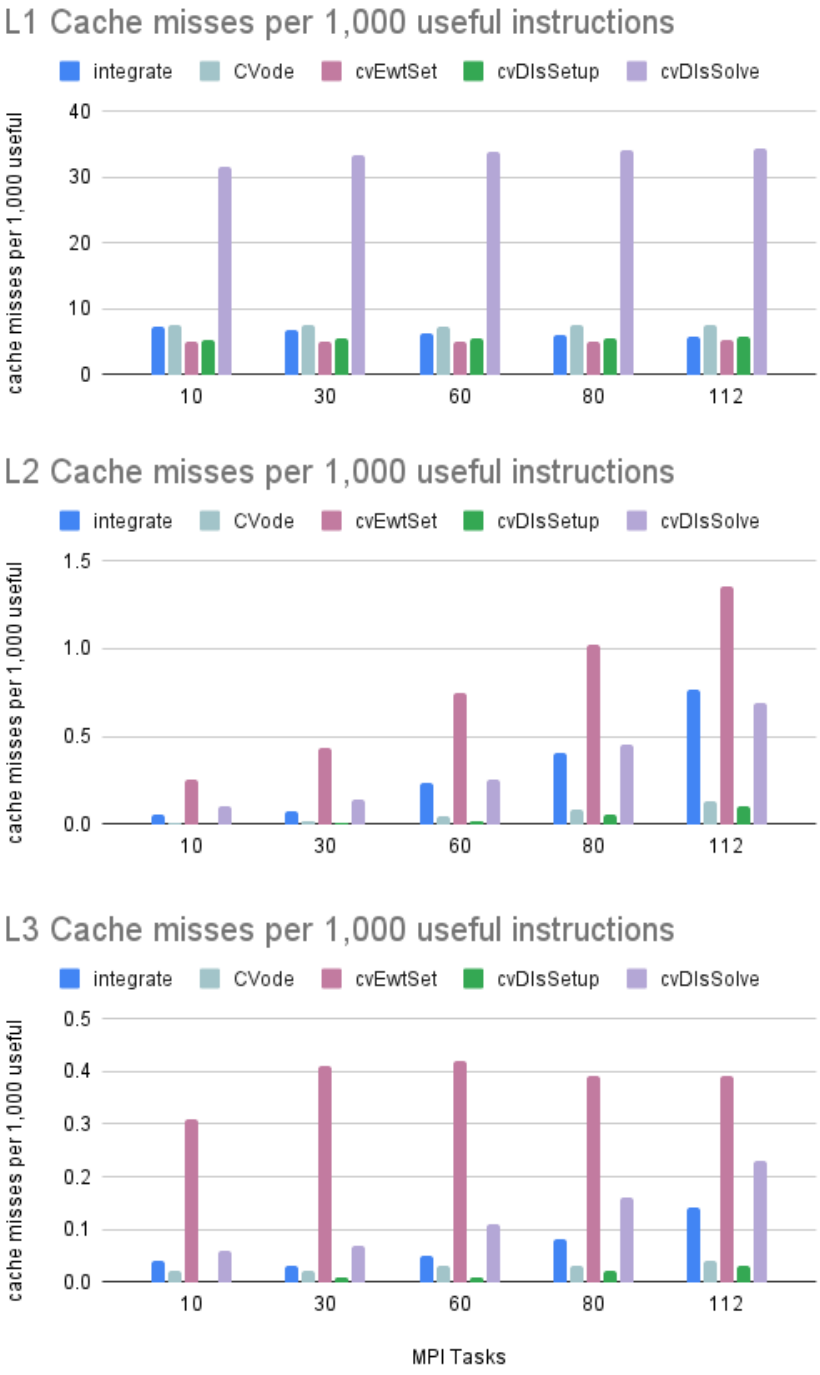

Figure 13: Cache misses per 1,000 useful instructions for L1, L2 and L3 caches. Execution with 112, 80, 60, 30 and 10 MPI tasks and 100,000 cells and 5 timesteps.

2.2.5 Vectorization

Vector instructions, also known as Single Instruction Multiple Data (SIMD) instructions, leverage processor hardware to perform multiple operations in parallel within a single instruction. To assess the effectiveness of this code vectorization, we can use the hardware counters for floating-point operations and vector instructions, which accumulate the number of floating-point operations and the number of vector instructions performed respectively. The ratio between these two hardware counters quantifies how vectorized the code is. The Marenostrum 5 CPU partition has a vectorial unit of 512-bit length[2], meaning that each SIMD instruction can operate simultaneously 8 double-precision and 16 single-precision floating point operations. If the compiler well vectorizes the application, the ratio needs to be close to the number of simultaneous operations of this processor.

For the single precision vectorization, we get an Extrae trace with `PAPI_SP_OPS` and `PAPI_VEC_SP` hardware counters. Section 2.2.5 shows that only at the initial stage of the computation there are some single-precision SIMD operations. The timeline shows the ratio between operations and single-precision SIMD instructions per task (y-axis) at each moment.

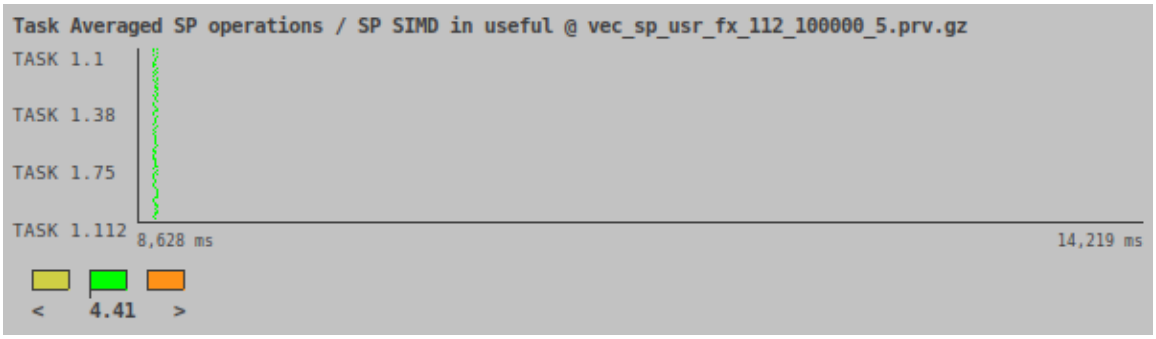

Figure 14: Ratio between single-precision floating point operations and single-precision SIMD instructions in the computation phase, execution with 112 MPI tasks, 100,000 cells and 5 time-steps.

We checked if there were any operations with no vectorization at all. To do that, we generated a timeline displaying the value of the `PAPI_SP_OPS` counter at each moment. Fig. 15 show the single-precisions floating point operations. We can see that in the whole execution, there are operations that are not vectorized at all.

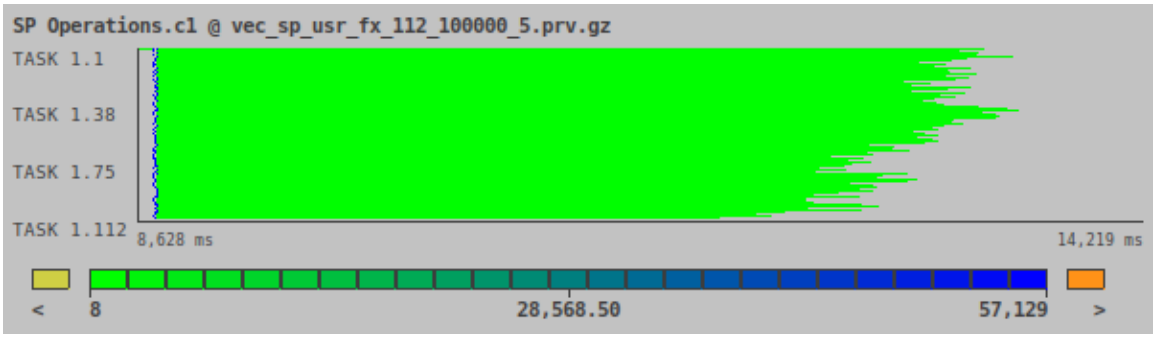

Figure 15: Single-precision floating point operations in the computation phase, execution with 112 MPI tasks, 100,000 cells and 5 time-steps.

Regarding the double-precision vectorization, we followed the same methodology as the single-precision case using `PAPI_DP_OPS` and `PAPI_VEC_DP` hardware counters. Fig. 16 shows the ratio between double-precision operations and SIMD double-precision instructions.

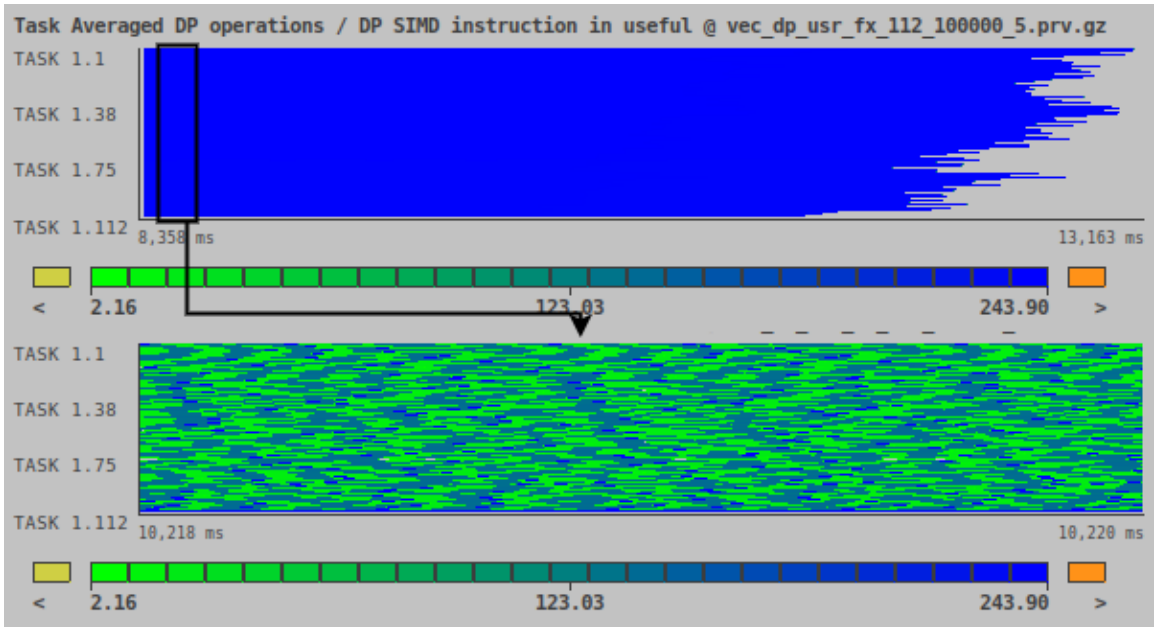

Figure 16: Ratio between double-precision floating point operations and double-precision SIMD instructions in the computation phase, execution with 112 MPI tasks, 100,000 cells and 5 time-steps.

We can see a different behavior than the single-precision case; there are SIMD instructions in the whole computation phase but the value of the ratio is higher than expected most of the time (200 double-precision floating point operations per vector instruction) this means that the double-precision operations also are not vectorized properly as there are a lot more operations than SIMD instructions.

To check the reason for this lack of vectorization in CAMP, we generated an optimization report using the compiler(using `-qopt-report=2 -qopt-report-phase=loop,vec -qopt-report-file=stderr`) aiming to check the vectorization in the most-time consuming user functions, placed on the CVode library. Table 2 contains a summary of the non-vectorized loops and the reason from the optimization report. We have divided it into two blocks, on the left are the non-vectorized loops that we assume as impossible to vectorized and on the right are the loops that potentially can be vectorized by the compiler, attending to the reason for non vectorizing that loop:

Non-vectorized loops: Impossible to vectorize		**Non-vectorized loops: Possible to vectorize**	
vector dependece	417	multiple exits loop cannot be vectorized	82
inner loop vectorized	267	too complex control flow	18
function call cannot be vectorized	24	loop iteration count cannot be computed before executing the loop	106
possible vectorization but seems inefficient	246	loop control variable was not idenfitied	4
insufficient number of iterations	1		
non-vectorized loop instance from multiversioning	192		
loop transformed to memset/memcpy	40		
	1187		**210**

Table 2: Non-vectorized loops and reasons reported in the optimizations report from the Intel compiler.

The group of *Possible to vectorize* needs a refactor to potentially enable vectorization on that

part of the source code, the potential gain on the loops without vectorization is proportional to the size of the vector unit of the machine. There are a lot of loops to review, but this is because the source uses similar structures; then we only need to find one optimization to each of the reasons listed on Table 2 and apply the changes to each loop. As an example, the error management strategy of the whole CVode is based on adding a break statement in the loops to finish it if some error happens which generates a multiple exits loop that cannot be vectorized.

There is the possibility also that some loops marked for us as possible to vectorize cannot be refactored due to the algorithm. As an example, the reason *loop iteration cannot be computed before executing the loop* is highly likely to be impossible to be refactored to enable vectorization.

3 Conclusions

The profiling analysis of the CAMP chemical model has proved a good overall performance and strong scaling. Across a complete analysis using PAPI performance counters we checked that the IPC and cache miss ratio have a very good performance but the CAMP dwarf does not exploit the vectorization.

Remarkably, this profiling analysis was conducted using only one Marenosturm 5 CPU node, no interconnection between different nodes was tested and it could be interesting in the future to check the performance across nodes. Since the unique MPI communication is done at the beginning of the execution it is expected that the effect of using the interconnection network will be small. Also, leaving empty threads on those nodes can be beneficial in terms of performance if the target is to scale to a lot of MPI tasks.

The major issue detected is related to the vectorization of the application, as described in the section Section 2.2.5. We checked that the computation phase has a very small vectorization ratio and using the performance report of the Intel compiler, we have detected possible loops that with a refactor can increase the vectorization effectiveness. Any optimization effort needs to target this issue before any other.

A secondary issue detected is an imbalance, around 20%, in the number of instructions executed by the different MPI tasks. As we have seen the timesteps can take different times and we know that CAMP uses a convergent algorithm that explains this variability in the number of instructions. An imbalance less or equal to 20% is acceptable, so it's something that for us doesn't need to be optimized.

Last but not least, we also detected in the first stages of the profiling a lack of scalability with 80 and 112 MPI tasks. After the study of the performance counters we have not detected any explanation for the IPC and frequency scalability, so we conclude that the bad metrics reported in Fig. 3 happen because from the execution of 10 MPI tasks the IPC has a very high value.

In conclusion, this profiling analysis of the CAMP CPU dwarf shows a very good overall, which was expected, as this dwarf splits the workload at the initialization phase and all the computation happens without any communication. The unique point that requires work is to optimize the vectorization of the 210 loops marked as possible to vectorize on Table 2.

4 Appendix

4.1 BSC Tools abstract performance model

We have used the abstract performance model from the BSC tools. This model defines several hierarchical metrics that provide an overview of an application. Fig. 17 shows all the metrics of the model and possible reasons for a bad score of each metric.

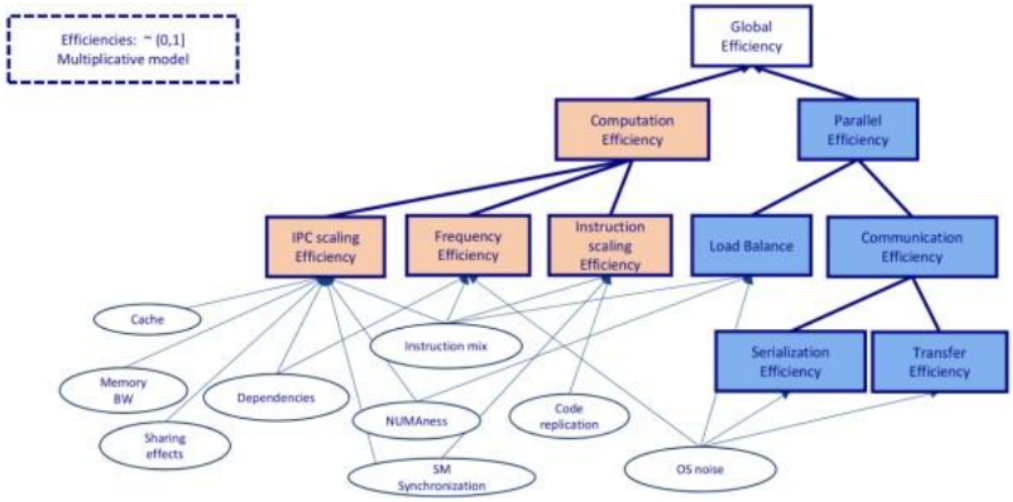

Figure 17: Hierarchical structure of the performance model. The metrics are displayed as rectangles and the possible explanations as circles.

On the one hand, the computation efficiency (or scalability) measures how the global computation of the application scales, using the following metrics:

- IPC scaling efficiency: It measures the global average Instructions per Cycle (IPC) scaling efficiency.
- Frequency efficiency: It measures the global average CPU clock frequency scaling efficiency.
- Instruction scaling efficiency: It measures the global average number of instructions scaling efficiency.

On the other hand, parallel efficiency (or scalability) measures how efficient is the parallelization by calculating which percentage of time is computation and which percentage of time the application is in the runtime (e.g. executing MPI code). The metrics used are:

- Serialization efficiency: It measures how much the code is serialized such as waiting due to dependency chains, imbalances that are compensated, etc.
- Transfer efficiency: It measures the time devoted to transferring data.
- Communication efficiency: It measures the loss of efficiency that is not caused by the global load imbalance, this is, due to serialization and transfer. It is based on the serialization and transfer efficiency.
- Load balance: It measures how globally balanced work is between all processes.

References

[1] Christian Guzman Ruiz et al. "Optimized thread-block arrangement in a GPU implementation of a linear solver for atmospheric chemistry mechanisms". In: *Computer Physics Communications* 302 (Sept. 2024), p. 109240. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2024.109240. URL: https://www.sciencedirect.com/science/article/pii/S0010465524001632 (visited on 05/23/2024).

[2] Intel Corporation. *Intel Xeon Platinum 8480 Processor specifications*. https://www.intel.com/content/www/us/en/products/sku/231746/intel-xeon-platinum-8480-processor-105m-cache-2-00-ghz/specifications.html. Accessed: 2024-08-20. 2024.