Scalable Parallel Astrophysical Codes for Exascale

Performance profiling and benchmarking

Deliverable number: D2.1

Version 2.0/2.0

Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Belgium, Czech Republic, France, Germany, Greece, Italy, Norway, and Spain under grant agreement No 101093441

Project Information

Project Acronym: SPACE
Project Full Title: Scalable Parallel Astrophysical Codes for Exascale
Call: Horizon-EuroHPC-JU-2021-COE-01
Grant Number: 101093441
Project URL: https://space-coe.eu

Document Information

Editor:	Lubomir, Riha - IT4I@VSB, Sijing, Shen - UiO
Deliverable nature:	Report (R)
Dissemination level:	Public (PU)
Contractual Delivery Date:	31.10.2023
Actual Delivery Date	23.07.2024
Number of pages:	142
Keywords:	performance analysis, POP methodology, region analysis
Authors:	Andrea Mignone, Marco Rossazza – UniTo Radim Vavrik, Kristian Kadlubiak – IT4I@VSB Ondrej Kozinski, Tomas Panoc – IT4I@VSB Benoît Commerçon, Tristan Coulange – CRAL CNRS Geray Karademir – LMU Khalil Pierre, Georgios Doulis – GUF Robert Wissing – UiO Luca Tornatore – INAF Harikrishnan Aravindakshan – KU Leuven Gino Perna – ENGINSOFT
Peer review:	Matthieu, Kuhn – Eviden Manolis Ploumidis, Manolis Marazakis – FORTH

History of Changes

Release	Date	Author, Organization	Description of changes
0.1	01.08.2023	Lubomir Riha, IT4I@VSB	Started the document and setup of the document structure.
0.8	15.09.2023	all authors	Version for reviews.
0.9	02.11.2023	all reviewers and all authors	Text updated based on reviews.
1.0	03.11.2023	Lubomir Riha, IT4I@VSB	Final edits before submission.
1.2	02.07.2024	Lubomir Riha, IT4I@VSB	Integrations, Added Section 1 entirely.
1.9	19.07.2024	Luca Tornatore, INAF	Review of section 1.
2.0	23.07.2024	Gino Perna, ES	Document finalization.

DISCLAIMER

The space above and below the message intentionally is left blank.

Executive Summary

The main goal of this deliverable is to provide SPACE CoE codes scalability on JU systems, performance assessments and identification of the regions of the codes that can be potentially extracted as mini-applications or kernels and optimized in the following activities of WP1 and WP2. To evaluate the scalability and efficiency of specific performance aspects in the SPACE CoE parallel codes, we use a performance model and analysis methodology developed within the POP and POP2 Centers of Excellence [1].

We consider this as a crucial point for two main reasons. The first one is that the POP methodology [2] can be considered a standardized approach to evaluate the performance of parallel codes and, as such, it allows one to compare the parallel performance metrics between different applications coming from different scientific domains and also using different programming models. For example, the POP methodology can provide the same efficiency metrics for MPI, MPI+OpenMP, or MPI+CUDA acceleration programming models. In order to strengthen the upcoming collaboration with POP3 CoE, all codes were instrumented and traced using state-of-the-art European performance analysis tools also used by POP. Furthermore, we are immediately ready to collaborate in a very efficient way with POP3 CoE, once an in-depth analysis will start, as we can directly provide the traces produced in the frame of this study to POP experts.

Description of the POP methodology and the subset performance metrics that have been used in this deliverable is presented in Section 2. In addition we also present all tools used and the tracing procedure. We employed the POP tools on all codes with the exception of ChaNGa, because it is written in Charm++ and POP tools at this point do not support this PGAS implementation. In this particular case we used special tools, Projections, described in Section 2.3. This section also describes the specific measure we adopted to analyze ChaNGa.

The summary of all experiments performed for this deliverable is shown in the table bellow. The table also contains links to the sections of particular codes. The important fact is that we have several strong scalability points to evaluate to see the trends in the POP metrics as one increases the level of parallelism.

Code	link	Programming model used for D2.1	Strong scaling [nodes] x (procs/threads)
Pluto	Section 3	MPI	8 - 128 x (128/1)
Gadget	Section 4	MPI + OpenMP	16 - 64 x (32/4)
iPic3D	Section 5	MPI	8 - 64 x (128/1)
RAMSES	Section 6	MPI	16 - 128 x (128/1)
BHAC	Section 7	MPI	2 - 16 x (128/1)
FIL	Section 8	MPI + OpenMP	8 - 64 x (8/16)
Changa	Section 9	Charm++	8 - 64 x (8/16)

Table 1: The summary table of the strong scalability tests used for performance evaluation.

All the performance data for this deliverable were collected on the Karolina cluster [3] CPU partition in IT4I, i.e., using compute nodes equipped with 2x AMD EPYC 7H12 (64-cores, 2.6 GHz nominal frequency) processors and 256 GB DDR4 3200 MT/s memory that are interconnected through the 100 Gb/s InfiniBand HDR100 network. The total theoretical peak performance of the partition (Rpeak) is 3.83PFLOP/s. The network topology is the non-blocking Fat Tree, which consists of 60 x 40-ports HDR switches (40 Leaf HDR switches and 20 Spine HDR switches).

There are several key points that this deliverable achieved:

1. we have used unified and standardized metrics to quantify the scalability properties of the codes and setup the baseline for further improvements;

2. the performance evaluation was done at scale; some codes were analyzed on up to 16,384 CPU cores (128 nodes of the Karolina cluster), allowing us to see bottlenecks that are not necessarily present at lower scales;

3. we have worked very closely with code owners to identify regions of interest and focused the work of the analysis of these particular regions;

4. for every annotated region, and when possible, we have evaluated the POP metrics and highlighted the potential reasons behind the observed limited scalability and parallel efficiency in general;

5. we have identified possible optimization actions for every annotated region.

We summarize the observations and recommendations in Table 2 in a compact way. At the same time, the table gives a feeling about the very large amount of work that has been done to prepare the final document.

Code name	Section	Issues detected from POP metrics: load balance	comm. efficiency	computation scalability	Region type	Recommended optimization	Target for GPU offload
Pluto	3						
	3.3 Region 1	x	N/A	good	compute	LB, INR	yes
	3.4 Region 3	x	N/A	good	compute	LB, INR	yes
	3.5 Region 6	x	N/A	good	compute	LB, INR	yes
	3.6 Region 9	x	N/A	good	compute	LB, INR	yes
	3.7 Region 10	x	transfer eff.	x	comm.	INA/comm.	no
Gadget	4						
	4.3 Region 0	x	x	good	compute/comm.	LB, INA/comm.	yes
	4.4 Region 1	x	x	good	comm.	LB, INA/comm.	no
	4.5 Region 2	x	x	good	comm.	LB, INA/comm.	no
	4.6 Region 3	x	good	good	compute	INR	yes
	4.7 Region 4	good	x	good	comm.	INR	yes
	4.8 Region 5	good	x	good	compute	INA/comm.	yes
	4.9 Region 6	x	good	good	compute	INR	yes
iPic3D	5						
	5.3 Region 1	x	x	good	compute/comm.	INA/comm	no
	5.4 Region 2	good	x	x	compute/comm.	INA/comm	no
	5.5 Region 3	x	x	good	compute/comm.	INA/comm	no
	5.6 Region 4	x	x	good	compute/comm.	INA/comm	no
RAMSES	6						
	6.3 Region 1	x	good	good	compute	INR	yes
	6.4 Region 2	x	good	good	comm.	INA/comm	no
	6.5 Region 3	x	good	x	comm.	INA/comm	no
BHAC	7						
	7.3 Region 1	x	x	x	compute	INR/comm	no
	7.4 Region 2	x	good	good	compute	LB	yes
FIL	8						
	8.4 Region 0	x	x	x	compute/comm.	LB, INA/comm	no
	8.5 Region 1	x	N/A	x	compute	INR	yes
	8.6 Region 2	x	N/A	x	compute	INR	yes
Changa	9						
	9.4 Region 2	x	N/A	good	compute	LB, INR	yes
	9.5 Region 3	x	N/A	good	compute/comm	LB, INA/comm	no
	9.6 Region 4	x	N/A	good	compute	LB	no
	9.7 Region 5	x	N/A	good	compute	LB	no

Table 2: A summary table of POP analysis of the selected regions of all codes including the region type and recommendations for the future optimization (time and energy) and co-design tasks in WP2. (Abbreviations: comm. - communication, LB - load balance, INA/comm - IntrA-Node/communication optimizations, INR - InteR-Node optimizations including single-core optimization like vectorization

Generally speaking, we can divide the regions into two categories: compute regions or communication-heavy regions. The first ones are, in general, good candidates for optimization for new processing architectures with an ever-growing number of processing cores with more powerful vector units or porting to GPU accelerators. The goal here is to achieve good ratio with respect to the theoretical peak performance for compute-bound parts of the codes, and optimal utilization of memory bandwidth for the memory-bound section of the codes. We should also investigate the new memory technologies like High-Bandwidth Memory (HBM) on general-purpose processors. The second group is the essential one if applications are supposed to scale to thousands of nodes on the current pre-exascale or upcoming exascale systems. All this effort will focus on maximizing the utilization of the current and future EuroHPC systems and their respective processing and accelerator architectures. We are confident that the results achieved during this first stage of the proposal will serve as a solid starting point for the next set of efforts and activities planned for the following months.

This document is Public (PU)
and was produced under SPACE project EU GA 101093441

Contents

List of Figures

List of Tables

1 Preamble and Additional Explanation Related to Re-submission

The expert opinion from the first project review on the D2.1 deliverable was: *D2.1: The report raises concerns, particularly regarding the assessment of "computation scalability" as "good" in several instances, such as iPiC3D, which is matching the actual data. How this assessment is conducted and why there is a discrepancy between the technical evaluation and the conclusions need to be clarified. The report requires reworking to address these discrepancies.* In this section, we would like to explain these concerns.

1.1 Explanation

In D2.1, all our evaluation criteria are based on the POP metrics. POP metrics allow the separation of codes' performances between several performance metrics in multiple "layers" and in an orthogonal way. For example, in Figure 10, we can see that **Global Efficiency** is separated into two disjoint metrics: **Computation Efficiency** and **Parallel Efficiency**.

Hence, POP metrics are very convenient for analyzing code performances as they allow one to finely characterize and classify performance issues, identifying where efforts must be produced to improve overall scalability.

Computation scalability is one of the POP metrics. Its meaning is described in Section 2.1. In short, This metric does only take *raw* computations into account (in terms of FLOPS), and not parallel efficiency (communications, load balance, ...).

We see possible misleading information in Figure 10 where instead of **Computation Scalability**, we used **Computation Efficiency**. **These two terms have the same meaning in the POP metrics.**.

The definition of **Computation Efficiency/scalability (CompE)** is: the ratio of total time in useful computation summed over all processes. For strong scaling (i.e. problem size is constant) it is the ratio of total time in useful computation for a reference case (e.g. on 1 process or 1 compute node) to the total time as the number of processes (or nodes) is increased. For CompE to have a value of 1 this time must remain constant regardless of the number of processes. Insight into possible causes of poor computation scaling can be investigated using metrics devised from processor hardware counter data. Two causes of poor computational scaling are:

- Dividing work over additional processes increases the total computation required,
- Using additional processes leads to contention for shared resources,

and we investigate these using **Instruction Scaling** and **Instructions Per Cycle (IPC) Scaling**. **Instruction Scaling** is the ratio of the total number of useful instructions for a reference case (for example, 1 processor) compared to values when increasing the number of processes. A decrease in Instruction Efficiency corresponds to an increase in the total number of instructions required to solve a computational problem. **IPC Scaling** compares IPC with the reference, where lower values indicate that the computation rate has slowed. Typical causes for this include the decreasing cache hit rate and the exhaustion of memory bandwidth. This can cause processes to be stalled while waiting for data.

If we now focus on the summary table of this deliverable, see Table 2, the iPic3D has "good" computational scalability/efficiency for Regions 1,2, and 4 (marked red in Table 3).

Code name	Section	Issues detected from POP metrics: load balance	comm. efficiency	*computation scalability*	Region type	Recommended optimization	Target for GPU offload
iPic3D	5.3 Region 1	x	x	*good*	compute/comm.	INA/comm	no
	5.4 Region 2	good	x	x	compute/comm.	INA/comm	no
	5.5 Region 3	x	x	*good*	compute/comm.	INA/comm	no
	5.6 Region 4	x	x	*good*	compute/comm.	INA/comm	no

Table 3: A summary table of POP analysis of the selected regions of iPIC3D code including the region type and recommendations for the future optimization (time and energy) and co-design tasks in WP2. (Abbreviations: comm. - communication, LB - load balance, INA/comm - IntrA-Node/communication optimizations, INR - InteR-Node optimizations including single-core optimization like vectorization

If we focus on Region 1, we must investigate Figure 49 (also see it below as Figure 1). In this case, the **Computational scalability/efficiency** is 98% for 2048 CPU cores, 95% for 4096 CPU cores, and 95% for 8192

CPU cores. This means that **computational scalability/efficiency** is always high, and therefore marked as "good" in table.

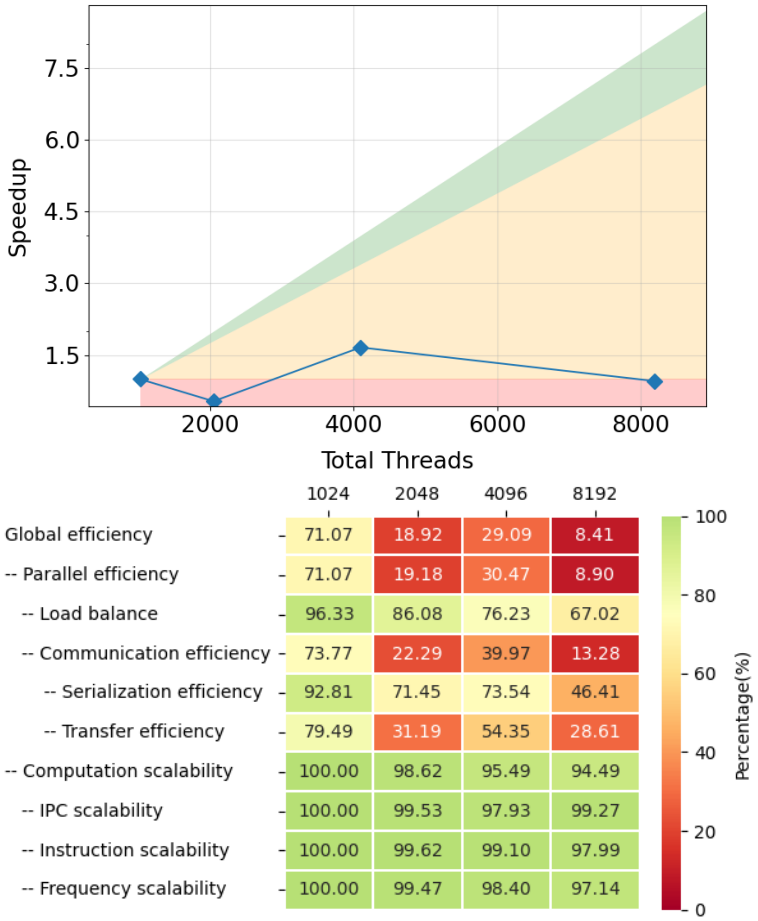


Figure 1: Strong scaling and POP efficiency metrics for Region 1 of iPic3D.

For Regions 3 and 4 the respective tables are Figure 53 and Figure 55. In both cases, the computational scalability/efficiency is in green figures and therefore in the summary table is marked as good.

However, as can be seen in these figures, the main issue that causes the overall scalability or Global efficiency (in POP metrics terms) to be low is **Parallel efficiency**, which is a combination of **Communication efficiency** and **Load balance**.

1.2 Motivation and Focus of D2.1

It is crucial to note that the performance measurements conducted for D2.1 had a specific objective. Our intention was to thoroughly stress-test the codes using small-scale use cases and conduct strong scaling tests with a large number of MPI processes. This approach allowed us to identify the weak points in the codes and bring them to the attention of the code developers. We did not aim to showcase the codes in their best light, as this would have rendered the profiling efforts ineffective. Additionally, we must consider that tracing tools like Extra-e introduced additional overhead, particularly when dealing with short communication or OpenMP regions in the code.

1.3 Scalability of codes

However, this does not mean that the codes do not scale at all. Instead, in this paragraph, we aim to demonstrate how each individual code scales on either on CPU or GPU clusters. These results are also presented in D1.3.

1.3.1 OpenGADGET

In the frame of SPACE, thanks to the collaboration with CINECA, we have been able to perform a large number of tests and assess both the strong and weak scaling of the code using up to 3072 nodes (90% of Leonardo's Booster partition). However, due to the limitations of the available computational resources, we have not been able to profile and assess the scaling of entire simulations but just of some initial time-steps.

We have chosen to simulate the computational boxes included in our scientific cases, explicitly generated for this purpose (the entire set of initial conditions amounts to 3TB). However, we do not have evolved realisations of these cases, since they are computationally very expensive. Then, we simulated only the beginning of the Universe's history at high redshift (we have generated the boxes at $z = 50$). The density distribution at that epoch is very homogeneous, and that is reflected in a non-deep tree and in a very non-clustered distribution of particles; this is a particularly favourable situation for our algorithm, and that is to be accounted for.

For the strong scaling, we used boxes with $1024^3, 2048^3$, and 4096^3 particles spanning the whole amount of nodes (see Figure 2) since a case that fits the lower end would lead to small amounts of data per process at the high end, becoming artificially communication-dominated. On the other hand, a case that could run on the whole machine could not fit in the memory of a small sub-set of nodes. The sequence $(1024^3, 1296^3, 1632^3, 2048^3, 4096^3)$ has been used for weak scaling tests.

In Figure 2 we show the results of the strong scaling test for both the gravity-only (upper panel) and the hydrodynamics (lower panel) test cases. The speed-up and the parallel efficiency are on the left and right columns, respectively. While the **1024^3 case** behaves smoothly, slowly decreasing in efficiency which remains $> 80\%$ for a $32\times$ scaling factor, the **2048^3 case** shows a sudden decrease beyond $16\times$ for both the Gravity and the Hydro cases.

Conversely, the **4096^3 case** (which we run only for the Gravity-only case due to the limits on the available computational time) exhibits a super-linear speed-up for $2\times$ and $4\times$, while a malfunction in the network led to a sudden increase in the communication time for $8\times$ (3072 nodes). We plan to repeat this run as soon as the whole machine is available for large tests.

As a general conclusion, **the current GPU implementation offers a viable opportunity to run a simulation with a net gain of about 3**$\times$, which is a very significant result given that state-of-the-art simulations require a computational effort of the order of 10^7 core-hours.

1.3.2 PLUTO

The `Boundary()` function handles nearly all inter-process communication by setting both internal (that is, inter-processor) and physical boundary conditions on all of the sides of the computational domain, by filling ghost zones for both cell-centered and face-centered data arrays. This routine is fundamental as it handles nearly all inter-process communications and is invoked one time per Runge–Kutta Stage (e.g., 3 times for a 3^{rd}-order time stepping.

The type of boundary condition at the leftmost or rightmost side of a given grid is specified by the integers `grid[dir].lbound` or `grid[dir].rbound`, respectively. If this value is non-zero, it indicates that the local processor borders a physical boundary. If the value is zero (indicating an internal boundary), two neighbouring processors sharing the same side fill ghost zones by exchanging data values. This process is repeated for each dimension and applies to both cell-centered and staggered data arrays.

Currently, our communication routines have been tested in the synchronous (blocking) implementation using MPI and the NCCL. Figure 3 reveals the current performance of the code on a weak scaling test up to 256 nodes (1 node = 4 GPUs or 32 cores when running on CPUs) on Leonardo.

1.3.3 BHAC

To date, BHAC has been compiled and run on the CPU partitions of the EuroHPC JU clusters Karolina, Leonardo DCGP, LUMI-C, and Vega. The benchmark problems (1D shocktube and 2D magnetised spherical accretion onto a Schwarzschild BH) that were run on these machines gave as expected similar results. Also, the strong (see Figure 4) and weak (see Figure 5) scaling performance of BHAC on Leonardo DCGP and LUMI-C shows similar behaviour up to roughly 7000 and 5000 cores, respectively.

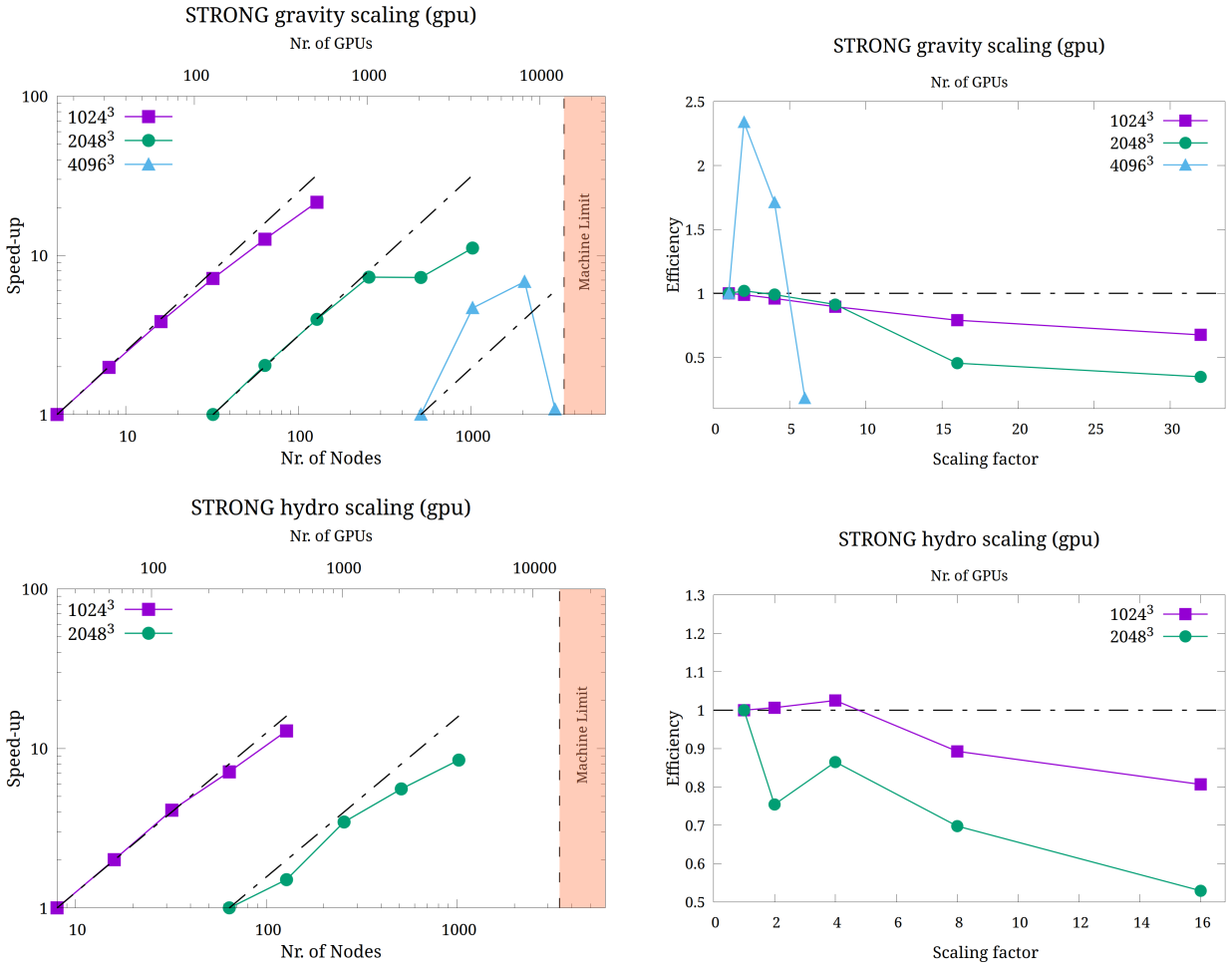


Figure 2: The complete results of **strong scaling** test for both the Gravity-only (**upper panel**) and Hydro (**bottom panel**) cases, from 4 up to **3072 nodes** on Leonardo Booster. The left column shows the number of nodes vs speed-up and the left node the scaling factor vs efficiency. The **Black dashed** lines represent ideal speedup. In all the runs, the **MPI/OpenMP** configuration is such that one MPI task was running per GPU, while all the 32 available CPU cores were filled with OpenMP threads, which amounts to 4 MPI tasks/node and 8 threads/task. Note that some issues arose in domain decomposition with the 2048^3 case after 512 nodes, and a network problem has invalidated the 3072 nodes run of the 4096^3 case.

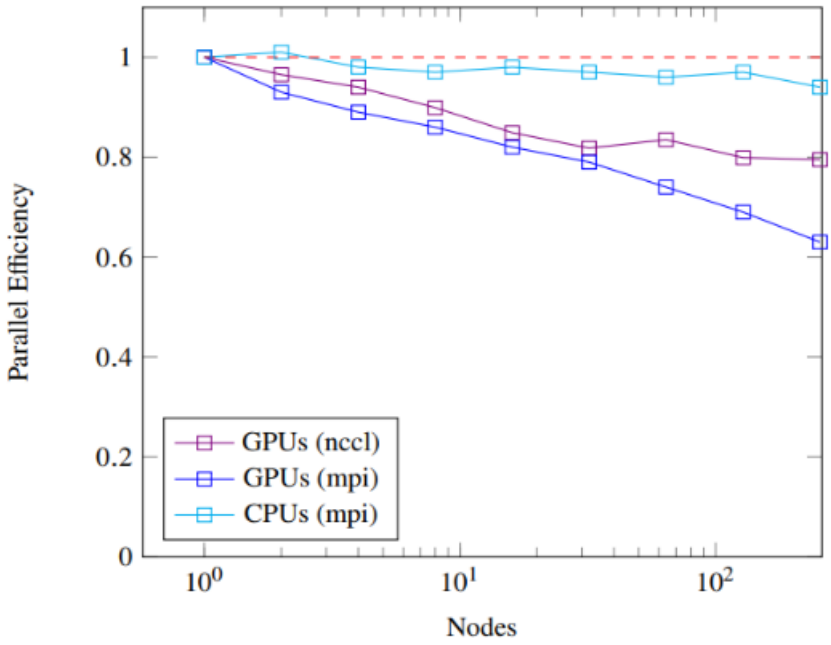


Figure 3: Weak scaling test of Pluto on Leonardo running the 3D MHD Orszag-Tang problem. Here 1 node equals 4 GPUs (for GPU runs) or 32 cores (for CPU runs).

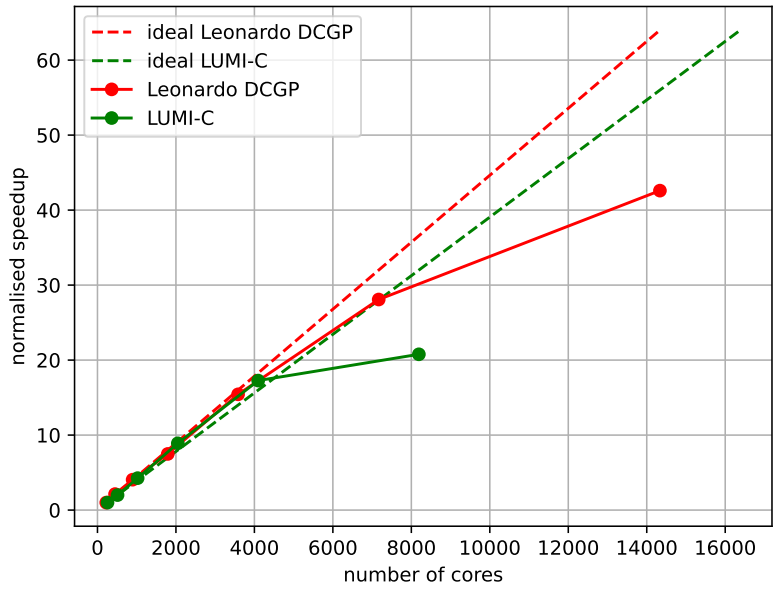


Figure 4: Strong scaling of BHAC on Leonardo DataCentric and LUMI-C.

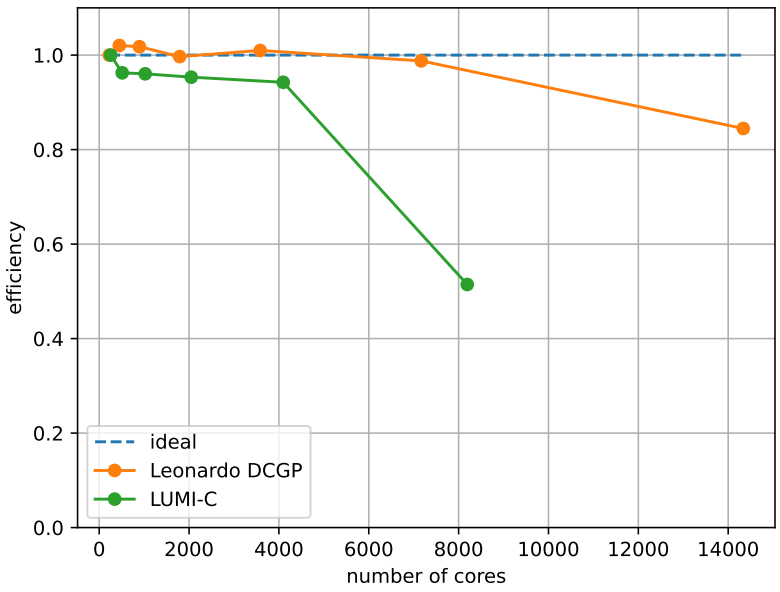


Figure 5: Weak scaling of BHAC on Leonardo DataCentric and LUMI-C.

Strong and weak scaling tests were performed on the EuroHPC JU supercomputers Leonardo DCGP @ CINECA and LUMI-C. Equipped with 1536 nodes with 112 cores each and 2048 nodes with 128 cores each, respectively, Leonardo DCGP and LUMI-C are ideal machines for scaling up to tens of thousands of cores. For the scaling the magnetised torus around a Kerr black hole described in D1.1 was used as a test case. The resolution used for the tests is $(r, \theta, \phi) = (1024, 448, 1024)$ on Leonardo DCGP and $(r, \theta, \phi) = (1024, 512, 1024)$ on LUMI-C resulting in a computational domain consisting of half a billion cells. The block based grid structure of BHAC divides the computational domain to a specific number of blocks containing a given number of cells. BHAC requires that the number of blocks in a simulation is equal to or greater than the number of cores used. Here, we allocate one block to each core used, increasing gradually the number of blocks covering the computational domain. For the strong scaling this is done by decreasing the number of cells per block while keeping the total number of cells constant and for the weak scaling by keeping the number of cells per block constant while increasing the total number of cells. Strong scaling results are depicted in Figure 4, BHAC scales good up to roughly 4000 cores and then because of issues related to load imbalance parallel efficiency drops. The weak scaling results are depicted in Figure 5, the efficiency is above 98% up to roughly 7000 cores on Leonard DCGP and above 94% up to roughly 4000 cores on LUMI-C.

1.3.4 ChaNGa

ChaNGa has, so far, been compiled and benchmarked on the Karolina-CPU, LUMI-C and Leonardo-DCGP clusters. In Figure 6, we see the scaling of the single time step and the gravity module on Karolina, LUMI and Leonardo, for a 25 Mpc cosmological simulation with 2 billion particles at high redshift. From the figure, we can see that there is loss of parallel efficiency at higher node counts for the single time step. This is because the runtime of gravity becomes comparable to that of domain decomposition (DD) and load balancing (LB). In a previous version of ChaNGa [4], an optimisation known as MetaLB was used to limit the overhead from DD and LB. This detects if there is an imbalance in the load of tree pieces and only performs a DD and LB step if this is the case. However, the MetaLB optimisation is currently not functioning properly in the latest version, as such DD and LB is always performed at every step in the current version of ChaNGa. From the figure we can see that the scaling single time step differ between clusters, which comes down to different scaling in DD and LB. This is likely due to the difference in MPI and architecture between the clusters. From past testing we know that Open MPI (Karolina+Leonardo) struggles in handling the large number of messages that ChaNGa produces, therefore other MPI layers have been preferred (mostly MVAPICH2).

This document is Public (PU)
and was produced under SPACE project EU GA 101093441

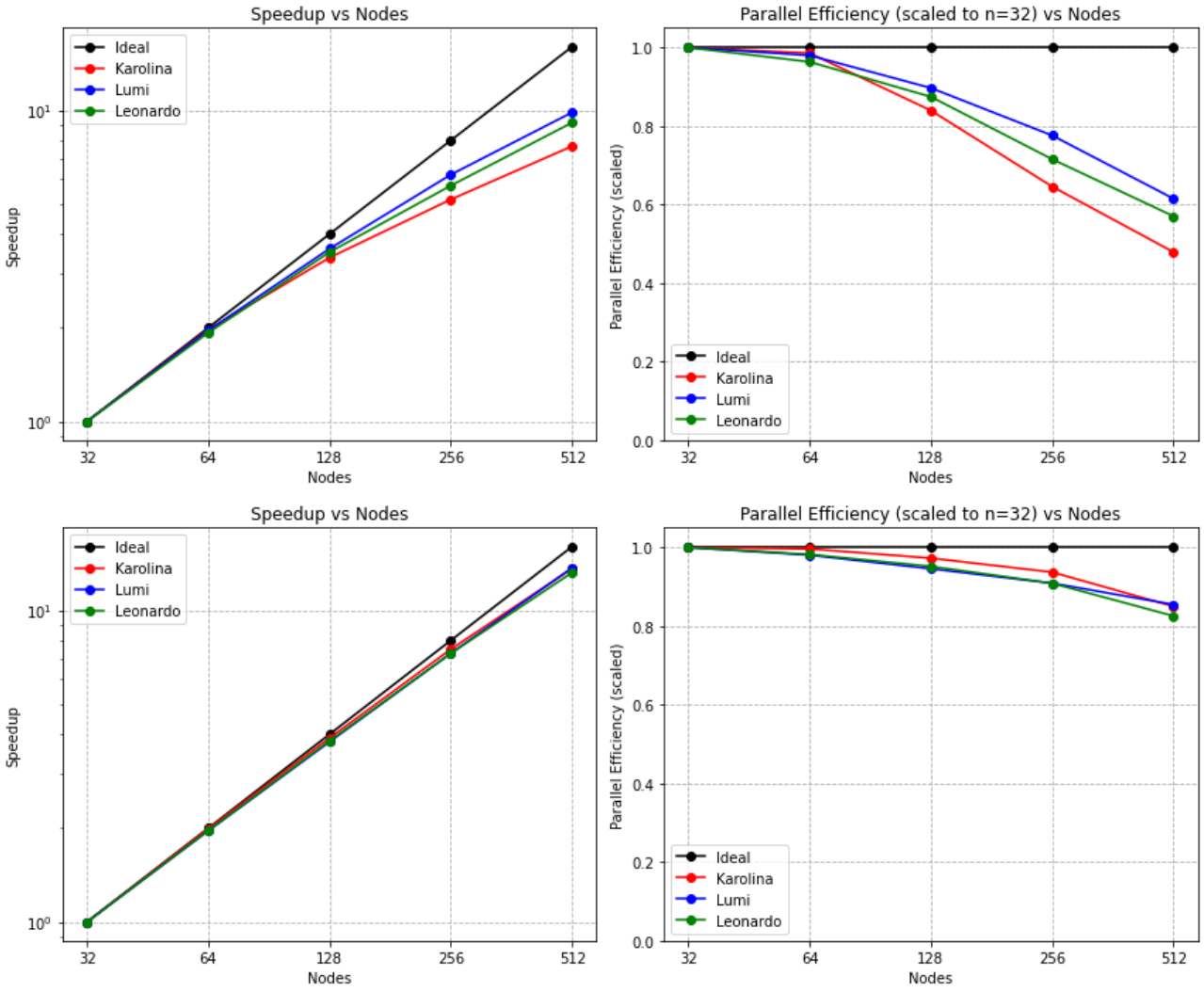


Figure 6: The top figure shows the scaling for a single time step, while the bottom show the scaling of only the gravity module. These simulations was performed on the Karolina-CPU, LUMI-C, and Leonardo-DCGP partitions using a cosmological box containing 2 billion particles within a 25 Mpc volume, using gravity only. The speedup curves have been normalized by the results from the 32-node run for each cluster. When the runtime of gravity becomes comparable to that of domain decomposition (DD) and load balancing (LB), the parallel efficiency declines. This is because DD and LB does not scale as well as gravity. The better single time step scaling seen in LUMI-C is due to better scaling of DD and LB (likely due to differences in MPI and architecture). We should note that this is without the use of the MetaLB optimisation used in [4](which limits the overhead by DD and LB), as this is currently not functioning properly in current version of ChaNGa/Charm.

1.3.5 FIL

To date, FIL has been compiled and run on the CPU partitions of the EuroHPC JU clusters Karolina and Leonardo. The scaling tests for FIL are almost complete and will be presented in the next deliverable. FIL has also been compiled and run on the HLRS HAWK cluster and the SuperMUC cluster where we have recently performed scaling tests. In Figure 7, we can see that FIL scales well up to 32000 cores for a unigrid setup. For the AMR scaling test, FIL performs worse and scales poorly after 2560 cores.

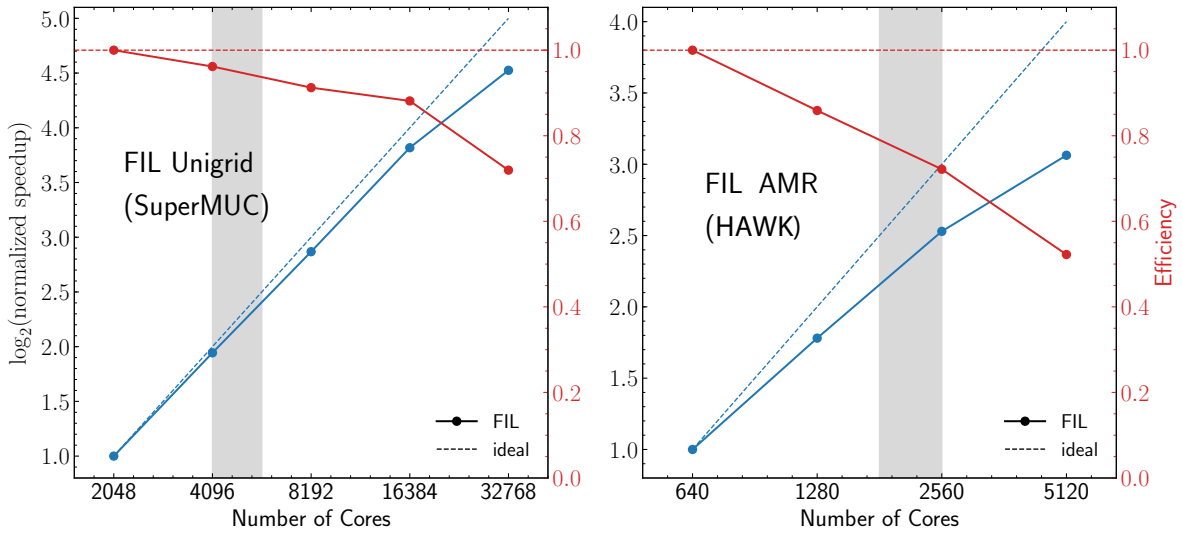


Figure 7: Scaling test of FIL AMR and Unigrid performance on HAWK https://www.cardiff.ac.uk/advanced-research-computing/about-us/our-supercomputers. In this picture dashed lines represent ideal scaling, solid lines are the current performances. Red lines are efficiency, blue lines normalised speedup

1.3.6 iPic3D

We have successfully compiled and run `iPic3D` on Leonardo DCGP, Leonardo Booster (with the particle mover module running on the GPU), LUMI-C, and Karolina CPU. We have applied for EuroHPC JU Call for Proposals for Development Access to run `iPic3D` on MareNostrum5 GPP, MeluXina CPU, Discoverer, and VEGA CPU. An overview of the scalability can be seen in picture **??**. In D1.3 we will present strong and weak scaling tests on these aforementioned clusters.

Additionally, `iPic3D` has been successfully compiled and ran on NVIDIA Grace (ARM Neoverse v2) with `nvhpc` compiler by Elisabetta Boella (E4) who is also directly involved in facilitating GPU offloading of `iPic3D`. She is currently running tests to compare the performance on ARM architecture with that of `x86_64`.

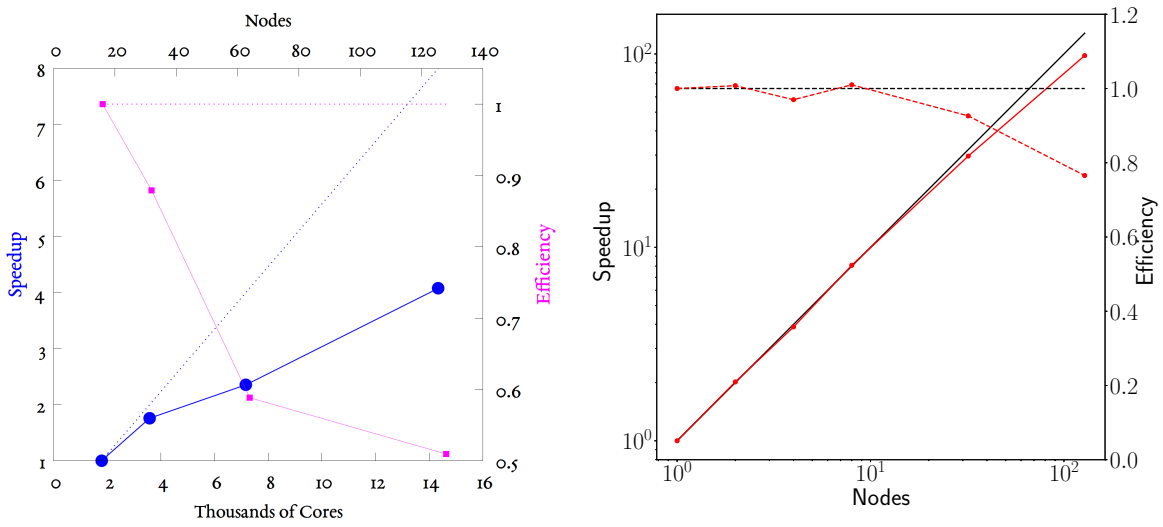


Figure 8: Scaling test for IPic3D. **Left panel** shows the results from a *strong scaling* test: speedup is the blue line and efficiency is the magenta line. Dashed lines are the ideal scaling. **Right panel** shows the results from a *weak scaling* test: both efficiency (horizontal red line) and speedup (oblique red line) are reported. Ideal scaling is highlighted as black solid lines.

The up-to-date scalability of the code is reported in Fig. 8.

1.3.7 RAMSES

We have installed and benchmarked RAMSES on the EuroHPC JU clusters (CPU only) - Karolina, Leonardo DCGP, and LUMI-C. We ran the three test cases that we presented in D1.1. Figure 9 shows a strong scaling test performed with RAMSES on the EuroHPC clusters Karolina, Leonardo DCGP, and LUMI-C.

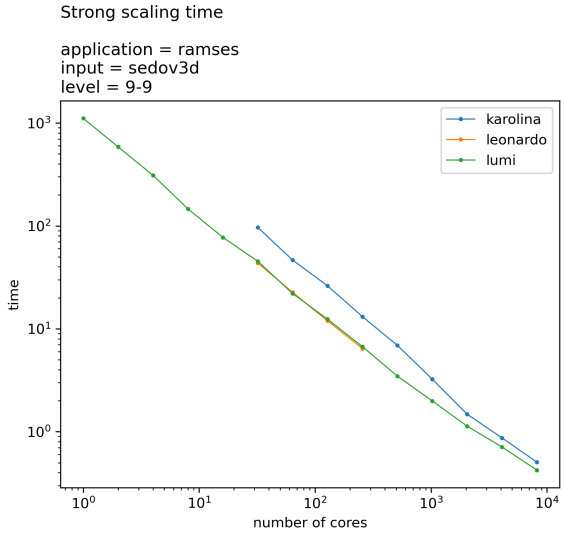


Figure 9: Strong scaling of RAMSES on a Sedov-Taylor blast test. The grid corresponds to the small configuration of RAMSES test case 1, i.e. a resolution of 512^3. The total wall-clock time is given as a function of the number of MPI processes. Overall, the Karolina, Leonardo DCGP, and LUMI-C clusters show similar performance.

2 Codes Performance

The main goal of this deliverable is to provide performance assessments of the SPACE CoE codes and identify the regions of the codes that can be potentially extracted as mini-applications or kernels and therefore further optimized in the following activities of the WP1 and WP2.

To evaluate the scalability and efficiency of particular performance aspects in the SPACE CoE parallel codes, we use a performance model and analysis methodology developed within the POP and POP2 Centers of Excellence [1].

We consider this as a crucial point for two main reasons. The first reason is that the POP methodology [2] can be considered a standardized approach to evaluate the performance of parallel codes and, as such, it allows one to compare the parallel performance metrics between different applications coming from different scientific domains and also using different programming models. For example, the POP methodology can provide the same efficiency metrics for MPI, MPI+OpenMP, or MPI+CUDA acceleration programming models.

The second reason is that if within this project we do not have enough resources or expertise to provide an extremely detailed performance evaluation, we can and will collaborate with POP3 CoE and their experts on such cases. Thanks to the initial work in WP2 reported in this deliverable, we now have a working annotation of the codes and functional tracing up to 16,384 CPU cores (either MPI only or MPI+OpenMP) using POP tools. This significantly improves the interaction with POP3 CoE, as their experts do not have to trace the codes by themselves, but only analyze the traces provided by us.

2.1 Performance analysis methodology

Attempting to optimize the performance of a parallel code can be a daunting task and often is difficult to know where to start. For example, we might ask if the way computational work is divided is a problem. Or perhaps the chosen communication scheme is inefficient? Or does something else impact performance? To help address this issue, POP has defined a methodology for the analysis of parallel codes to provide a quantitative way of measuring the relative impact of the different factors inherent in parallelization. This section introduces these metrics, explains their meaning, and provides information on the thinking behind them.

A feature of the methodology is that it uses a hierarchy of metrics, each metric reflecting a common cause of inefficiency in parallel programs. These metrics then allow comparison of parallel performance (e.g., over a range of thread/process counts, across different machines, or at different stages of optimization and tuning) to identify which characteristics of the code contribute to inefficiency.

The first step to calculating these metrics is to use a suitable tool (e.g., Extrae) to generate trace data whilst the code is executed. The traces contain information about the state of the code at a particular time (e.g., it is in a communication routine or doing helpful computation) and also contain values from processor hardware counters (e.g., number of instructions executed, number of cycles).

The metrics are then calculated as efficiencies between 0 and 1, with higher numbers being better. In general, we consider efficiencies above 0.8 as acceptable, whereas lower values indicate performance issues that need to be explored in detail.

The approach outlined here is applicable to various parallelism paradigms. However, for this deliverable, we distinguish two models of metrics: 1) for codes based on distributed-memory computing (i.e., message passing communication model) and 2) for hybrid codes combining distributed-memory and shared-memory computing (e.g., MPI and threading with OpenMP). In the following subsections, we describe both models.

2.1.1 Metrics for codes with the distributed-memory computational model

At the top of the hierarchy, see Figure 10, is **Global Efficiency (GE)**, which we use to judge the overall quality of parallelization. Typically, inefficiencies in parallel code have two main sources:

- Overheads imposed by the parallel nature of a code
- Poor scaling of computation with increasing numbers of processes

and to reflect this, we define two submetrics to measure these two inefficiencies. These are **Parallel Efficiency** and **Computation Efficiency**, and our top-level GE metric is the product of these two sub-metrics:

$$\text{GE} = \text{Parallel Efficiency} * \text{Computation Efficiency} \quad (1)$$

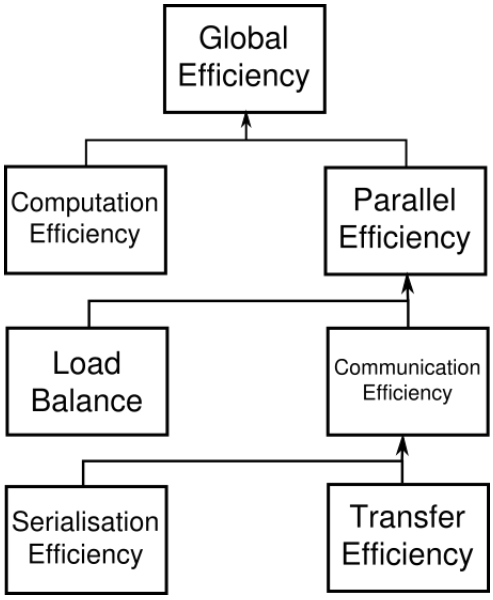


Figure 10: POP metrics hierarchy

Parallel Efficiency (PE) reveals the inefficiency in splitting computation over processes and then communicating data between processes. As with GE, PE is a compound metric whose components reflect two important factors in achieving good parallel performance in code:

- Ensuring even distribution of computational work across processes
- Minimising time communicating data between processes

These are measured with **Load Balance Efficiency** and **Communication Efficiency**, and PE is defined as the product of these two sub-metrics:

$$\text{PE} = \text{Load Balance} * \text{Communication Efficiency} \tag{2}$$

Load Balance (LB) is computed as the ratio between average useful computation time (across all processes) and maximum useful computation time (also across all processes):

$$\text{LB} = \text{average computation time} \,/\, \text{maximum computation time} \tag{3}$$

Communication Efficiency (CommE) is the maximum across all processes of the ratio between useful computation time and total runtime:

$$\text{CommE} = \text{maximum computation time} \,/\, \text{total runtime} \tag{4}$$

CommE identifies when code is inefficient because it spends a large amount of time communicating rather than performing useful computations. CommE is composed of two additional metrics that reflect two causes of excessive time within communication:

- Processes waiting at communication points for other processes to arrive (i.e. serialization)
- Processes transferring large amounts of data relative to the network capacity

These are measured using Serialization Efficiency and Transfer Efficiency. In obtaining these two submetrics, we first calculate (using the Dimemas simulator) how the code would behave if run on an idealized network where transmission of data takes zero time.

Serialisation Efficiency (SerE) describes any loss of efficiency due to dependencies between processes causing alternating processes to wait:

$$\text{SerE} = \text{maximum computation time on ideal network} \,/\, \text{total runtime on ideal network} \tag{5}$$

Transfer Efficiency (TE) measures inefficiencies due to time in data transfer:

$$\text{TE} = \text{total runtime on ideal network / total runtime on real network} \tag{6}$$

These two sub-metrics combine to give **Communication Efficiency**:

$$\text{CommE} = \text{Serialisation Efficiency} * \text{Transfer Efficiency} \tag{7}$$

The final metric in the hierarchy is **Computation Efficiency (CompE)**, which is a ratio of total time in useful computation summed over all processes. For strong scaling (i.e. problem size is constant) it is the ratio of total time in useful computation for a reference case (e.g. on 1 process or 1 compute node) to the total time as the number of processes (or nodes) is increased. For CompE to have a value of 1 this time must remain constant regardless of the number of processes.

Insight into possible causes of poor computation scaling can be investigated using metrics devised from processor hardware counter data. Two causes of poor computational scaling are:

- Dividing work over additional processes increases the total computation required
- Using additional processes leads to contention for shared resources

and we investigate these using **Instruction Scaling** and **Instructions Per Cycle (IPC) Scaling**.

Instruction Scaling is the ratio of the total number of useful instructions for a reference case (e.g., 1 processor) compared to values when increasing the number of processes. A decrease in Instruction Efficiency corresponds to an increase in the total number of instructions required to solve a computational problem.

IPC Scaling compares IPC to the reference, where lower values indicate that the rate of computation has slowed. Typical causes for this include decreasing cache hit rate and exhaustion of memory bandwidth. These can leave processes stalled and waiting for data.

2.1.2 Metrics for codes with the hybrid computational model

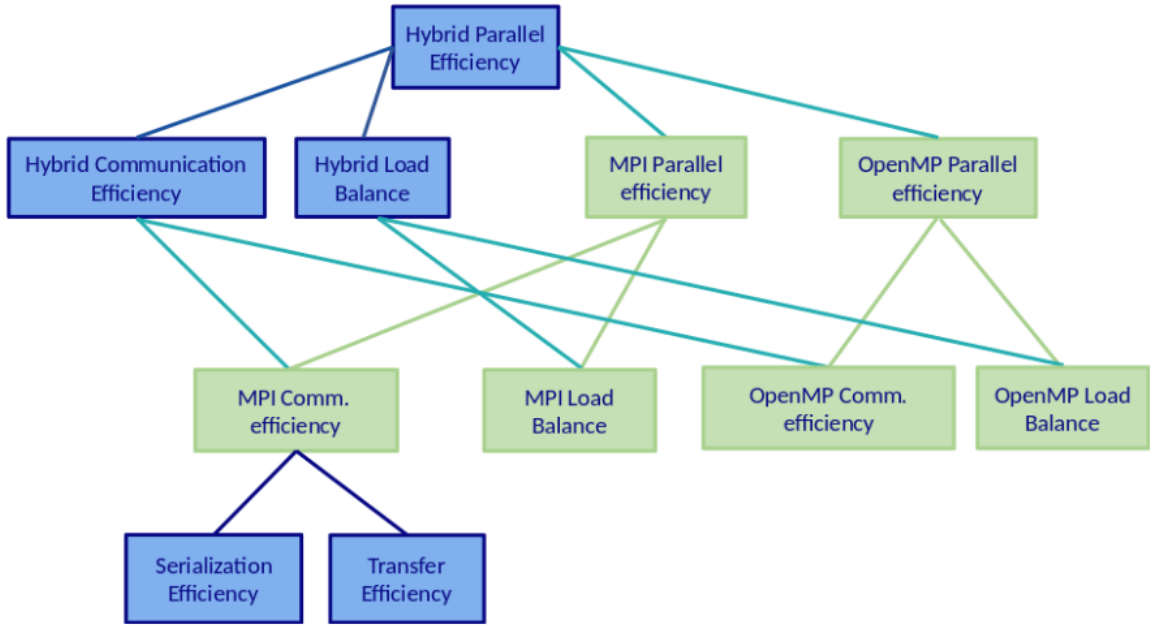


Figure 11: POP multiplicative hybrid metrics hierarchy

Similarly to the previous model, we start describing the hybrid model from the top of Figure 11. We use a multiplicative hybrid model in which the higher-level metrics can be computed as a product of the lower-level metrics. **Hybrid Parallel Efficiency** can be computed from Hybrid Communication Efficiency and Hybrid Load Balance Efficiency, or MPI Parallel Efficiency and OpenMP Parallel Efficiency. The resulting number reflects the percentage of time spent outside of both parallel runtimes (i.e., MPI and OpenMP).

This document is Public (PU)
and was produced under SPACE project EU GA 101093441

Hybrid Communication Efficiency (HCE) shows how much loss is caused by communication. It can be computed directly:

$$\text{HCE} = \text{maximum computation time on all processes and threads} \;/\; \text{total runtime} \tag{8}$$

HCE can be decomposed into MPI Communication Efficiency and OpenMP Communication Efficiency. **MPI Communication Efficiency (MCE)** describes a loss caused by MPI communication. It is computed by a simple formula:

$$\text{MCE} = \text{maximum time outside of MPI calls} \;/\; \text{total runtime} \tag{9}$$

Similarly to Communication Efficiency described in the previous section, MCE can be decomposed and obtained from **MPI Transfer Efficiency** and **MPI Serialization Efficiency**. They are equivalent to the corresponding metrics from the previous section, i.e., Transfer Efficiency and Serialization efficiency.

OpenMP Communication Efficiency (OCE) captures an overhead caused by OpenMP communication constructs such as thread synchronization and scheduling. The formula for OCE is:

$$\text{OCE} = \text{maximum computation time on all processes and threads} \;/\; \text{maximum time outside of MPI} \tag{10}$$

Hybrid Load Balance Efficiency (HLBE) shows how well the distribution of work between processes and threads is done. It is defined as:

$$\text{HLBE} = \text{average computation time} \;/\; \text{maximum computation time} \tag{11}$$

HLBE can be divided into MPI Load Balance Efficiency and OpenMP Load Balance Efficiency. **MPI Load Balance Efficiency (MLBE)** is defined as same as Load Balance in the previous section. **OpenMP Load Balance Efficiency (OLBE)** reflects how evenly work is distributed between threads. The metric can be computed from HLBE and MLBE.

MPI Parallel Efficiency describes the efficiency of parallelization from the MPI point of view. It is the result of a product of MCE and MLBE. Analogously, **OpenMP Parallel Efficiency** focuses on the OpenMP parallelization and it is computed from OCE and OLBE.

Computation Efficiency, Instruction Scaling, and IPC Scaling defined in the previous section are valid for the hybrid model, too.

2.2 Performance analysis tools

In the SPACE CoE project, we will analyze application behavior using various open-source as well as commercial tools [1] to identify the performance bottlenecks of these applications. The performance analysis methodology of the POP-COE project will be used [2].

1. **Extrae**

   Extrae [5] provides performance data collection mostly using the preload mechanism of the linker which enables omitting compilation hooks and executing the unmodified production binary. It intercepts the main parallel runtime environments (MPI, OpenMP, OmpSs, Pthreads, CUDA, OpenCL, SHMEM) and supports all major programming languages (C, C++, Fortran, Python, JAVA). Extrae also provides Basic analysis framework, that automatically generates POP metrics from the Extrae trace file, otherwise it must be evaluated manually.

2. **Paraver**

   Paraver [6] is a trace-based performance analyzer with great flexibility to explore and extract information from Extrae output files. Paraver provides two main visualizations: timelines that graphically display the evolution of the application over time, and tables (profiles and histograms) that provide statistical information. These two complementary views allow easy identification of computational inefficiencies such as load balancing issues, serialisations that limit scalability, cache and memory impact on the performance, and regions with generally low efficiency.

   Furthermore, Paraver contains analytic modules, for example the clustering tool for semi-automatic detection of the application structure, and the tracking tool to detect where to improve code to increase scalability.

[1] The listed commercial tools are available at IT4Innovations or FZJ cluster for the users. No license is required to buy under the SPACE CoE project

3. **PyPOP**

   PyPOP [7] is a python package for calculating POP metrics from application profiles, primarily designed for literate programming using Jupyter notebooks. PyPOP currently consumes Extrae output trace files only.

4. **Region extraction tool for automated analysis of regions**

   It is a tool that can automatically extract subtraces for selected regions in the applications using Extrae-based annotations and calculate the POP metrics for a given region. This functionality is achieved by a combination of different tools designed for the post-processing of Extrae traces and PyPOP. The tool is currently under continuous development and has not been publicly released yet.

5. **Projections**

   Projections [8] is a performance analysis/visualization framework that helps you understand and investigate performance-related problems in Charm++ applications. It is a framework with an event-tracing component, which allows for control over the amount of information generated.

2.3 Charm++ performance analysis specifics

Charm++ is a parallel object-oriented programming paradigm that provides a high-level abstraction of a parallel program. Programs written in Charm++ are decomposed into a number of cooperating message-driven objects. Method invocation on an object causes the Charm++ runtime system to send a message to the object, which may be local or remote in a parallel computation. This message triggers the execution of code within the object asynchronously.

This asynchronous nature, together with other aspects, requires a special approach when it comes to performance analysis. The Extrae and Paraver tools do not have the native support for the Charm++ codes. There is possibility to use these tools, however, the amount of information is rather limited. For this reason, we decided to use native performance analysis tool called Projections. Due to the limitations of this tool, we were not able to provide all the common metrics for the Charm++ codes. In particular, we were not able to capture information regarding instruction count, IPC and frequency and as such metrics derived from these values are not included. Projection tool also lack the capability to simulate ideal network, a feature essential for serialization efficiency and transfer efficiency metrics, so they were omitted as well. We use an alternative definition of communication efficiency motioned in Section 2.1. Other POP metrics were computed according to their respective definitions.

In assessments of performance of the respective regions of Charm++ codes, there was no way to include information regarding communication and idle times. The Projections tool provides us with aggregated runtime of respective regions only. Therefore, the assessment of regions represent scaling of computation. In other words, the assessment neglects the communication and other overheads.

2.4 Hardware platform used for performance assessments

All the performance data for this deliverable were collected on the Karolina cluster [3] CPU partition in IT4I, i.e., using compute nodes equipped with 2x AMD EPYC 7H12 (64-core, 2.6 GHz nominal frequency) processors and 256 GB DDR4 3200 MT/s memory that are interconnected through the 100 Gb/s InfiniBand HDR100 network. The total theoretical peak performance of the partition (Rpeak) is 3.83PFLOP/s. The network topology is the non-blocking Fat Tree, which consists of 60 x 40-ports HDR switches (40 Leaf HDR switches and 20 Spine HDR switches).

Partition	number of node	CPU	memory	Network
CPU partition	720	2×AMD Zen2 EPYC 7H12 2.6 GHz nominal frequency 3.3 GHz boost frequency 64 cores with AVX2 support	256 GB 2 GB/core	1x 100Gbit/s (Infiniband HDR100)

Table 4: Summary of the HW platform used for performance evaluation.

This document is Public (PU)
and was produced under SPACE project EU GA 101093441

2.5 Performance assessments procedure

In this deliverable, we focus on the performance analysis of the CPU versions of the code to set a baseline for further optimizations. All experiments were executed on the CPU partition of the Karolina cluster. For tracing and performance analysis, we used Extrae 4.0.4 and Paraver 4.10.6 tools, both developed at the Barcelona Supercomputing Center. Since these tools do not have support for Charm++, for ChaNGa performance evaluation, we have used a native performance analysis tool for Charm++ Projections 11.0. The tool details are in the above section.

The benchmarking procedure to get final traces with Extra-e was as follows. In the single Slurm jobscript one should execute the following sequence:

1. 1 warmup run with non-instrumented version of the code,
2. 3 runs of non-instrumented version of the code to verify that instrumentation is not a cause for the observed overheads (record the execution time of the application using build in timers)
3. 3 runs of instrumented version of the code to obtain the traces - we will use the best trace (trace with the shortest execution time).

The summary of all experiments performed for this deliverable is shown in the next table. The important fact is that we have several strong scalability points to evaluate to see the trends in the POP metrics as one increases the level of parallelism.

Code	link	Programming model used for D2.1	Strong scaling [nodes] x (procs/threads)
Pluto	Section 3	MPI	8 - 128 x (128/1)
Gadget	Section 4	MPI + OpenMP	16 - 64 x (32/4)
iPic3D	Section 5	MPI	8 - 64 x (128/1)
RAMSES	Section 6	MPI	16 - 128 x (128/1)
BHAC	Section 7	MPI	2 - 16 x (128/1)
FIL	Section 8	MPI + OpenMP	8 - 64 x (8/16)
Changa	Section 9	Charm++	8 - 64 x (8/16)

Table 5: The summary table of the strong scalability tests used for performance evaluation.

In Section 10 we present a Table 39 summarizing our findings on all codes at the coarse level. Essentially, it is a summary of the next more than 100 pages, and it shows all the evaluated regions and recommendations.

2.6 Performance assessments reports structure

Starting from the next section, we dedicate one section of this deliverable to one code. Each section is organized as follows. **Subsection 1** provides a short description of the code to introduce the problems that can be solved with a given code. **Subsection 2** contains a use case description that has been used for performance evaluation. **Subsection 3** describes a high-level structure of the code, including the regions of interest, and shows how the regions are annotated. It also provides a visual representation of all regions within the application runtime.

Subsection 4 focuses on a single iteration of the code, which mainly represents the key workload without the pre- and post-processing. For this region, we also provide the scalability, as well as all key POP metrics.

The following subsections focus on regions of interest that are analyzed one per subsection. For every region we provide a region description to inform a reader about the workload, visual representation of the region and its trace, scalability metrics, and POP performance metrics. Finally, we discuss the observed performance metrics. The basic analysis of performance metrics provides a baseline for any subsequent analysis performed after particular optimizations or updates to the application. The strong scaling charts also include three regions that represent a percentage of the ideal (linear) scaling: Green is 100–80 %, orange means 80–0 %, and red indicates a speedup less than 1, that is, the slowdown. In general, the performance metrics presented in the heat map tables indicate the area of inefficiency, but finding the root cause of the particular issue requires a deeper analysis, which is beyond the scope of this deliverable.

The final subsection provides conclusions for a given application based on performance analyzes.

3 PLUTO

PLUTO [9] is designed to integrate a general system of conservation laws that we write as

$$\frac{\partial U}{\partial t} = -\nabla \cdot T(U) + S(U)\,. \tag{12}$$

Here U denotes a state vector of conservative quantities, T(U) is a rank 2 tensor (the rows of which are the fluxes of each component) and S(U) defines the source terms. Additional source terms may implicitly arise when taking the divergence of T(U) in a curvilinear system of coordinates. An arbitrary number of advection equations may be added to the original conservation law.

Although the components of U are the primary variables being updated, fluxes are computed more conveniently using a different set of physical quantities, which we take as the primitive vector V. Numerical integration of the conservation law above is achieved through shock-capturing schemes using the Finite-Volume (FV) formalism where volume averages evolve in time. Generally speaking, these methods comprise three steps: an interpolation routine followed by the solution of Riemann problems at zone edges and a final evolution stage. In PLUTO, this sequence of steps provides the necessary infrastructure of the code; see the schematic diagram in Figure12: first, volume averages U are more conveniently mapped into primitive quantities V. Left and right states $V_{+,L}$ and $V_{-,R}$ are reconstructed inside each zone and a Riemann problem is then solved between them to obtain the numerical flux function F_+ at interfaces. The solution is finally advanced in time.

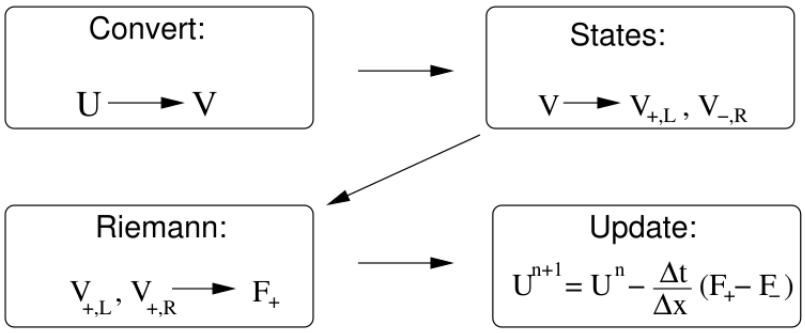


Figure 12: Diagram of the Reconstruct-solve-average (RSA) strategy

3.1 Use-case description

The chosen use-case description is the relativistic magnetized blast wave problem in 3D. It consists of a highly pressurized region inside a sphere embedded in a static uniform medium with lower pressure. The magnetic field is constant and threads through the whole computational domain. The blast wave problem has been specifically designed to show the scheme's ability to handle strong shock waves propagating in highly magnetized environments. Depending on the strength of the magnetic field, it can become a rather arduous test leading to negative densities or pressures if the divergence-free condition is not adequately controlled and if the numerical scheme does not introduce proper dissipation across curved shock fronts.

Below we report the problems parameters: density (`rho`), internal and external pressures (`p (in)` and `p (out)`), magnetic field (`B`), the number of dimensions and the reference.

```
rho  |   p (in) | rho   |   p (out) |  B  | Dim  | Ref
-----|----------|-------|-----------|-----|-------------------------------
0.01 |    1     | 1.e-4 |    3.e-5  | 1.0 |  3   | [Mig_etal2007], sec 5.7
```

The corresponding problem header definition file is:

```
// definitions.hpp

#define  PHYSICS                     RMHD
#define  DIMENSIONS                  3
#define  GEOMETRY                    CARTESIAN
#define  BODY_FORCE                  NO
#define  COOLING                     NO
```

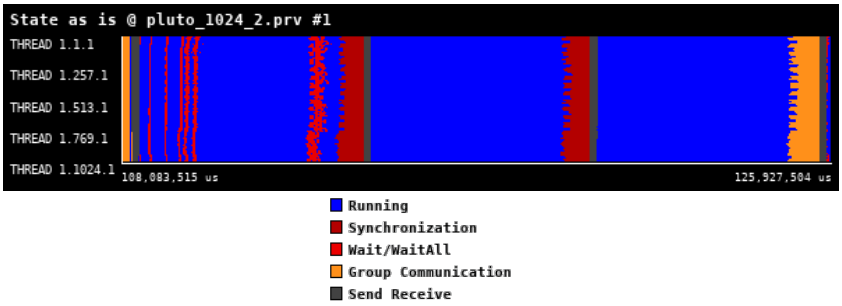
```
#define  RECONSTRUCTION                 LINEAR
#define  TIME_STEPPING                  RK3
#define  EOS                            IDEAL
#define  ENTROPY_SWITCH                 NO
#define  RADIATION                      NO
#define  DIVB_CONTROL                   CONSTRAINED_TRANSPORT
#define  ASSIGN_VECTOR_POTENTIAL        YES
#define  LIMITER                        MINMOD_LIM
#define  CT_EMF_AVERAGE                 UCT_HLL
#define  CT_EN_CORRECTION               YES
#define  GAMMA_EOS                      (4./3.)
```

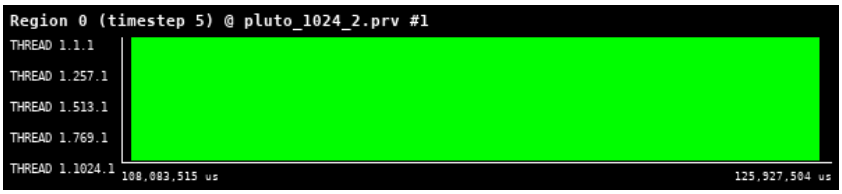
Finally, the corresponding relevant part of the run-time initialization file (`pluto.ini`) is

```
// pluto.ini

CFL              0.25
CFL_max_var      1.1
tstop            4.0
first_dt         1.e-4
Solver           hll
```

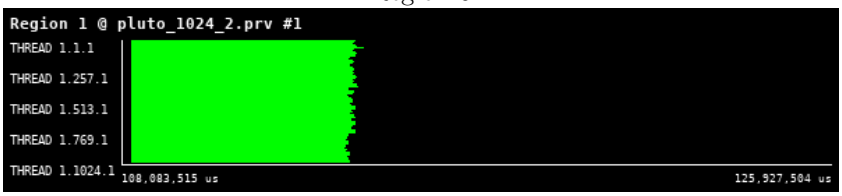
3.2 High-level code structure

The computational part of the code is defined by the while loop that cycles as long as a final simulation time has been reached or the maximum number of steps has been reached. The main operation executed in the body of the while loop is called `AdvanceStep()`. This operation advances equations in time with a Runge-Kutta (RK) time integrator of desired order. For simplicity, the code block describing the code structure depicts a Runge-Kutta integration of order 1. For the 3rd-order Runge-Kutta, as in our selected case, the `AdvanceStep()` function consists of 3 similar stages. Then, each Runge-Kutta stage calls once `UpdateStage()` (which implements a single RK stage) and once `Boundary()`. The `UpdateStage()` function advances the equations in conservative form during a single stage by calling `d->fluidRiemannSolver()` (which computes the conservative fluxes using a Riemann solver) followed by `RightHandSide()` (which computes the actual right hand side of the equation) and also `CT_Update()` (which computes the solution increment for staggered magnetic fields). These operations are repeated once for every dimension. The high level structure of the code is shown in Figure14. The visual representation of individual regions in the scope of the single time step is shown in Figure 13.

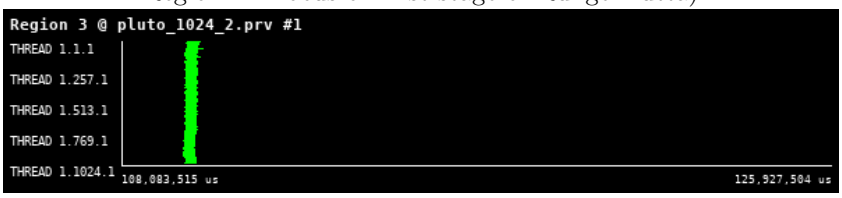




Region 0



Region 1 - Focus on first stage of Runge-Kutta)



Region 3 - Focus on X dimension of the simulation only

Region 6 - Focus on X dimension of the simulation only

Region 9 - Focus on X dimension of the simulation only

Region 10 - Focus on first stage of Runge-Kutta)

Figure 13: Traces for all regions in PLUTO from Paraver showing the high-level structure of the code.

```
while (last_step!=1){

  REGION_START(0);

  AdvanceStep (){ /* RK_STEP ROUTINE
    REGION_START(1);

    REGION_START(10);
    Boundary (){...}
    REGION_STOP (10);

    UpdateStage (){ /* UPDATE_STAGE ROUTINE
      REGION_START(2);

      REGION_START(3);
      RightHandSide (){...}
      REGION_STOP (3);

      REGION_START(6);
      d->fluidRiemannSolver(d, Dts, grid, &box){
      REGION_STOP (6);

      REGION_START(9);
      CT_Update(d, Us, dt, grid){...}
      REGION_STOP (9);

      REGION_STOP(2);
    } /* END UPDATE_STAGE ROUTINE

    REGION_STOP(1);

  } /* END RK_STEP ROUTINE

  REGION_STOP(0);
}
```

Figure 14: Simplified high-level code structure of PLUTO.

3.3 Single time step structure - Region 1

`AdavanceStep()` advances equations with Runge-Kutta time integrator of the 3-rd order. Where U^n denotes

$$\boldsymbol{U}^* = \boldsymbol{U}^n + \Delta t \boldsymbol{\mathcal{L}}^n,$$
$$\boldsymbol{U}^{**} = \frac{1}{4}(3\boldsymbol{U}^n + \boldsymbol{U}^* + \Delta t \boldsymbol{\mathcal{L}}^*),$$
$$\boldsymbol{U}^{n+1} = \frac{1}{3}(\boldsymbol{U}^n + 2\boldsymbol{U}^{**} + 2\Delta t \boldsymbol{\mathcal{L}}^{**}).$$

a state vector of conservative quantities at the time n and L^n is the right-hand side operator that depends on the numerical flux functions that follow the solution of Riemann problems at cell interfaces.

We focus on the first stage of the integration, the predictor step (`g_intStage=1`) to profile only once the inner routines called in `AdvanceStep()`. Different stages of the RK scheme are similar when it comes to computational load.

Figure 15: Zoomed traces for Region 1 of the PLUTO from Paraver showing the structure of the region.

The performance of the evaluation of the entire iteration region follows.

Number of processes	1024	2048	4096	8192	16384
Elapsed time (sec)	5.663043	3.020909	1.629731	0.901387	0.501324
Efficiency	1.0	0.937308	0.868708	0.785323	0.706011
Speedup	1.0	1.874616	3.474833	6.282588	11.296174
Average IPC	1.327813	1.340291	1.338780	1.356088	1.345968
Average frequency (GHz)	1.782482	1.782190	1.790789	1.796826	1.806674

Table 6: Overview of the key performance metrics of the Region 1 of PLUTO.

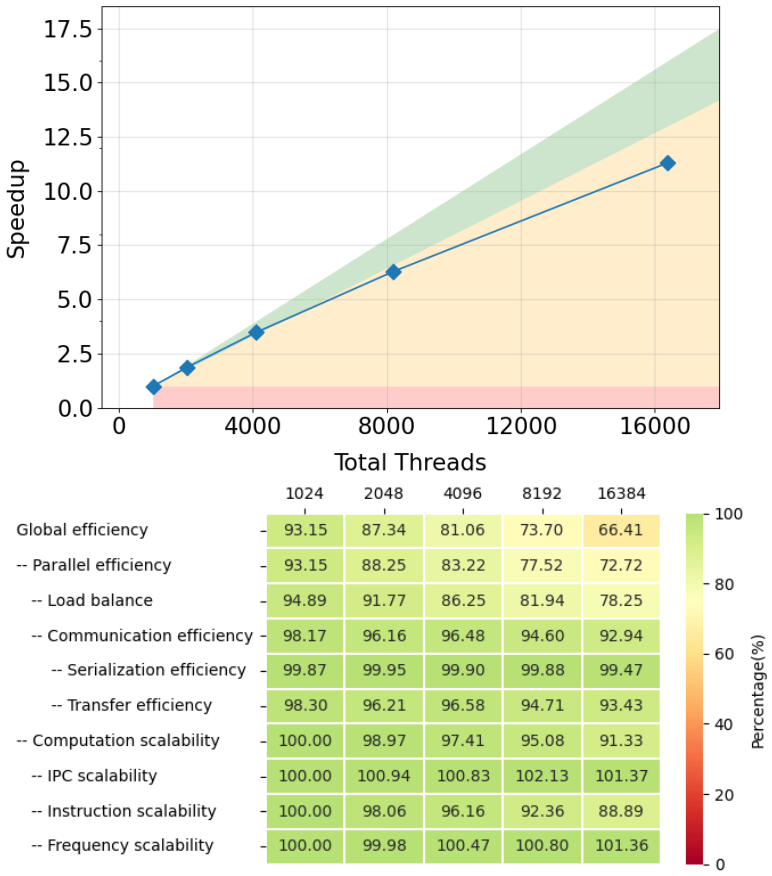


Figure 16: Strong scaling and POP efficiency metrics for Region 1 of PLUTO.

This region includes the kernel regions 3, 6, 9, and 10, which are used for further detailed evaluations in the following sections. The timeline of the region together with the communication pattern is shown in Figure 15. As such it contains several compute kernels without communication and one region where global communication emerges and therefore synchronizes code between iterations. From the performance evaluations, see Figure 15 and Table 6, The Global Efficiency for 16,384 MPI ranks is approximately 66% where the most prominent reason is load imbalance, which we recommend investigating in the follow-up work.

3.4 Right Hand Side - Region 3

This is one of the routines at the core of the computation. It is invoked three times (one time per direction) and as many times as the number of Runge-Kutta stages, so 9 times (one per direction for three RK stages) times for every step. It consists of two computational loops iterating over the whole active domain. The function computes the right hand side of the MHD (General Relativistic Magneto-hydrodynamics) equations in different geometries, taking contributions from one direction at a time. Traces are obtained only of the kernel along x-direction. The right hand side (R) consists of the following contributions:

$$R = -\frac{\Delta t}{\Delta V}(A_{i+1/2}F_{i+1/2} - A_{i-1/2}F_{i-1/2}) + \Delta t\, S, \tag{13}$$

where $A_{i\pm1/2}$ are the right (+) and left (−) interface areas, $F_{i\pm1/2}$ are the interface fluxes, Δt is the time step while S is source term including geometrical source terms and gravity.

In order to compute R, this function takes the following steps:

- initialize R with flux differences
- add geometrical source terms
- add gravity.

We focus on the predictor step, `g_intStage=1` and x-direction `g_dir=IDIR` to profile only once the routine. Computational loads of different directions are similar.

Figure 17: Zoomed traces for Region 3 of the PLUTO from Paraver showing the structure of the region.

Number of processes	1024	2048	4096	8192	16384
Elapsed time (sec)	0.22669	0.116135	0.068069	0.027751	0.036582
Efficiency	1.0	0.975976	0.832574	1.021089	0.387298
Speedup	1.0	1.951952	3.330297	8.168715	6.196763
Average IPC	0.689928	0.691791	0.684749	0.851153	0.940801
Average frequency (GHz)	2.072517	2.072597	2.072089	2.072995	2.072600

Table 7: Overview of the key performance metrics of the Region 3 of PLUTO.

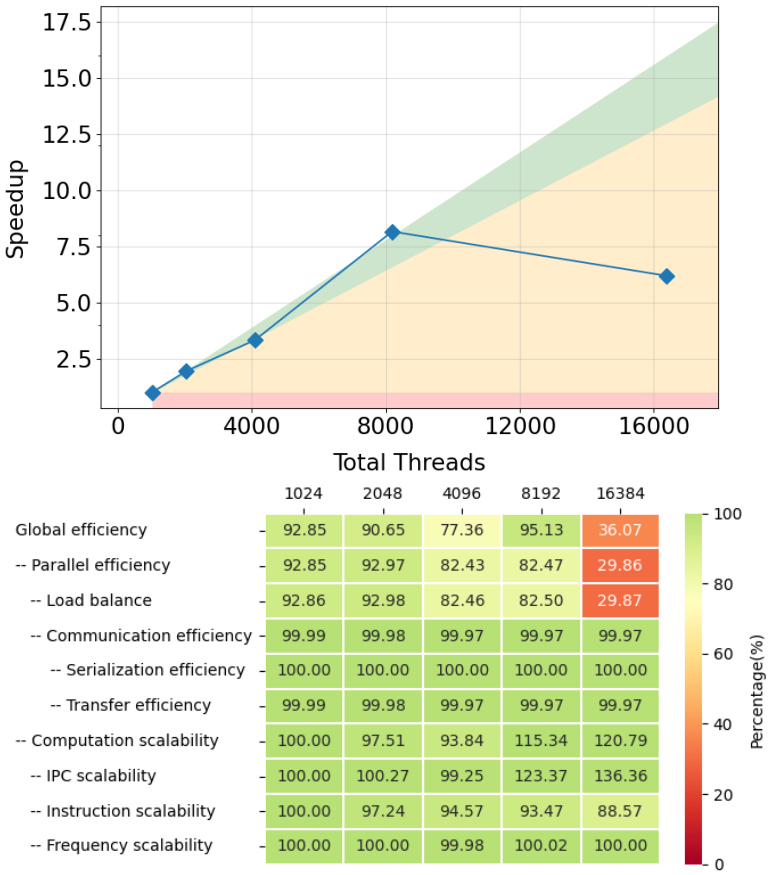


Figure 18: Strong scaling and POP efficiency metrics for Region 3 of PLUTO.

The function does not contain data communication between processors (or GPUs). It is therefore selected for the potential exploitation of advanced vectorization units or accelerators. The loss of scaling observed for more than $\gtrsim 800$ threads deserves more investigation, but it may be attributed to a load imbalance.

3.5 Riemann Solver - Region 6

The Riemann solver represents the most complex single kernel and, as the `RightHandSide()`, is called 9 times per step. Here we solve the Riemann problem for the RMHD equations using the Harte-Lax-van Leer (HLL) Riemann solver. On input, this function takes left and right primitive sweep vectors `sweep->vL` and `sweep->vR` at zone edge `i+1/2`. On output, it returns flux and pressure vectors at the same interface `i+1/2` (note that the index i refers to `i+1/2`). Also during this step, it computes the maximum wave propagation speed (`cmax`) for explicit time step computation. We focus on the predictor step, `g_intStage=1` and x-direction `g_dir=IDIR` to profile only once the routine.

Figure 19: Zoomed traces for Region 6 of the PLUTO from Paraver showing the structure of the region.

Number of processes	1024	2048	4096	8192	16384
Elapsed time (sec)	0.64654	0.36685	0.196132	0.124846	0.148533
Efficiency	1.0	0.881205	0.824113	0.647338	0.272052
Speedup	1.0	1.76241	3.296453	5.1787	4.352837
Average IPC	2.038645	2.056873	2.033582	1.947460	2.003751
Average frequency (GHz)	1.488987	1.476381	1.498008	1.499206	1.468662

Table 8: Overview of the key performance metrics of the Region 6 of PLUTO.

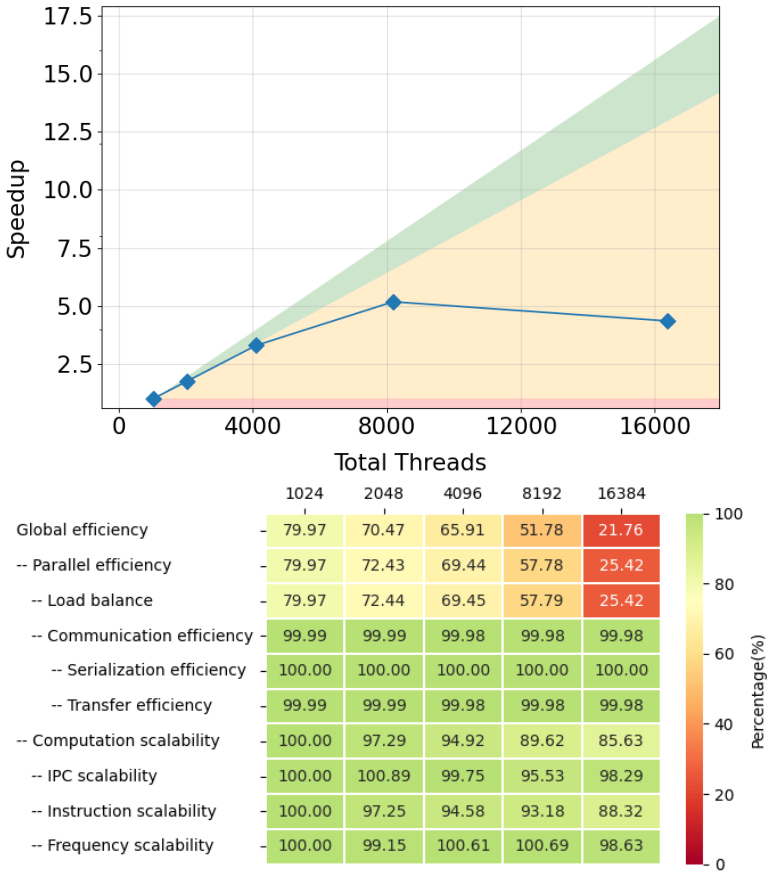


Figure 20: Strong scaling and POP efficiency metrics for Region 6 of PLUTO.

This region is computationally intensive, and it does not contain any communication between MPI processes (Figs. 19 - 20). Therefore, it can be selected for the potential exploitation of advanced vectorization units on new ARM based processors or GPU accelerators. The loss of performance visible at $\gtrsim 800$ threads resembles the same trend already discussed in Region 3 (right hand side computation). This is not surprising as the two functions are called in sequence and both may reflect the unbalanced workload. More work is needed in order to address this shortcoming. This will be taken care of in the next month's activities.

3.6 CT_Update - Region 9

This routine ensures a divergece-free update of the staggered magnetic field in the constrained transport formalism and it is called 3 times per step. It is based on a discrete version of Stoke's theorem.

The update consists of a single Euler step:

$$B_s \Longrightarrow B_s + \Delta t R \tag{14}$$

where `B_s` is the main staggered array used by PLUTO, B_s is the magnetic field to be updated and R is the right hand side already computed during the unsplit integrator.

We focus on the predictor step, `g_intStage=1` to profile only once the routine.

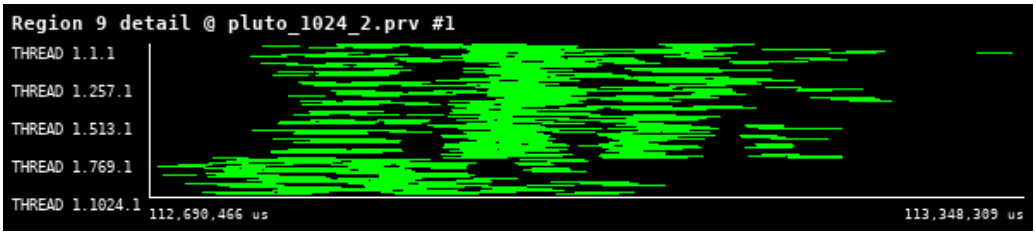
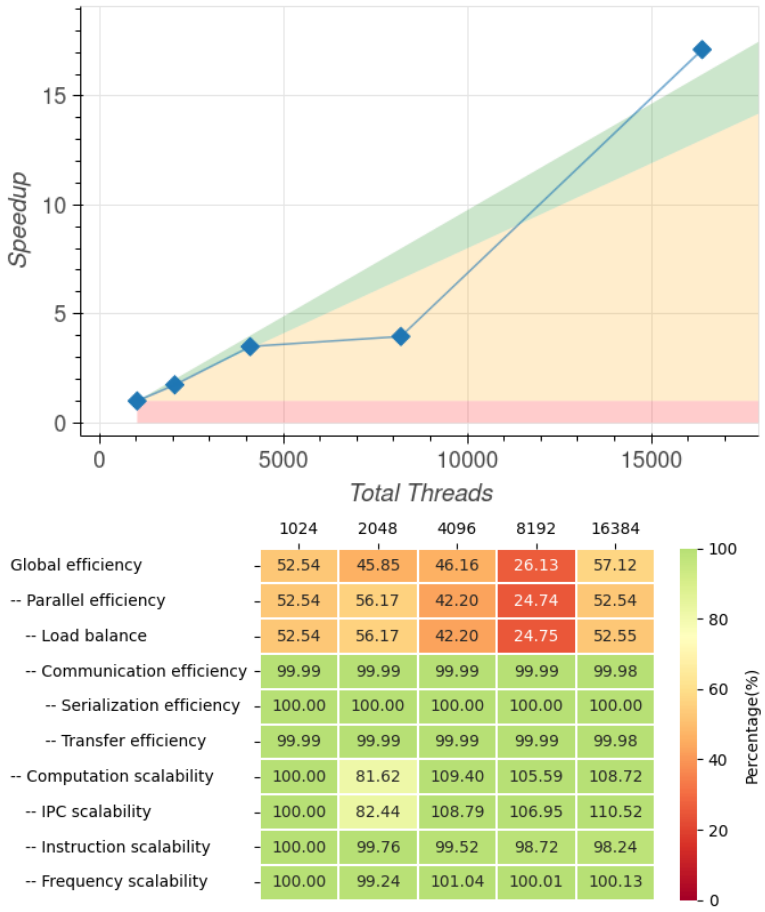


Figure 21: Zoomed traces for Region 9 of the PLUTO from Paraver showing the structure of the region.

Number of processes	1024	2048	4096	8192	16384
Elapsed time (sec)	0.063406	0.036422	0.018114	0.015994	0.0037
Efficiency	1.0	0.870436	0.875097	0.495545	1.071047
Speedup	1.0	1.740871	3.500386	3.964362	17.136757
Average IPC	3.836924	3.163268	4.174214	4.103711	4.240599
Average frequency (GHz)	1.867786	1.853641	1.887301	1.867900	1.870253

Table 9: Overview of the key performance metrics of the Region 9 of PLUTO.



	1024	2048	4096	8192	16384
Global efficiency	52.54	45.85	46.16	26.13	57.12
-- Parallel efficiency	52.54	56.17	42.20	24.74	52.54
-- Load balance	52.54	56.17	42.20	24.75	52.55
-- Communication efficiency	99.99	99.99	99.99	99.99	99.98
-- Serialization efficiency	100.00	100.00	100.00	100.00	100.00
-- Transfer efficiency	99.99	99.99	99.99	99.99	99.98
-- Computation scalability	100.00	81.62	109.40	105.59	108.72
-- IPC scalability	100.00	82.44	108.79	106.95	110.52
-- Instruction scalability	100.00	99.76	99.52	98.72	98.24
-- Frequency scalability	100.00	99.24	101.04	100.01	100.13

Figure 22: Strong scaling and POP efficiency metrics for Region 9 of PLUTO.

This region is only used during the magnetic field update with the constrained transport formalism (see Figs. 21 - 22). Unlike the other regions, this one is strictly multidimensional, and it is called only once per time-stepping stage. For this reason, however, array access is necessarily strided. While this region does not contain communication, it needs to be further analyzed and optimized for faster array access, especially in view of its porting on GPUs architectures.

3.7 Boundary - Region 10

This is a fundamental routine as it contains basically all the process communications and it is called 3 times per step.

The Boundary() function sets both internal and physical boundary conditions on one or more sides of the computational domain. It is used to fill ghost zones of both cell-centred and face-centred data arrays.

The type of boundary conditions at the leftmost or rightmost side of a given grid is specified by the integers `grid[dir].lbound` or `grid[dir].rbound`, respectively. When this value is different from zero, the local processor borders the physical boundary and the admissible values for `lbound` or `rbound` are `OUTFLOW`, `REFLECTIVE`, `AXISYMMETRIC`, `EQTSYMMETRIC`, `PERIODIC`, `SHEARING` or `USERDEF` (in our test problem, outflow boundary are used). Conversely, when this value is zero (internal boundary), two neighbouring processors that share the same side need to fill ghost zones by exchanging data values. This step is done here only for parallel computations on static grids.

We focus on the predictor step, `g_intStage=1` to profile only once the routine.

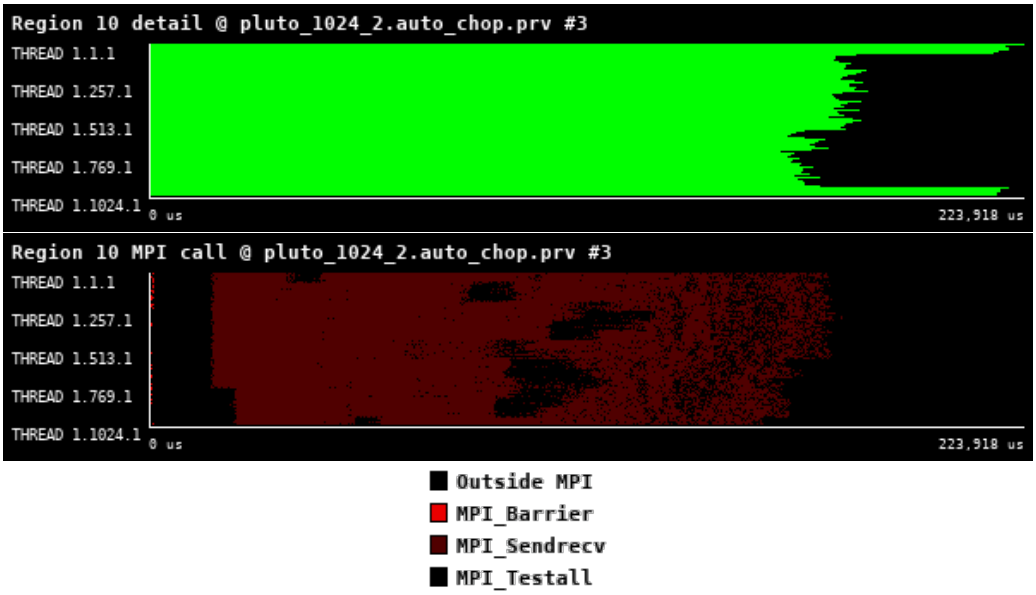


Figure 23: Zoomed traces for Region 10 of the PLUTO from Paraver showing the structure of the region.

Number of processes	1024	2048	4096	8192	16384
Elapsed time (sec)	0.223618	0.183395	0.097785	0.124054	0.05205
Efficiency	1.0	0.609662	0.571708	0.225323	0.268513
Speedup	1.0	1.219324	2.286833	1.802586	4.296215
Average IPC	0.697869	0.804410	0.948088	0.962978	1.297987
Average frequency (GHz)	2.065552	2.064866	2.062466	0.478057	2.063176

Table 10: Overview of the key performance metrics of the Region 10 of PLUTO.

This document is Public (PU)
and was produced under SPACE project EU GA 101093441

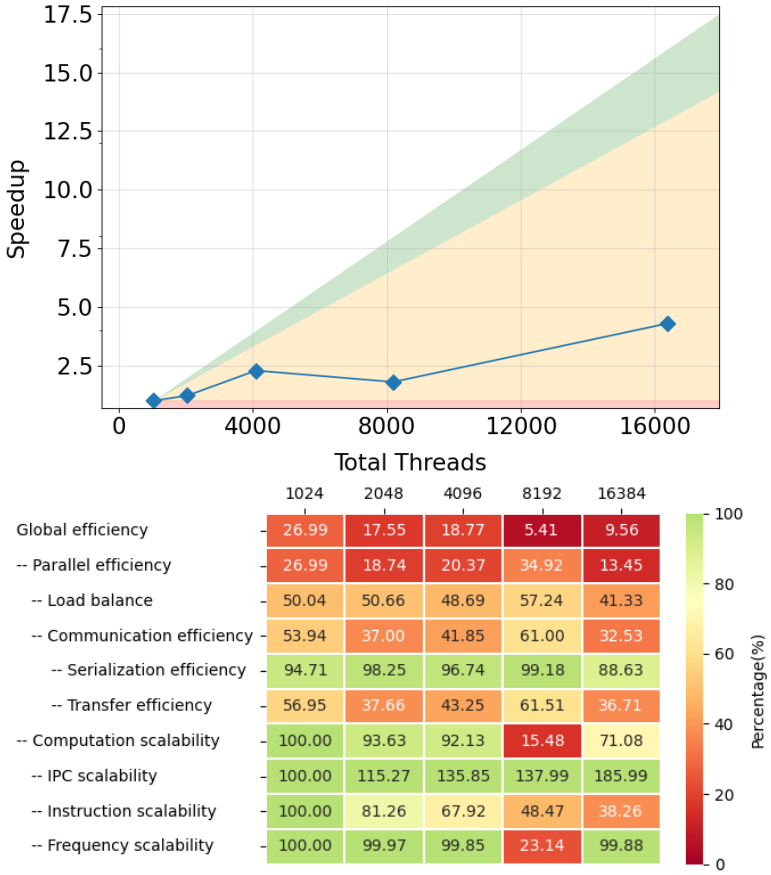


Figure 24: Strong scaling and POP efficiency metrics for Region 10 of PLUTO.

The loss of performance observed for more than $\gtrsim 200$ threads (see Figure 24) may be caused by an incorrect workload already present during the initial MPI domain decomposition phase. Note that this region is where most of the inter-CPU or inter-node data communication takes place. This region is based on the message passing interface (MPI) protocol. While in previous versions of the code the data exchange was based upon MPI derived data types, it has here been rewritten using more basic operations, based on standard buffer copy, followed by `Send/Recv` call and the copy back onto the main array. In terms of debugging and profiling, this function is the one with the highest priority.

3.8 Conclusion

Performance analysis on PLUTO shows a good scaling but it also highlights important aspects to look at. The critical aspect of every region seems to be the "Load Balance". The reason behind the low values of it need to be investigated.

Regions #03 (Right hand side) and #06 (Riemann solver) are computationally intensive although they do not contain communication between CPUs. Both can therefore be selected for potential exploitation of advanced vectorization units or accelerators. The loss of performance visible at $\gtrsim 800$ threads indicates a strong workload unbalance. More work is needed in order to address this shortcoming. This will be taken care of in the next month's activities.

The `CT_Update()` (region #09) is used during the magnetic field update with the constrained transport formalism. Unlike the other regions, this one is strictly multidimensional and it is called only once per time-stepping stage. For this reason, however, array access is necessarily strided. While this region does not contain communication it sill needs to be further analyzed and optimized for faster array access, specially in view of its porting on GPUs architectures.

This boundary region (#10) is based on the message passing interface (MPI) protocols. While in previous versions of the code the data exchange was based upon MPI derived data type, it has here been rewritten using with more basic operation, based on standard buffer copy, followed by `Send/Recv` call and the copy back onto the main array. Region 10 analysis highlight possible critical issues in "Transfer Efficiency". This region requires special attention containing all inter-process communication.

In terms of debugging and profiling, this function is the one with the highest priority.

4 OpenGadget

OpenGadget is a code for cosmological simulations. It solves the gravitational and hydrodynamical equations that rule the formation and evolution of cosmic structures.

It computes the gravitational forces with a hierarchical tree algorithm in combination with a particle-mesh scheme for long-range gravitational forces. Then the fluid flows are computed using smoothed particle hydrodynamics (SPH) or Meshless Finite Mass (MFM). In addition, OpenGadget contains multiple sub-modules to describe various physical processes (e.g. radiative cooling, star formation, stellar feedback, magnetic fields, black holes, just to name the major ones) that shape the fundamental properties of baryons in the Universe.

OpenGadget can be used for a wide variety of astrophysical problems, ranging from colliding and merging galaxies to the formation of large-scale structure in the Universe. It can also be used to study the dynamics of the intergalactic medium, or star formation and its regulation by feedback processes.

OpenGadget originally evolved from the publicly available Gadget2 code [10]. OpenGadget has been improved significantly compared to its base version by, e.g. adding a new state-of-the-art SPH implementation [11], a MFM solver [12] as well adding OpenACC support to run on GPUs [13].

4.1 Use-case description

In this first performance evaluation, cosmological boxes of different sizes have been used. As the denomination suggests, cosmological boxes are built as fair samples of the Universe. In these boxes, the matter distribution and clustering are substantially homogeneous and their statistical properties are aimed to be similar to those of the Universe (at the scale of interest). The fact that the cosmological structures (e.g.: galaxies and clusters of galaxies) are homogeneously distributed in the computational domain renders this problem more easily balanced in terms of workload. These cosmological boxes (hereafter simply "box") are characterized by two fundamental parameters: (i) the physical size of the volume it represents, which is commonly expressed as the edge length of the box in Megaparsec (Mpc) and (ii) the number N_p of particles that are used to sample the distribution of matter within the volume. The last number is commonly referred to as $N_p = 2 \times N_G^3$, where N_G is the grid number used to generate the particles in the initial conditions, and the factor $\times 2$ descends from the fact that we include both ordinary baryonic and dark matter particles.

The physical size of the box affects the number and size of sub-structures, such as galaxies and clusters of galaxies, which are simulated. In general, it can be said that the larger the volume the larger the number of structures and the maximum size of those sub-substructures. The number of particles, instead, affects the mass resolution; in other words, the larger the number of particles used, the higher the accuracy in both following the dynamics and describing the internal structures of collapsed objects.

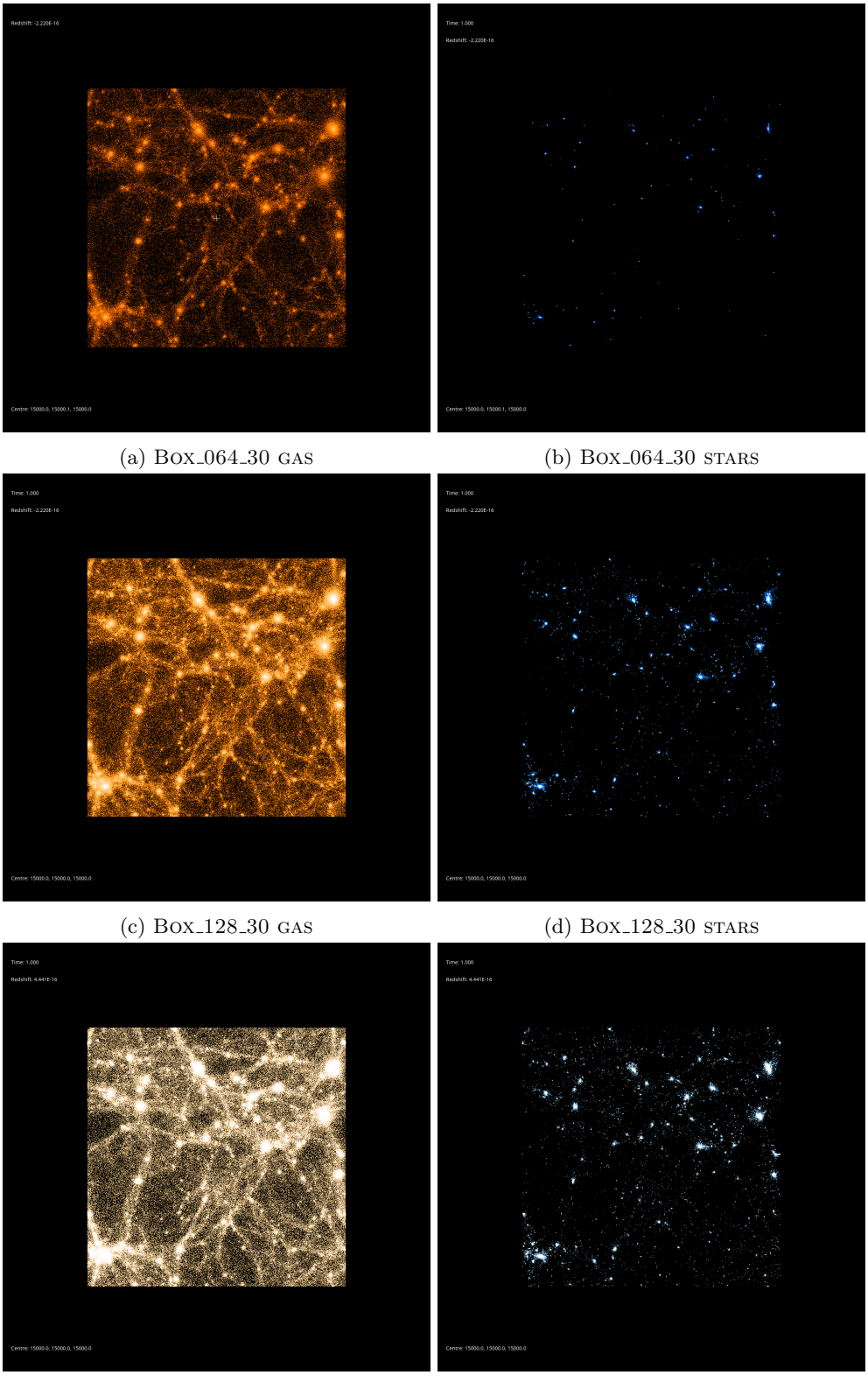
For this study, we have conducted simulations of a set of boxes whose parameters are details in Tab. 11.

Box size/edge length / N_p	64^3	128^3	256^3	800^3
30 Mpc	Box_064_30	Box_128_30	Box_256_30	–
60 Mpc	–	Box_128_60	–	–
120 Mpc	–	–	Box_256_120	–
375 Mpc	–	–	–	Box_800_375

Table 11: The set of cosmological boxes generated for the profiling activity along the project. The boxes are referred to as Box_N_p_size where the "size" refers to the side length of the box.

Figure 25 and 26 show a projection of the gas and stellar distribution, coloured by density, at redshift 0 (i.e. after $\sim$ 13 billion years of evolution). Here the simulated volume remains constant, while the accuracy and resolution are increasing with the number of particles N_p. As the number of particles increases, the computational cost of simulating the same region increases too. This increase does not scale linearly due to the costs related to the calculation of the gravitational tree, which scales with the number of particles as $N\,logN$. We therefore expect the computation to be more demanding by a factor of minimum $\gtrsim 8$ while moving from the Box_064_30 to Box_128_30.

Enlarging the number of particles does not only have an effect on the calculation of the gravitational tree but also on the frequency of its calculation. As a larger number of particles results in a more accurate description of forces between particles, the length of time steps has to be reduced. This results in a more expensive calculation as a larger number of time steps has to be calculated to run the simulation for the same global time. It has been found empirically – since it is difficult to draw theoretical expectations for such a complex code –, that an

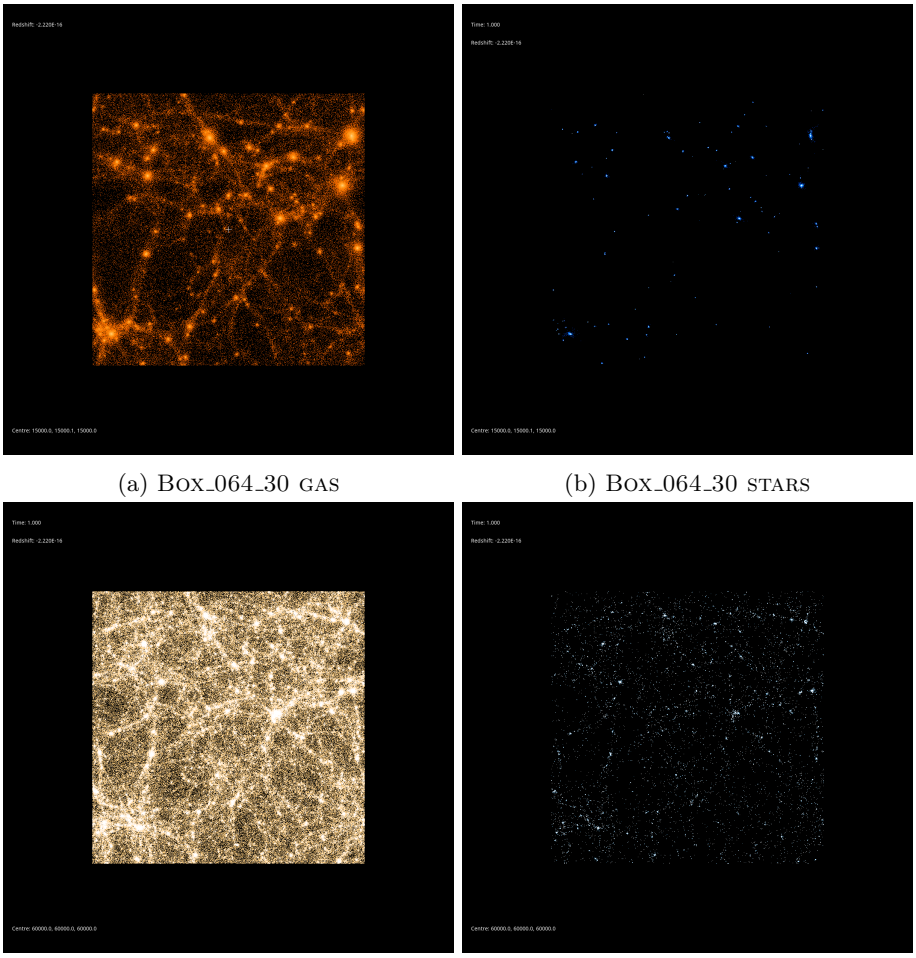


(a) Box_064_30 gas (b) Box_064_30 stars

(c) Box_128_30 gas (d) Box_128_30 stars

(e) Box_256_30 gas (f) Box_256_30 stars

Figure 25: Projection of gas (left column) and stellar (right column) distribution at $z \sim 0$ (i.e. at the present time, after $\sim$ 13 billion years of evolution) in the 30 Mpc boxes. Each row shows the simulation of the same box resolved with a different number N_p of particles: 64^3, 128^3 and 256^3 in the top, medium and bottom rows respectively. As the level of details increases from top to bottom, i.e. with increasing N_p, the resolution increases as well.



(a) Box_064_30 gas (b) Box_064_30 stars

(c) Box_256_120 gas (d) Box_256_120 stars

Figure 26: Projection of the gas (left column) and stellar (right column) distribution at $z \sim 0$ (i.e. at the present time, after $\sim$ 13 billion years of evolution) in the 64^3,30 Mpc and 256^3, 120 Mpc boxes (top row and bottom row respectively), where lighter colour indicate a stronger matter density. While the box size remains constant the number of particles N_p increases from top to bottom, by which the level of detail increases simultaneously.

increase in the number of particles by a factor of 2, results in a ~ 12 times longer run time. Hence, we expect that the computational cost of Box_128_30 and Box_256_30 are $\sim \times 12$ and $\sim \times 144$ larger with respect to Box_64_30.

With the boxes defined above, we are able to study the Strong Scaling, where we have defined two different cases:

1. small case: Box_256_30 (2×256^3 initial particles) run with [1:128] nodes. This test may be affected by the parallel overhead with more than 16 nodes because of the relatively small amount of particles per node;

2. large case: Box_800_375 (2×800^3 initial particles) run with [16:256 nodes]. This test has been selected for testing between 16 and 64 compute nodes in the following sections.

It is also important to note that the same scaling comparison should be conducted at different cosmic epochs as the clustering of the matter is strongly different at different redshifts, becoming much more significant at present times compared to earlier times. That translates into two basic effects: (*i*) the average distance of particles in collapsed structures is smaller. This results in larger forces, which requires a more accurate integration and therefore smaller time-steps; (*ii*) the gravitational tree is in average much deeper and the tree-related operations become more costly.

In this document, for a matter of simplicity, we focus on the case of the Strong Scaling using the small boxes at $z \approx 0$, where the clustering of the matter is the largest. We were not able to simulate the 800^3 case down to the relevant cosmic epochs, due to the limited run time provided.

4.2 High-level code structure

In simple terms the code consists of two parts. Firstly the initialisation, which consists of reading initial conditions , reading the parameter file and initializing the required physical modules. And secondly the body of the code, an "infinite" time loop. Each of these time steps consists of a series of subsequent calculations: the estimate of the gravitation acceleration, of the hydrodynamical acceleration and of the "extra-physics" (with which we indicate all the physical processes for baryons, like the radiative cooling the star-formation, the stellar feedback, the black-holes feedback,...). In addition, the domain decomposition, which amounts to the re-distribution of particles to keep the work as balanced as possible, is performed depending on the evaluation of some global and local conditions. OpenGadget's main function is based on this while loop iterating through the time steps until the end time is reached. For the performance test we have chosen to concentrate on the high-level sections that are responsible for the majority of the computation and communication. A simplified view, including the profiling regions, is shown in Figure 27.

```
int main(){
    do_initialisation();

    while(1){
        EXTRAE_EVENT_START( NumCurrentTimeStep ) // step region, or region 0

        write_snapshot_if_desired();

        if (Time >= maxTime){
          write_snapshot();
          break;
        }

        find_timesteps();

        do_first_halfstep_kick();

        find_next_sync_point_and_drift();

        // compute gravitational forces
        EXTRAE_EVENT_START( EXTRAE_REG_DD1 ) // region 1
        domain_decomposition_intensity_decision();
        EXTRAE_EVENT_STOP( EXTRAE_REG_DD1 )

        set_non_standard_physics_for_current_time();

        EXTRAE_EVENT_START( EXTRAE_REG_DD2 ) // region 2
        domain_decomposition_intensity_execute();
        EXTRAE_EVENT_STOP( EXTRAE_REG_DD2 )

        EXTRAE_EVENT_START( EXTRAE_REG_GRAV ) // region 3
        compute_grav_accelerations();
        EXTRAE_EVENT_STOP( EXTRAE_REG_GRAV )

        // compute fluid flows
        EXTRAE_EVENT_START( EXTRAE_REG_DENS ) // region 4
        compute_densities();
        EXTRAE_EVENT_STOP( EXTRAE_REG_DENS )

        EXTRAE_EVENT_START( EXTRAE_REG_HYDRO ) // region 5
        compute_hydro_accelerations();
        EXTRAE_EVENT_STOP( EXTRAE_REG_HYDRO )

        do_second_halfstep_kick();

        // calculate additional physics
        EXTRAE_EVENT_START( EXTRAE_REG_PHYS )  // region 6
        calculate_non_standard_physics();
        EXTRAE_EVENT_STOP( EXTRAE_REG_PHYS )

        EXTRAE_EVENT_STOP(GET_EVENT_NUMBER( All.NumCurrentTiStep-1 ))
    }
}
```

Figure 27: Simplified high-level code structure of OpenGadget.

Figure 28: Traces for all regions in Gadget from Paraver showing the high-level structure of the code.

4.3 Timestep - Region 0

This region contains the whole computation of a single time step. This includes the writing of snapshots (even though we avoided performing I/O in the present tracing activity), the gravity and hydro calculations as well as the calculation of additional physics. It also includes some activities that we are not tracing explicitly as single regions: the determination of the individual subsequent time-step for the particles, the half kicks and the drift of particles, among some minor others. Embedded within this region are the EXTRAE regions 1 to 6 (see next sections for a detailed description of each region).

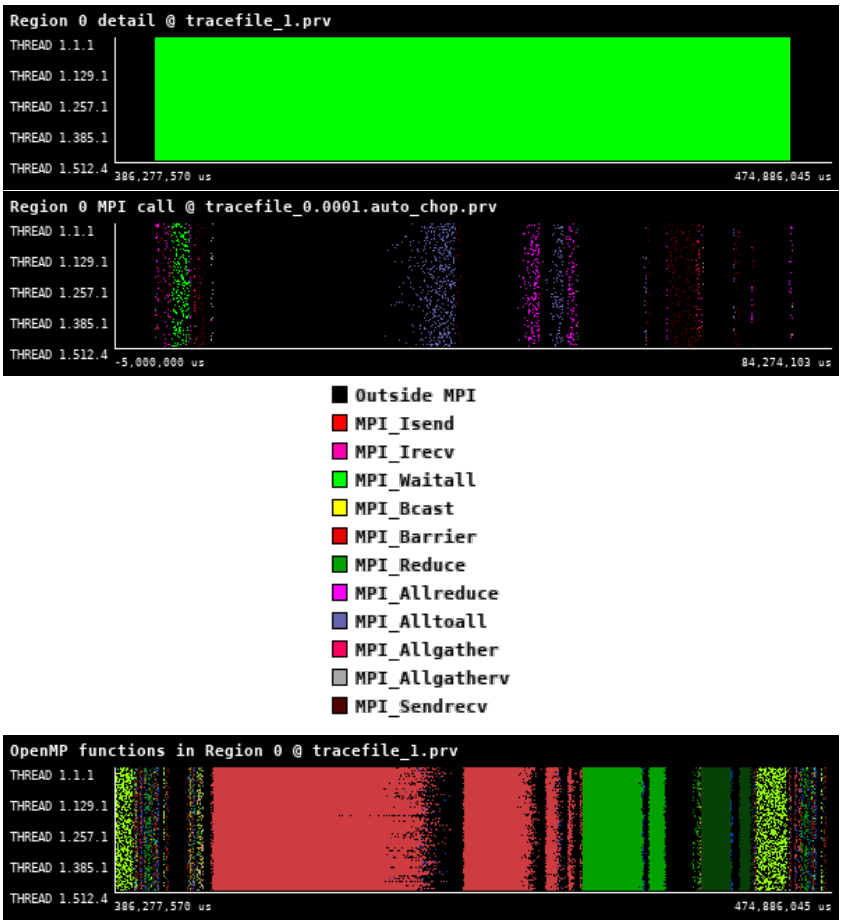


Figure 29: Zoomed traces for Region 0 of the Gadget from Paraver showing the structure of the region. For OpenMP timeline the different colours denote different OpenMP parallel regions or functions.

By its very nature, this "macro-region" encompasses

1. All the communications performed along a simulation run (except for the initialization part that we are omitting). The main communication patterns involved are the following:
   - broadcasts/synchronization; these are communications involving few bytes that are mostly about propagating/collecting values with one-to-may, many-to-many, many-to-one patterns to ensure synchronization.
   - P2P sendrecv; these are communications mostly performed in the tree-related routines. They happen both in the domain decomposition and, above all, in the tree-walk that is exploited for the neighbour search, which in turn is an ubiquitous task.
2. All the calculations performed in a simulation run. There is a huge diversity of math operations with very different arithmetic intensities and vectorization potential.

- The Gravitational part basically consists of accumulating $\frac{m_j}{r^2}$ terms over $j = 1, .., N$ neighbours each with mass m_j, or from multi-polar expansion of distant tree nodes. The force from most distant particles is calculated with a Particle-Mesh and hence via FFT.

  We remind the reader that a Particle-Mesh algorithm consists in the following steps:

  (a) spreading of particles' relevant properties over a grid; in this case, that amounts to obtaining the density field ρ over the grid.

  (b) solving the equations using FFT over the grid; in this case, we solve the Poisson equation $\Delta\Phi = 4\pi G\rho$, where Φ is the gravitational potential.

  On the other side, the hydrodynamical force is significantly more complex from the computational point of view. First, it needs to

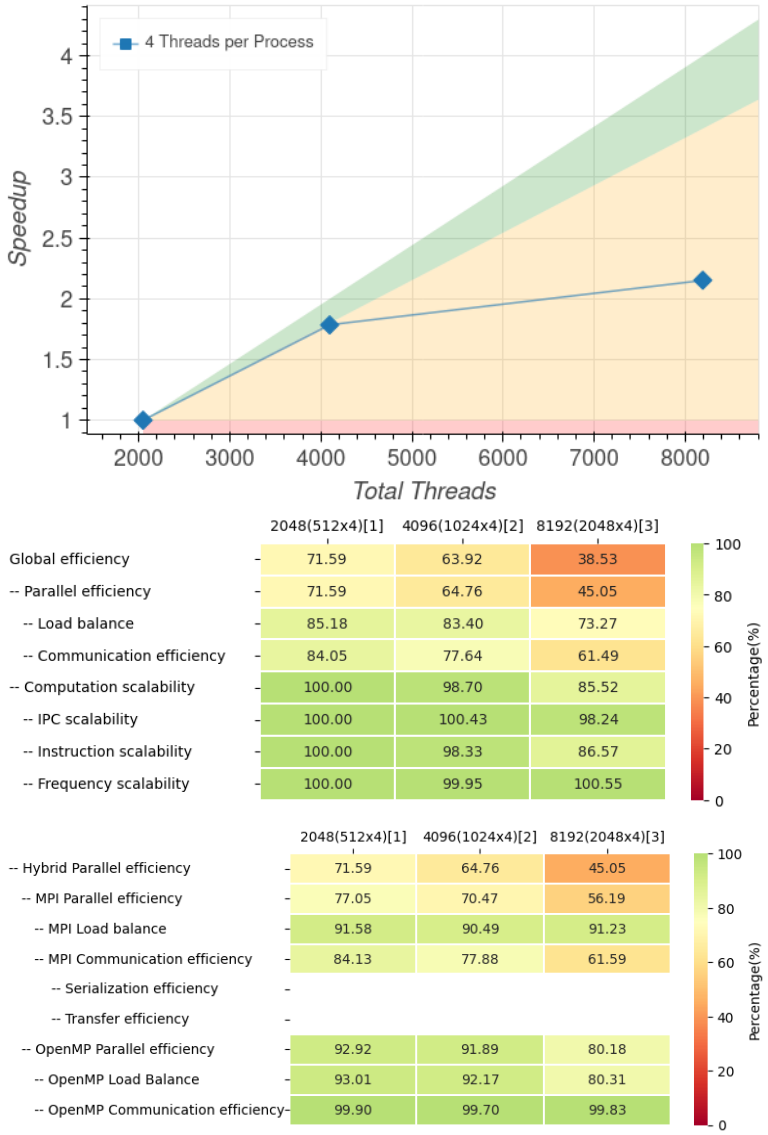


Figure 30: Strong scaling and POP efficiency metrics for Region 0 of Gadget.

Number of processes	2048	4096	8192
Elapsed time (sec)	79.273902	44.39593	36.827857
Efficiency	1.0	0.892806	0.538138
Speedup	1.0	1.785612	2.152553
Average IPC	1.131789	1.136640	1.111919
Average frequency (GHz)	3.186542	3.184966	3.204001

Table 12: Overview of the key performance metrics of Region 1 of Gadget.

We have selected a timestep where the domain decomposition occurs (see Regions 1 and 2, Section 4.4 and 4.5 respectively). The overall time-step execution is shown in Figure 29. The POP metrics and scalability are shown in Figure 30 and Table 12. Overall, the problems that limit the scalability identified here are the MPI communication efficiency and load balance, which can be mostly ascribed to the domain decomposition. We describe the causes of limited scalability in the following sections.

4.4 Region 1 - Domain decomposition: intensity decision

The domain decomposition is responsible for distributing the workload across the different tasks. This is done using a Hilbert space-filling curve, where multiple segments of the curve, each corresponding to a region of space, are assigned to one MPI process. The assignment accounts for the estimated computational costs of the particles that lie in each segment of the curve: sorting the segments, that have equal memory occupation, by their computational cost allows distributing them to the MPI tasks achieving a good work balance (at the cost of some load imbalance). The larger the number of segments, the more accurate the achievable balance. Actually, since the domain decomposition may be quite expensive, the code tests whether smaller adjustments could be made to the current domain decomposition; in practice, whether re-assigning particles that moved into the domain of another MPI task may be sufficient to balance the workload. This region decides if the domain decomposition needs to be done or not, or whether a small adjustment is in order.

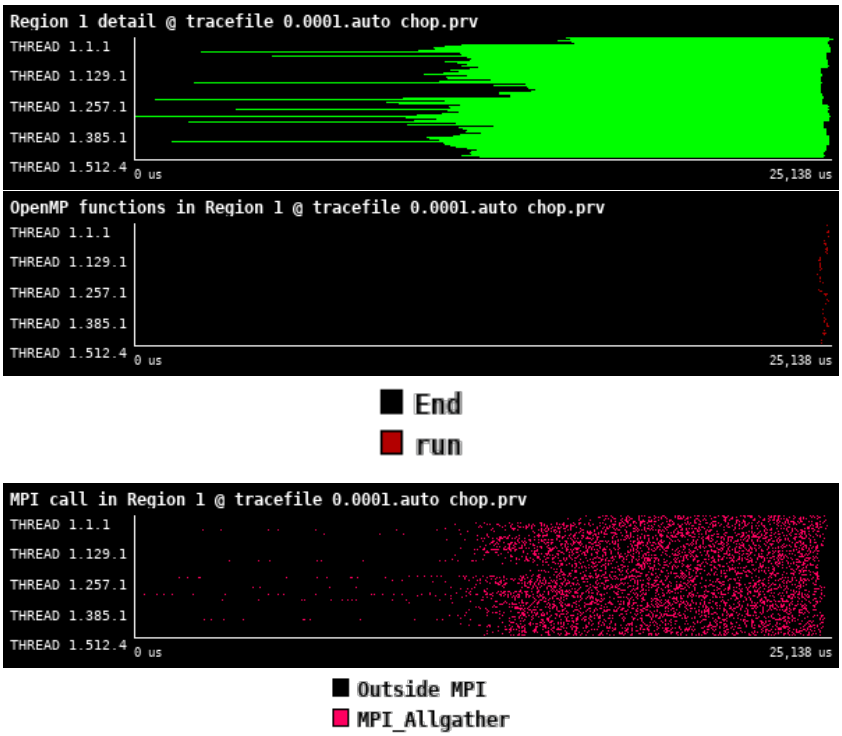


Figure 31: Zoomed traces for Region 1 of the Gadget from Paraver showing the structure of the region.

Number of processes	2048	4096	8192
Elapsed time (sec)	0.024979	0.016295	0.009519
Efficiency	1.0	0.766462	0.65603
Speedup	1.0	1.532924	2.62412
Average IPC	1.367154	2.011667	2.399641
Average frequency (GHz)	0.554462	0.900147	1.111310

Table 13: Overview of the key performance metrics of Region 1 of Gadget.

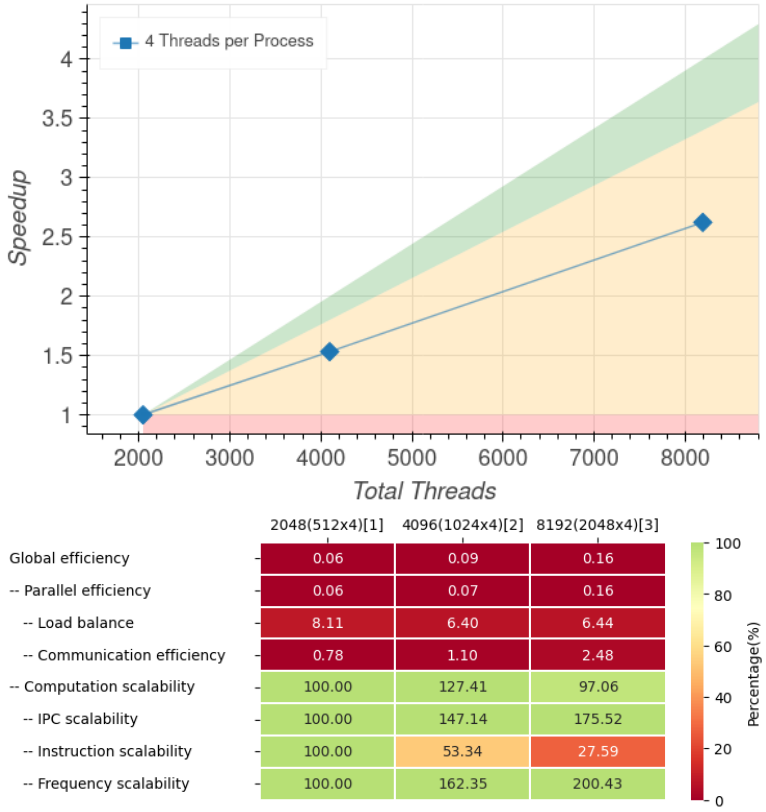


Figure 32: Strong scaling and POP efficiency metrics for Region 1 of Gadget.

As expected, this is a very short region which mostly contains only communication and almost no computation. In fact, it amounts to verifying what particles moved out of the domain of their owner task and by how much.

The poor instruction scalability means that useful calculations (user code processing) do not scale: in other words, the amount of work per process remains almost the same. That could be interpreted as the consequence that even if the number of MPI tasks is increased, and hence the number of particles per task decreases, the average communication surface remains the same because the domain decomposition is made by multiple chunks of the Peano-Hilbert curve.

High load imbalance in this case is caused by the fact that the region itself is short and the useful run-time is extremely short.

The low communication efficiency comes from the fact that all the threads spend most of the time in communication routines for small messages.

All in all, the main message from this analysis is that the communication pattern, with many multiple calls to an all-to-all collective, should be replaced by a more efficient one.

4.5 Region 2 - Domain decomposition: intensity execute

In this region, the domain decomposition is executed and afterwards the force-tree is updated.

```
domain_decomposition(){
    // build top-tree and produce the sequence of leaves;
    determineTopTree();
    // split the leaves into balanced domains according to work
    findSplit_work_balanced();

    // assign domains to MPI processes;
    call assign_load_or_work_balanced();

    if (domain decomposition violates memory constraints){
        // split the leaves into balanced domains according to load
        findSplit_load_balanced();
        // assign domains to MPI processes;
        assign_load_or_work_balanced();
    }

    while(particles on different process than assigned){
        // select as many particles as possible/neccessary for export within memory constraints
        countToGo();
        // send particles to respective process
        exchange();
    }
}

force_update_tree(){
    for(all active particles){
        // kick the parent nodes with this momentum difference
        force_kick_node();
    }

    // finish kicking nodes
    force_finish_kick_nodes();
}
```

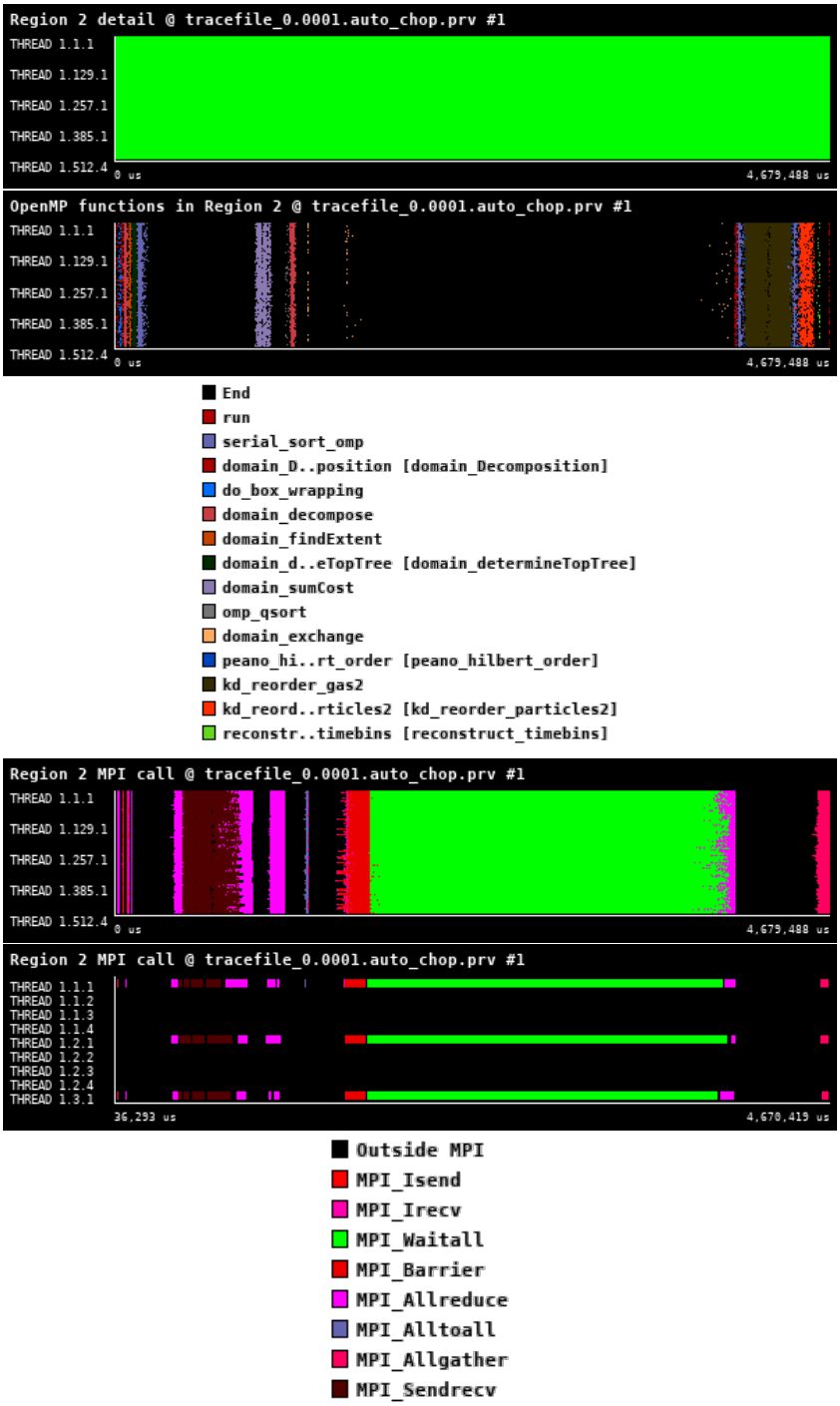


Figure 33: Zoomed traces for Region 2 of the Gadget from Paraver showing the structure of the region.

Number of processes	2048	4096	8192
Elapsed time (sec)	4.678906	3.046684	6.629154
Efficiency	1.0	0.767868	0.176452
Speedup	1.0	1.535737	0.705807
Average IPC	0.624207	0.634947	0.624791
Average frequency (GHz)	3.206817	3.187740	3.205016

Table 14: Overview of the key performance metrics of Region 2 of Gadget.

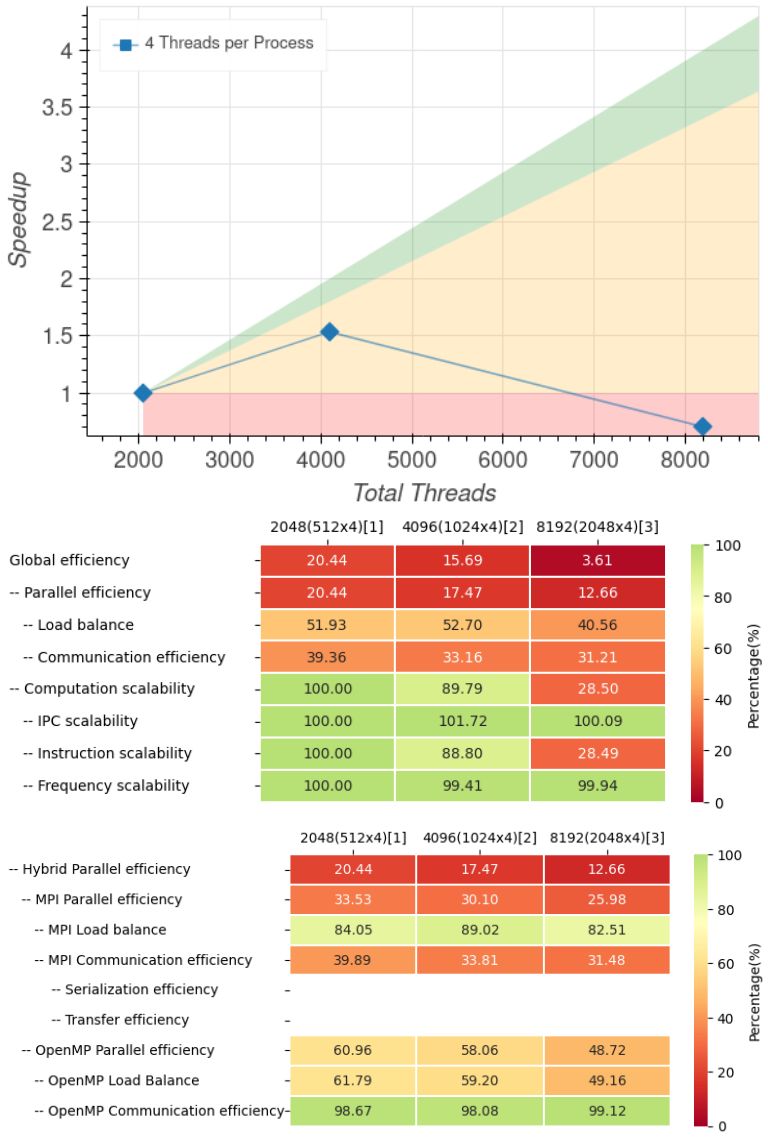


Figure 34: Strong scaling and POP efficiency metrics for Region 2 of Gadget.

The reason for low communication efficiency is that over 50% of the runtime is spent on `MPI_Waitall`.

This code section is not fully threaded: in fact, the OpenMP threads are not utilized when `MPI_Waitall` is running which is the reason for the limited OpenMP load balance This region is clearly dominated by communication, as expected. The insufficient threadization determines a very poor performance: in fact, during the MPI operations, done only by the master thread, the other CPU cores are idling. Here the obvious improvement is to enhance the work-sharing among the threads so that the master thread could perform communication of small data chunks while the other threads are processing. When carefully designed, a new scheme will also cure the large amount of time spent in the `MPI_Waitall`.

4.6 Region 3 - Compute gravitational accelerations

In this region, the gravitational accelerations for all synchronous particles are computed. If needed, a new tree is constructed; otherwise, the dynamically updated tree is used. When needed, relevant data of particles are communicated among MPI tasks via point-to-point communications.

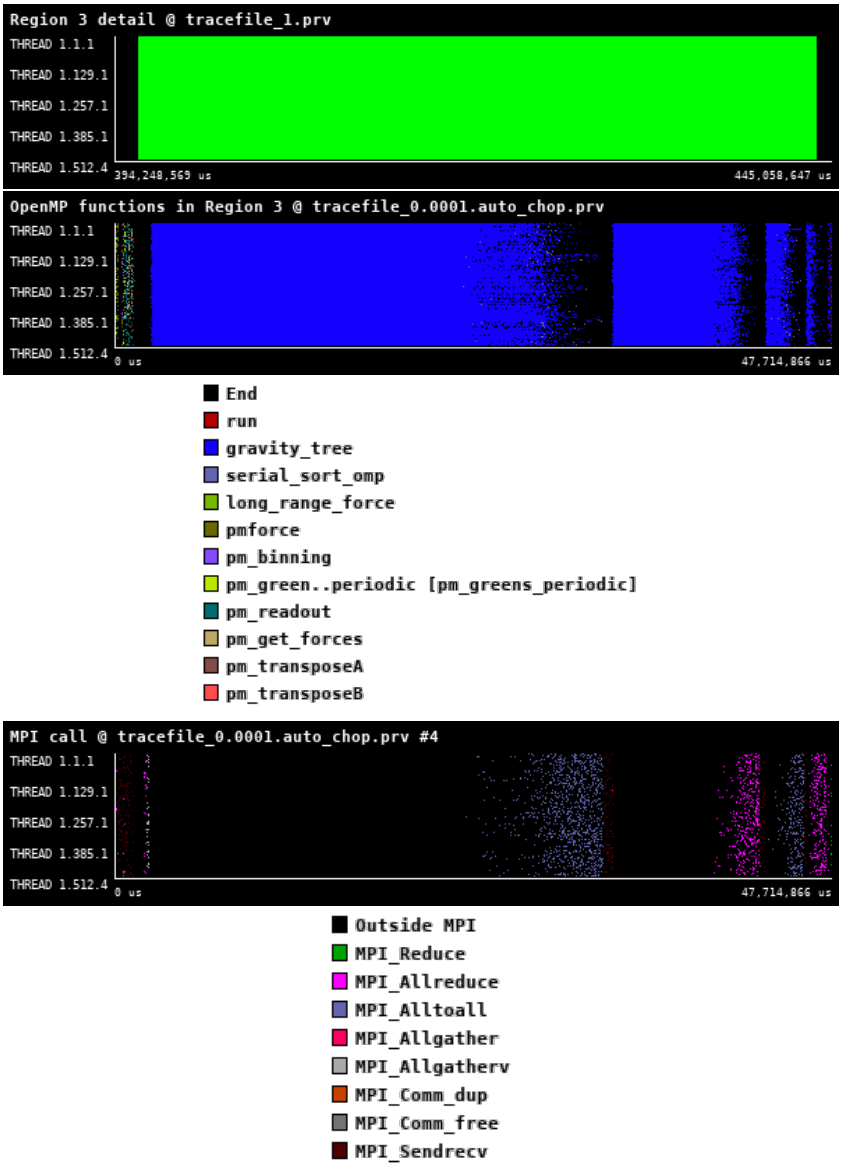


Figure 35: Zoomed traces for Region 3 of the Gadget from Paraver showing the structure of the region.

Number of processes	2048	4096	8192
Elapsed time (sec)	47.714394	25.446344	18.755917
Efficiency	1.0	0.937549	0.635991
Speedup	1.0	1.875098	2.543965
Average IPC	0.961925	0.970340	0.987195
Average frequency (GHz)	3.190112	3.187294	3.207830

Table 15: Overview of the key performance metrics of Region 3 of Gadget.

This region includes three different computations:

1. direct summation; close particles' contributions to the gravitational force on the target particles are directly summed as $\sum m_j/r_{ij}^2$.
2. distant tree contributions; distant particles' contribution is taken into account in approximate form as multipolar expansion of an equivalent mass distribution.
3. long-range forces; the contribution from very distant particles is accounted for via a Particle-Mesh algorithm.

The stages 1 and 2 involve the walking of the tree for contributions of particles (or tree nodes in case 2) from other MPI tasks. Step 3, instead, involves an FFT that is handled via the `fftw` library.

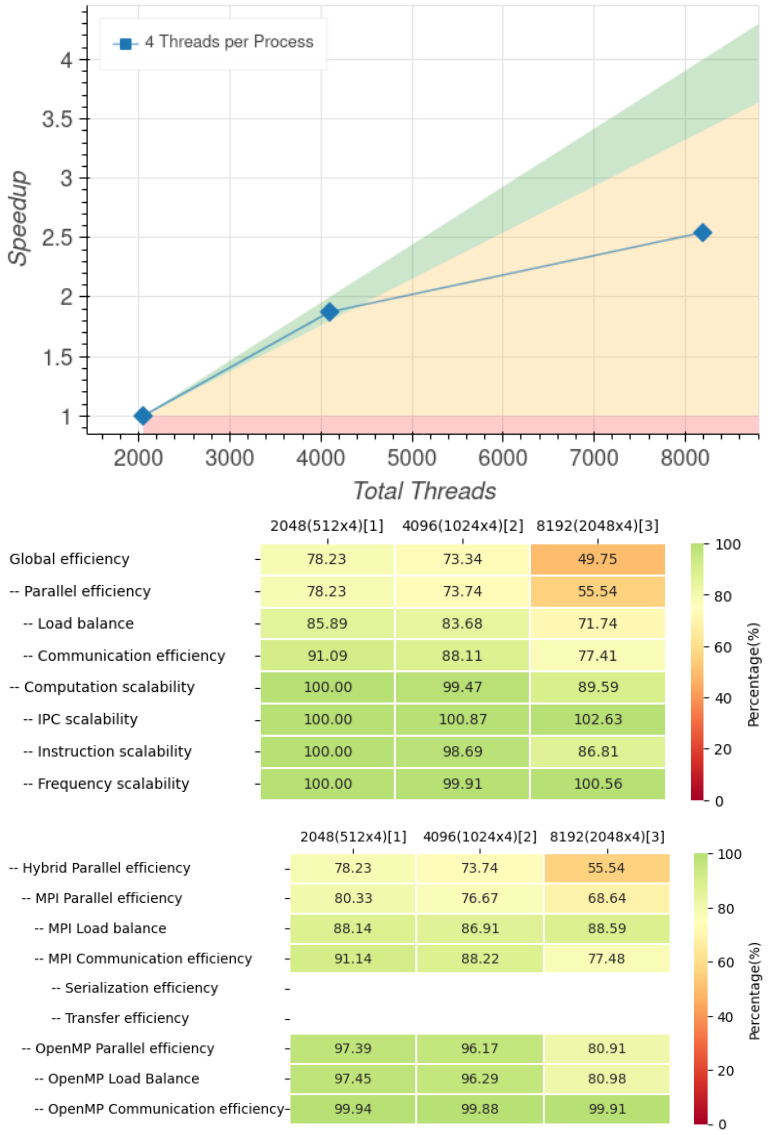


Figure 36: Strong scaling and POP efficiency metrics for Region 3 of Gadget.

A fact is that the `ftw` library can decompose a domain only by 2d slabs; since, in the case under exam, the grid size is 800^3, then the number of slabs computed in parallel is 800. That means that only 800 MPI tasks are actually participating in the `FFT`. This may explain, at least partially, the poor parallel scalability in the third measure ($800/2048 MPI \simeq 40\%$).

Focusing specifically on the tree performance will require a dedicated set of profiling runs that exclude the PM algorithm.

4.7 Region 4 - Compute densities

In this region, the local densities for all synchronous SPH (Smoothed-particle hydrodynamics) particles are computed as well as the number of neighbours in the current smoothing radius, and the divergence and rotation of the velocity field. The local density of each gas particle is defined as $\sum_0^{N_{sph}} W(m_j, h_{ij})$, where $W()$ is the SPH smoothing kernel and h_j is the normalized distance of each of the $j = 1, .., N_{sph}$ neighbours (note that $N_{grav} \neq N_{sph}$). The neighbours of a gas particle are defined as other gas particles lying in a spherical region that encompasses N_{sph} particles. Hence, finding the density requires individuating the mentioned spherical region which in turn is an iterative process that starts from a guess of its radius h and updates it until the sphere of radius h encompasses N_{sph} gas particles. While accumulating the local density results, all the physical properties whose sph-value is defined using the density ρ as a weight function are estimated. Those properties vary depending on the active physical modules in a given run.

This phase involves point-to-point communications among the MPI tasks whenever the neighbour particles of a given target particle are not in the computational domain of the MPI task that hosts that same target particle.

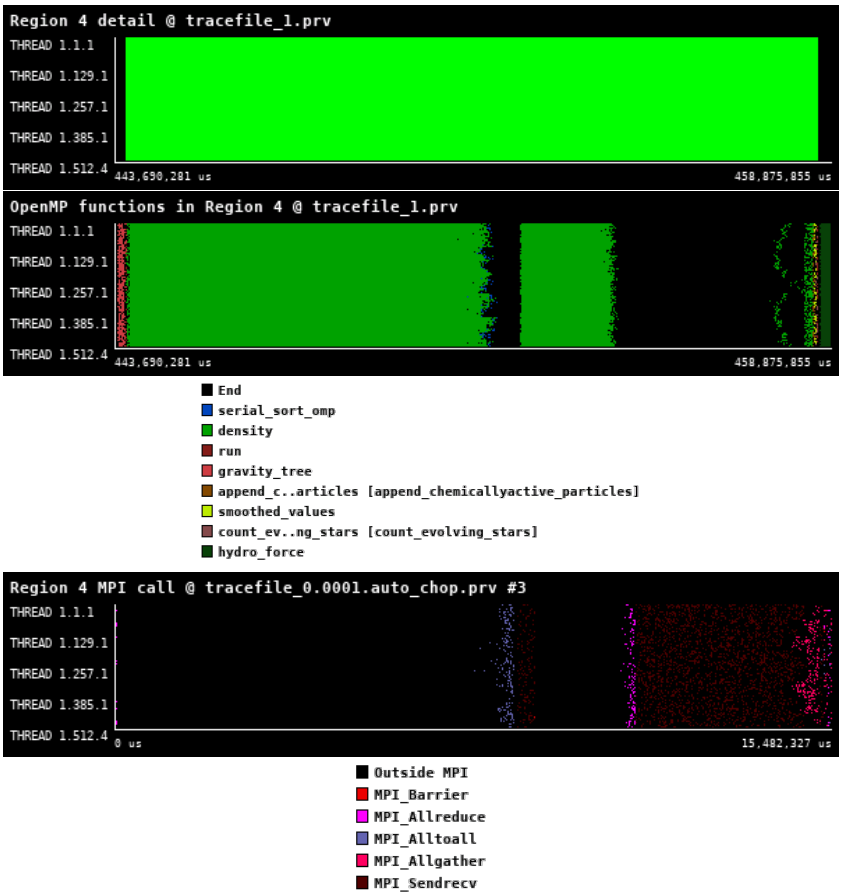


Figure 37: Zoomed traces for Region 4 of the Gadget from Paraver showing the structure of the region.

Number of processes	2048	4096	8192
Elapsed time (sec)	15.482099	9.667074	7.004041
Efficiency	1.0	0.800764	0.552613
Speedup	1.0	1.601529	2.210452
Average IPC	1.490568	1.485078	1.439259
Average frequency (GHz)	3.220052	3.218160	3.225717

Table 16: Overview of the key performance metrics of Region 4 of Gadget.

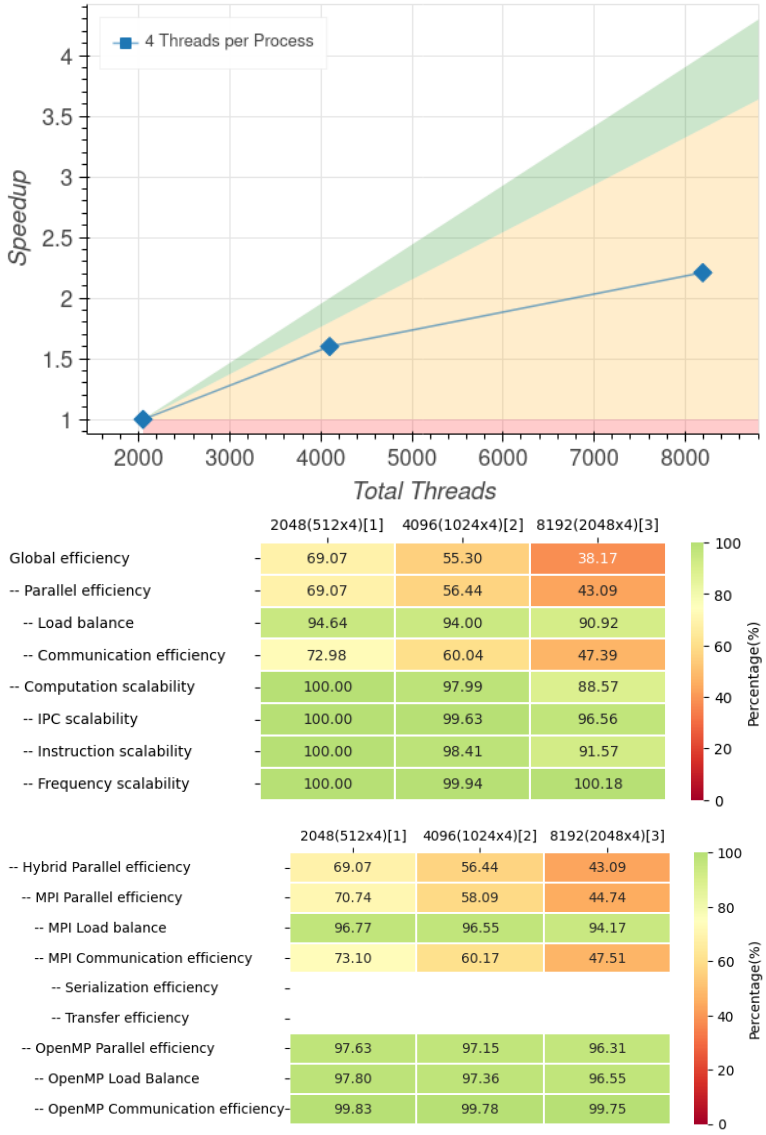


Figure 38: Strong scaling and POP efficiency metrics for Region 4 of Gadget.

The region's structure shown in Figure 37 clearly exhibits the typical expected pattern for a homogeneous matter distribution (as we have at early Universe epoch like the one that we are tracing. Note: the sampling has actually caught the tail of the previous step, the estimate of gravitational forces, that appears at the beginning as red dots; and the successive phase, the hydro forces, that appears at the end as dark green dots. We just ignore those tails in the following comments). In fact, there are 3 green sections (labelled "density"), separated by MPI regions. The first green section corresponds to the local processing of neighbours and to the collection of the communication matrix. It is followed by an MPI section made of `MPI_Sendrecv` and some collectives, in which data are point-to-point exchanged among MPI tasks. Then a second green "density" section corresponds to the accounting for data from the neighbours resident in different MPI tasks. After that, another communication round follows, to refine the tail of particles that did not yet converge to the desired number of neighbours. A final accounting for these last data then leads to the evaluation of physical quantities that use the local density as the weighting function (the pale green dots, labelled "smoothed_values").

In this region the computation, which is completely local and depends on the number of particles per task, is scaling almost perfectly. The key issue is the MPI communication efficiency. This means that processes that spend the least amount of time in communication still spend over 50% (for 8192 cores) in communication routines. We may speculate that the efficiency could be even worse for later epochs when the distribution of the gas is much more clumpy and the domain decomposition is possibly more inclined to favour the work

balance instead of load balance. It is difficult to disentangle at this stage of the analysis the possible sources of inefficiencies and the simple fact that this problem is increasingly communication-bound as the number of local particles decreases; that is because the communication surface then increases (more particles per each MPI task will have neighbours belonging to other MPI tasks). This may advocate for a different ratio of MPI tasks / OpenMP threads; in fact, the OpenMP efficiency is good, and increasing the number of OpenMP threads would make the number of MPI tasks smaller; then, the domains would be proportionally larger and the communication surface smaller for the same number of total used cores.

While we will inquire more in detail about possible sources of inefficiency in this region, we also conclude that a fundamental result is a confirmation that "strong scaling" a problem requires changing the resource allocation among processes and threads.

4.8 Region 5 - Hydrodynamical forces

On the escort of the SPH smoothing length h that has been determined in the density loop (see par. 4.7), the hydrodynamical forces and their impact on the physical properties of each active SPH particle are estimated. That is achieved through a second loop over the SPH particles in which for every particle relevant data on its neighbourhood (identified as the sphere of radius h_i centered on the particle i) are collected and processed. The same communication pattern as in the density loop applies.

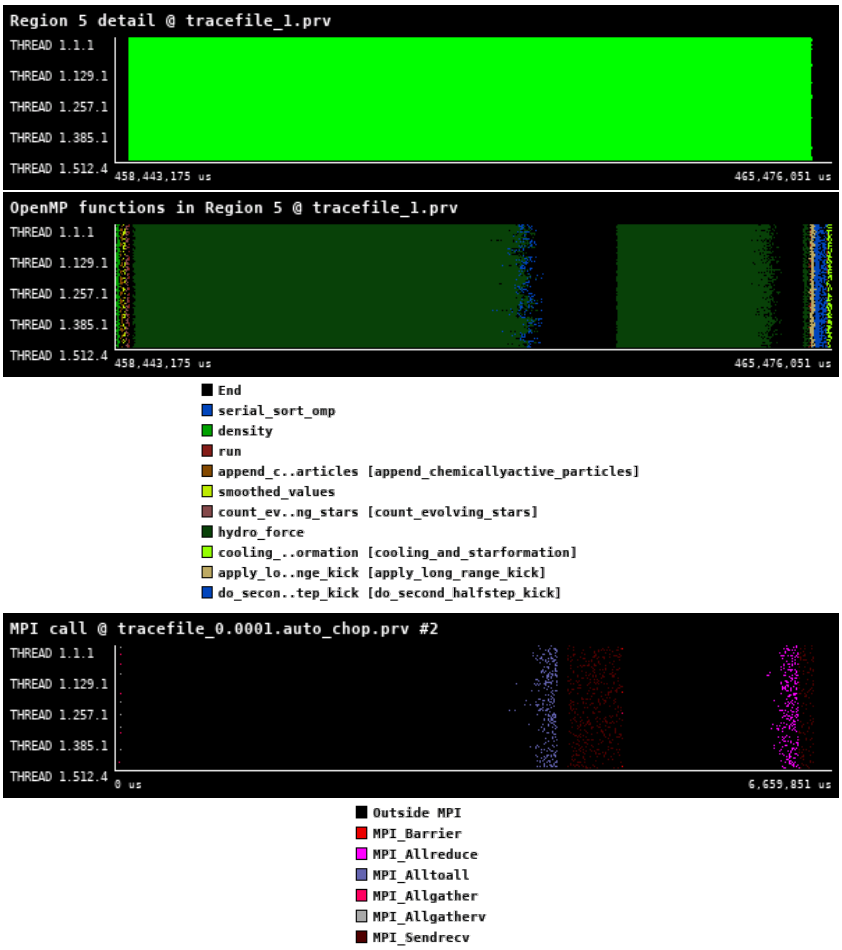


Figure 39: Zoomed traces for Region 5 of the Gadget from Paraver showing the structure of the region.

Number of processes	2048	4096	8192
Elapsed time (sec)	6.659527	3.835512	3.135103
Efficiency	1.0	0.86814	0.531045
Speedup	1.0	1.736281	2.124181
Average IPC	1.434770	1.424834	1.326382
Average frequency (GHz)	3.077587	3.091464	3.135689

Table 17: Overview of the key performance metrics of Region 5 of Gadget.

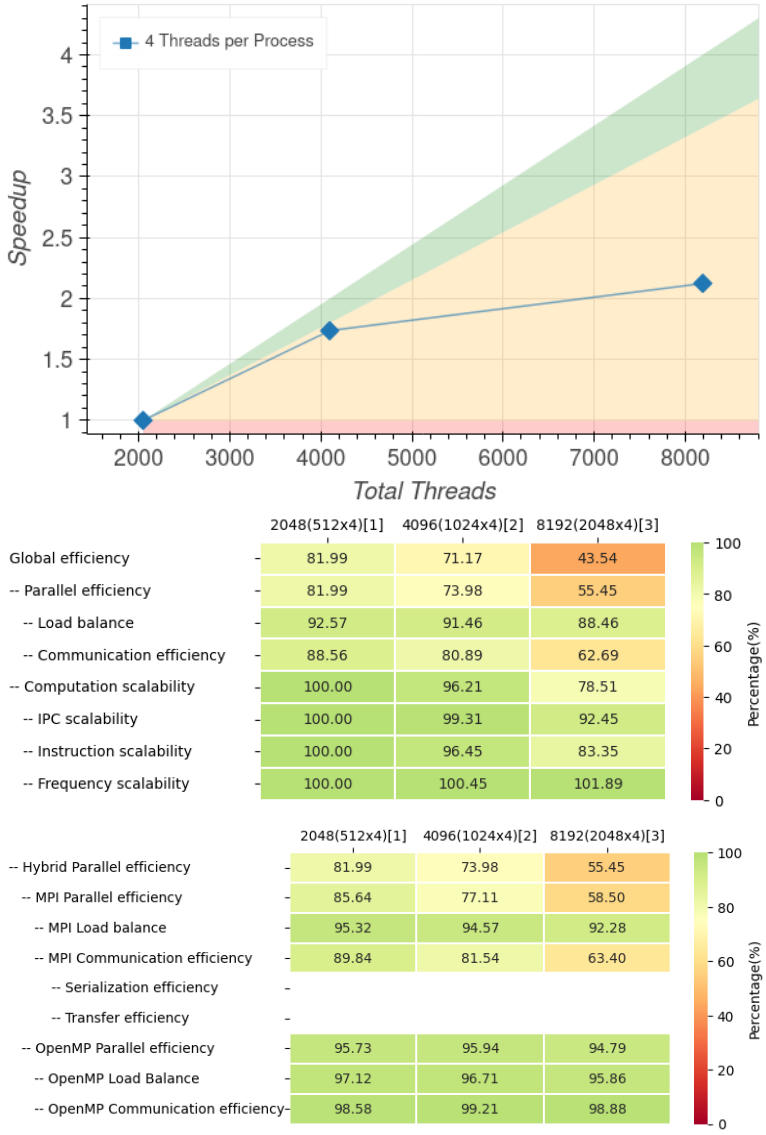


Figure 40: Strong scaling and POP efficiency metrics for Region 5 of Gadget.

This region has a clear expected pattern: *(i)* a first (dark green) region of local computations and collection of details about the pattern of point-to-point communications, *(ii)* a communication round, that consists mostly in `MPI_Sendrecv`, and *(iii)* finally a last computation on the communicated data, followed by a short final phase of collective reduction.

While the load balance and the instruction scalability are pretty good, although not optimal, as before the key problem observed is the low communication efficiency. Processes that spend the least amount of time in communication still spend about 40% (for 8192 cores) in communication routines. Although this problem is less communication-bound than the density itself, it is possible to recognize the same behaviour. Since reducing the communication volume is not possible, the potential optimization is to re-design the pattern in order to overlap computation and communication. In other words, instead of having big communications phases, a possibility is to specialize one OpenMP thread in performing communications while the other threads in the pool process the received data.

4.9 Region 6 - Extra-physics

A large variety of physical effects is included in physical modules that could be activated at compile-time. Among the most significant are heat conduction, radiative transfer, radiative cooling, star formation, stellar feedback, chemical enrichment, the network of non-equilibrium chemistry, cosmic rays and the black-holes accretion and feedback.

In the specific example that we are analyzing here, we have chosen to activate the most standard modules: heat conduction, radiative cooling, star formation, stellar feedback and chemical enrichment.

Some of these processes, namely the radiative cooling, the non-equilibrium chemistry and the star formation, are purely local processes, meaning that they depend only on the physical properties of the target particles. Other processes, like stellar feedback and enrichment or the black holes accretion and feedback, involve their neighbourhood. Hence, for non-local processes, loops and communication patterns are very similar to those in density and hydrodynamical phases.

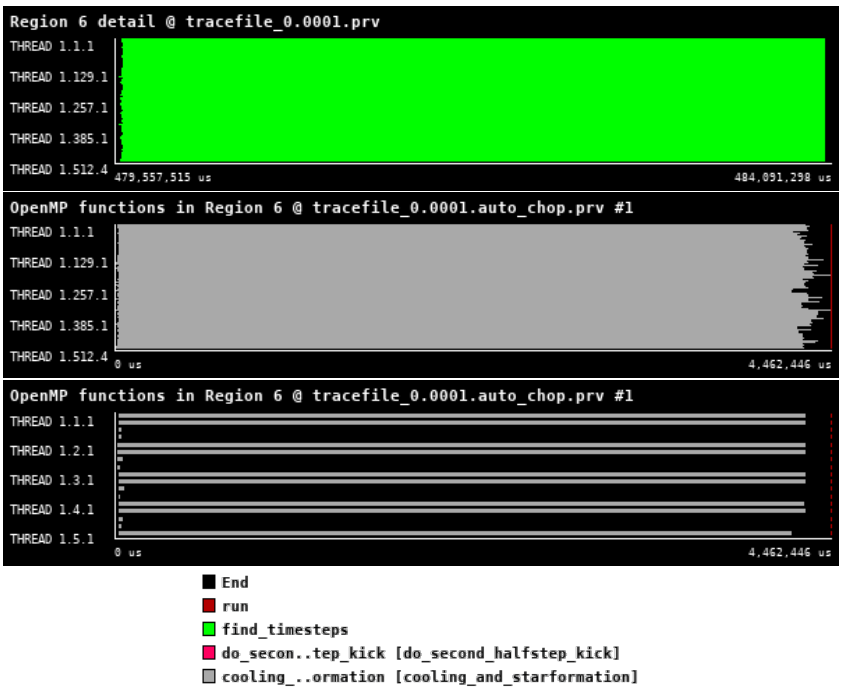


Figure 41: Zoomed traces for Region 6 of the Gadget from Paraver showing the structure of the region.

Number of processes	2048	4096	8192
Elapsed time (sec)	4.461576	2.237596	1.231189
Efficiency	1.0	0.996958	0.905949
Speedup	1.0	1.993915	3.623795
Average IPC	1.815168	1.815085	1.814466
Average frequency (GHz)	3.223408	3.219953	3.228040

Table 18: Overview of the key performance metrics of Region 6 of Gadget.

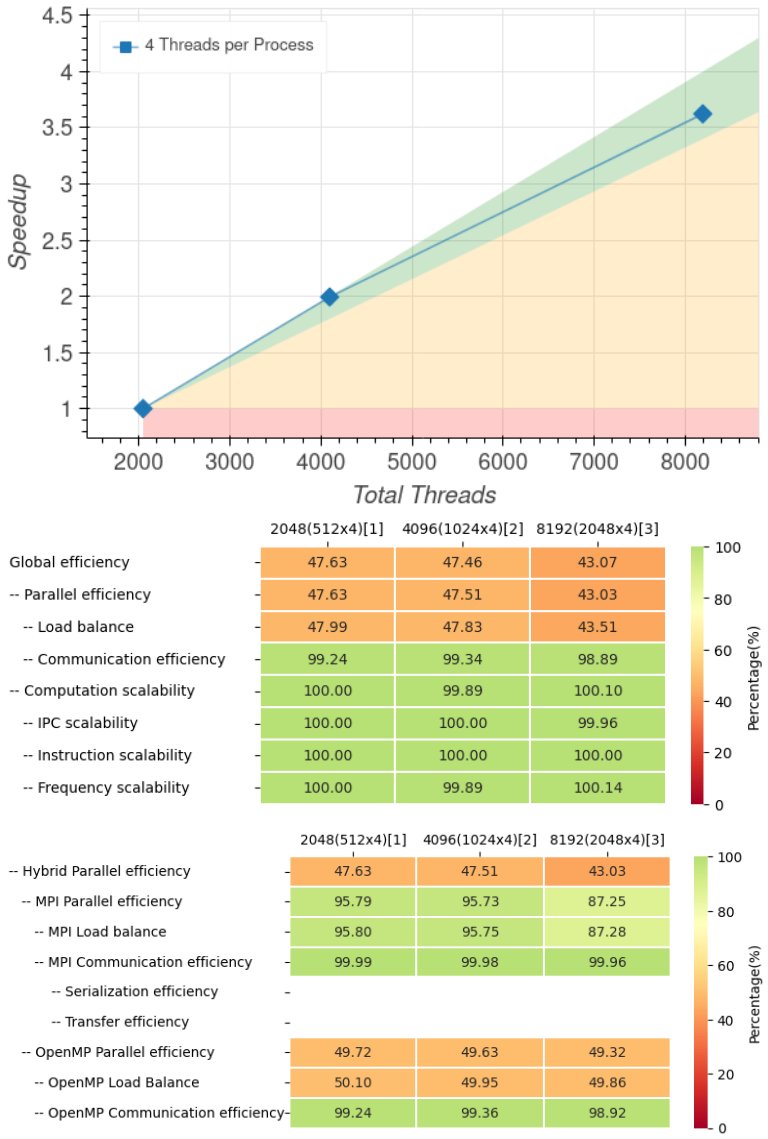


Figure 42: Strong scaling and POP efficiency metrics for Region 6 of Gadget.

This section of the program is primarily focused on performing computations on local particles. As a result, both the computational part and the MPI part scale quite well. However, the main source of inefficiency lies in the OpenMP-ization, which is currently not optimized well enough for this section. This issue is clearly highlighted by the analysis. Currently, there is a problem with load imbalance within the OpenMP region. The reason behind this issue is that only two out of the four threads are performing calculations, while the remaining two remain idle. We are currently unable to explain this phenomenon, but it is clear that the OpenMP implementation needs improvement, if not a complete redesign.

4.10 Conclusions

The performance analysis on this use case for OpenGADGET has shown a good scaling of the computational part and some current limitations in both some MPI communication schemes and the parallel load balance.

The domain decomposition section needs a re-writing of its architecture that includes an efficient thredization and a re-design of the calculation-communication patterns. The gravitational part has good behaviour in the tree-related sections but suffer from severe limitations in the Particle-Mesh, due to the limited domain decomposition capability of the `fftw` library; implementing a stencil decomposition would bring great benefit.

The density and hydrodynamical sections show similar behaviour, linked to the iteration of the same base scheme: local computation - communication - computation on received data. The analysis highlights that there is room for significant improvement here in the form of an overlapping of computation and communication. The last region, the "extra-physics" exhibits reasonable results in terms of parallel behaviour; however, we remind the reader that in the tested regime this was expected. We need to further investigate its behaviour at later cosmic simulation epochs.

In general, a much better vectorization and computation efficiency can be achieved, as shown by the average IPC. This calls for a profound re-design of the structures used to represent the data and is a long-term plan in this project for OpenGadget.

5 iPic3D

iPic3D [14] is a general-purpose computational code developed in the kinetic domain to study collisionless plasma dynamics using the 3D Implicit Particle-in-Cell (PIC) method [15, 16]. Here in the iPic3D, i stands for scheme that is used: implicit. iPic3D employs the implicit moment method to achieve an efficient implementation of the implicit formulation that allows about a multiscale discretization in time and space of the governing equations. The use can select what scales to resolve and the unresolved scales do not cause any numerical instability by virtue of the implicit nature of the temporal discretization. Hence, we can use time steps and grid spacing that are typically order 10-100 larger than time steps and grid spacing used in traditional PIC codes.

Fig 43 depicts the flow chart of the PIC method for the Vlasov-Maxwell systems [17].

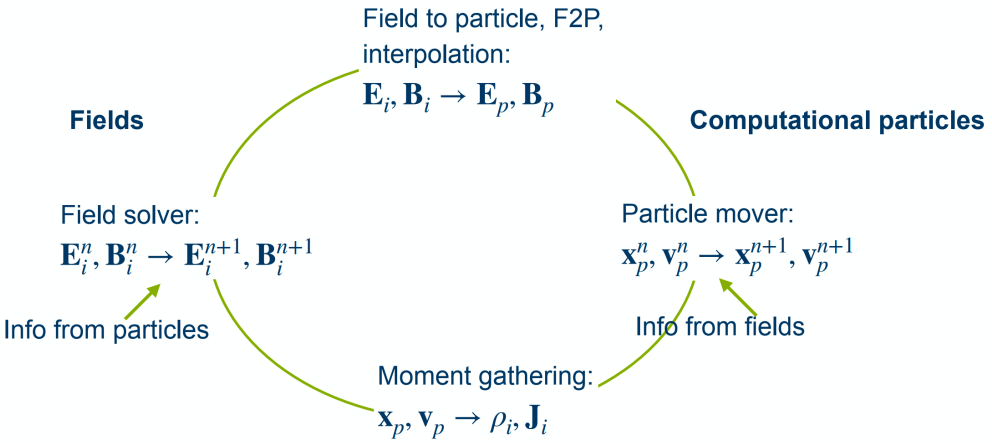


Figure 43: The flowchart of PIC method.

In the execution workflow of iPic3D, two pivotal steps are distinctly defined: initially, the electric **E**, magnetic **B**, velocity **v**, and position **r** fields are initialized on the computational grid, holding to a pre-defined setup. Subsequently, the simulation enters a cyclic phase wherein Maxwell's equations and the particle equations of motion are solved simultaneously across the grid, persisting through a specified number of iterations.[15].

The geometry of the iPic3D code as the name implies is 3D. The iPic3D works on a 3D grid, which help us to obtain a comprehensive understanding of various space phenomena like magnetic reconnection, study of instabilities like Kelvin-Helmholtz (KH) instability, lower hybrid instability, studies to understand the characteristics of turbulence etc. Reduced 2D and 1D runs can be done with the same code. In the uses case however we focused on 3D.

5.1 Use-case description

The scenario presented aligns with the Geospace Environmental Modeling Reconnection Challenge, a standardized simulation framework utilized by the scientific community to analyze magnetic reconnection events.

Figure 44 delineates the various phenomena and aspects of this simulation, highlighting the potential areas of study. Magnetic reconnection and associated processes are investigated within a Harris sheet configuration, subjected to a specific set of initial conditions. The primary reconnection occurs at the Harris current sheets, where magnetic field lines converge, interact, and subsequently diverge. Secondary reconnections, driven by the turbulence within the outflows, may also occur. These secondary reconnections contribute to further plasma acceleration and the transport of energy and material within the system.

From the simulation study, the results shown in Figure 35 depicts electron flow lines navigating complex magnetic reconnection sites. Color variations are reflecting local electric field intensities, highlighting the dynamics that lead to electron acceleration and turbulent plasma flows.

The specifics of the initialization are provided in the input file, shown in Figure 46. We need to increase the grid sizes and number particles for a better understanding of the phenomena. It is worth noting that we can change the grid size (nxc, nyc, nzc are the cells in each direction) and number of processors used (XLEN, YLEN, ZLEN form the processors topology) and evaluate its performance.

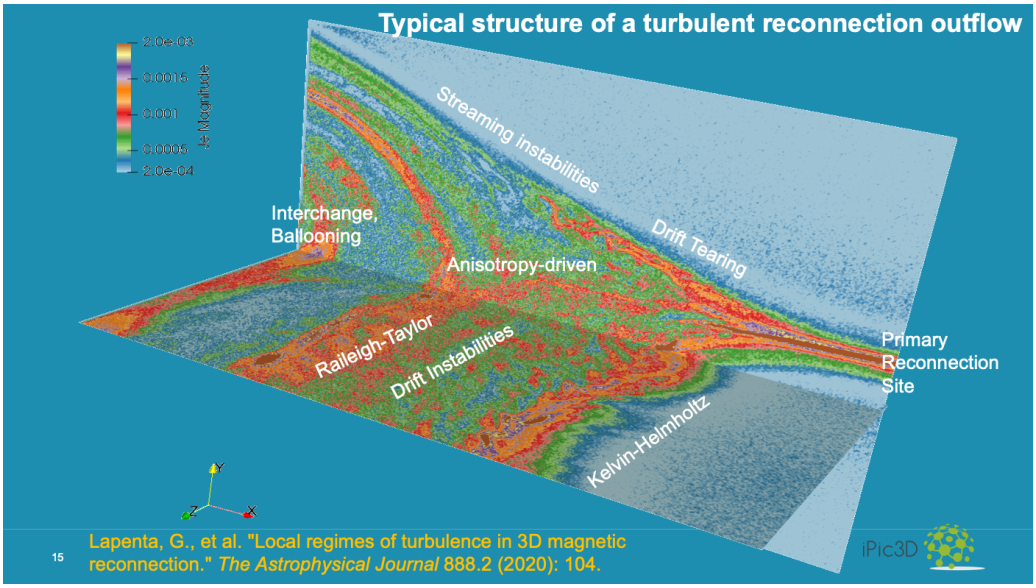


Figure 44: Various phenomena and aspects of this simulation, highlighting the potential areas of study.

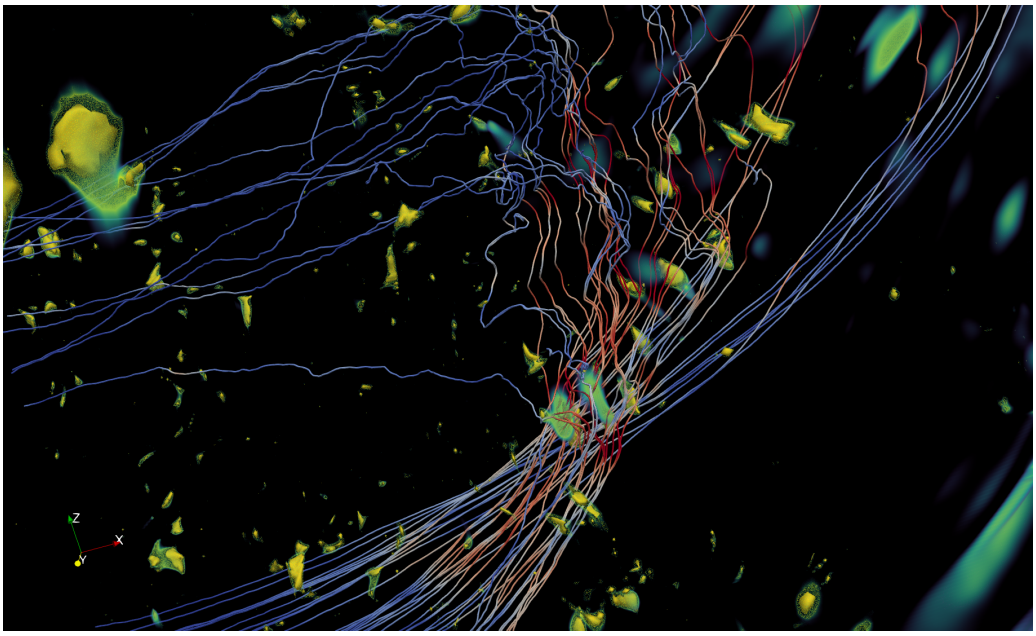


Figure 45: The visualization presents electron flow lines traversing through a primary magnetic reconnection site and interacting with adjacent reconnection regions.

```
SimulationName    = GEM                          # Simulation name for the output

#  %%%%%%%%%%%%%%%%%%%% Magnetic Reconnection %%%%%%%%%%%%%%%%%%%
B0x = 0.0097
B0y = 0.00
B0z = 0.00

# External magnetic field parameters:
B1x = 0.000
B1y = 0.000
B1z = 0.000

delta = 0.5

#  %%%%%%%%%%%%%%%%%%%% TIME %%%%%%%%%%%%%%%%%%%%
dt = 0.1                # dt = time step
ncycles = 500           # cycles
th = 0.5               # th =   decentering parameter
c = 1.0                 # c = light speed

#  %%%%%%%%%%%%%%%%%%%% SMOOTH %%%%%%%%%%%%%%%%%%%%
Smooth = 0.5            # Smoothing value (5-points stencil)
Nvolte = 6 ####Cyles of smoothing and this must be even

# %%%%%%%%%%%%%%%%%%%% BOX SIZE %%%%%%%%%%%%%%%%%%
Lx =   14.0              # Lx = simulation box length - x direction
Ly =   8.0             # Ly = simulation box length - y direction
Lz =   8.0              # Lz = simulation box length - z direction

x_center =   1.         # Lx = simulation box length - x direction in m
y_center =   1.         # Ly = simulation box length - y direction in m
z_center =   1.         # Lz = simulation box length - z direction in m
L_square =   .1

CoilD = 0.0###magnetic coil diameter -- Proposed WB8 simulation dimensions.
CoilSpacing= 0.0#               #spacing parameter for magnets

nxc =  512              # nxc = number of cells - x direction
nyc =  512             # nyc = number of cells - y direction
nzc =  512           # nzc = number of cells - z direction

# %%%%%%%%%%%%%%%%%%%% MPI TOPOLOGY %%%%%%%%%%%%%%%%%%
XLEN = 64               # Number of subdomains in the X direction
YLEN = 64             # Number of subdomains in the X direction
ZLEN = 64             # Number of subdomains in the X direction

# %%%%%%%%%%%%%%%% PARTICLES %%%%%%%%%%%%%%%%%%
#    0 = electrons
#    1 = protons
#    2,3,4,5,... = ions
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ns = 4                          # Number of particles
rhoINIT         = 1.0  1.0 0.01 0.01          # Initial density (make sure you are neutral)
rhoINJECT       = 0  0    0    0          # Injection density (make sure you are neutral)
TrackParticleID = 0    0    0    0      # TrackParticleID[species] = 1=true, 0=false

npcelx          = 10   10  10  10            # Particles per cell in X
npcely          = 10   10  10  10          # Particles per cell in Y
npcelz          = 10   10  10  10          # Particles per cell in Z
NpMaxNpRatio    = 12.0                # Maximum number of particles allocated

qom             = -256.0 1.0 -256.0   1.0        # Charge/mass ratio

uth             = 0.0045 0.0063 0.0045 0.0063     # Thermal velocity in X
vth             = 0.0045 0.0063 0.0045 0.0065    # Thermal velocity in Y
wth             = 0.0045 0.00631 0.0045 0.0065    # Thermal velocity in Z
u0              = 0.0     0.0    0.0 0.0    # Drift velocity in X
v0              = 0.0     0.0    0.0     0.0   # Drift velocity in X
w0              = 0.00325 -0.01624 -0.01624 -0.01624   # Drift velocity in X

# %%%%%%%%%%%%%% Periodicity in each direction %%%%%%%%%%%%%%%%%
PERIODICX       = 1                 # In direction X (1=true, 0=false)
PERIODICY       = 1             # In direction Y (1=true, 0=false)
PERIODICZ       = 1                 # In direction Z (1=true, 0=false)

cylindrical = 0

# %%%%%%%%%%%%%% boundary conditions %%%%%%%%%%%%%%%%%
# If the PERIODIC flag is active in the previous section
# periodic boundary conditions will be imposed
#
# PHI Electrostatic Potential
# 0,1 = Dirichilet boundary condition ;
# 2   = Neumann boundary condition
    bcPHIfaceXright = 1
    bcPHIfaceXleft  = 1
    bcPHIfaceYright = 1
    bcPHIfaceYleft  = 1
    bcPHIfaceZright = 1
    bcPHIfaceZleft  = 1

#    EM field boundary condition
#    0 = perfect conductor
#    1 = magnetic mirror
    bcEMfaceXright = 0
    bcEMfaceXleft =  0
    bcEMfaceYright = 0
    bcEMfaceYleft =  0
    bcEMfaceZright = 0
    bcEMfaceZleft =  0

#    Particles Boundary condition
#    0 = exit
#    1 = perfect mirror
#    2 = riemission
    bcPfaceXright = 1
    bcPfaceXleft =  1
    bcPfaceYright = 1
    bcPfaceYleft =  1
    bcPfaceZright = 1
    bcPfaceZleft =  1

# %%%%%%%%%%%%%% Numerics options %%%%%%%%%%%%%%%%%
verbose                 = 1       # Print to video results
Vinj                    = 0.0     # Velocity of the injection from the wall
CGtol                   = 1E-3    # CG solver stopping criterium tolerance
GMREStol                = 1E-3    # GMRES solver stopping criterium tolerance
NiterMover              = 3       # mover predictor_corrector iteration
FieldOutputCycle        = 100     # Output for field
ParticlesOutputCycle    = 100     # Output for particles if 1 it doesnt save particles data
RestartOutputCycle      = 4000    # restart cycle
DiagnosticsOutputCycle = 1        # Diagnostics cycle
```

Figure 46: The configuration of the simulation showing the setup of the use-case for iPic3D.

5.2 High-level code structure

The main body of the code is shown below. There are three important regions in this main body please see the code block 47. They are initialization, main loop and finalization. During the initialization phase, an instance of the `c_Solver` class from the `iPic3D` namespace is created, named `KCode`. Subsequently, the **Init**, `InjectBoundaryParticles`, and `GatherMoments` methods are invoked on this instance. These methods set up the simulation environment, incorporating boundary conditions, and collating initial data. In the Main Loop phase, a loop runs from the first cycle `(KCode.FirstCycle())` to the last cycle `(KCode.LastCycle())`, at a iteration-based simulation. Within each iteration or cycle, an event is initiated using the `Extrae_event` function [5], which profiles or tracks the code's performance. This code is parallelized using MPI. As the loop progresses, the `UpdateCycleInfo` method refreshes any information pertinent to the current cycle. The electric fields are then computed with the `CalculateField` function, and the particles are displaced using the `ParticlesMover` function. The **B** field is determined using the `CalculateBField` function. Moments are subsequently aggregated with the `GatherMoments` function. If an error is detected during particle movement, the loop will terminate prematurely. Finally, the output is written to files using the `WriteOutput`, `WriteConserved`, and `WriteRestart` methods, and the event that started with `Extrae_event` is concluded. Thus we will be able to get the performance of our code using the extrae function [5]. In the Finalization phase, the rank of the current process is obtained using the `getmyrank` function. Following this, the 'Finalize' method of the `KCode` is invoked to free up resources or finalize the data. The main function then returns a value of 0, indicating a successful execution.

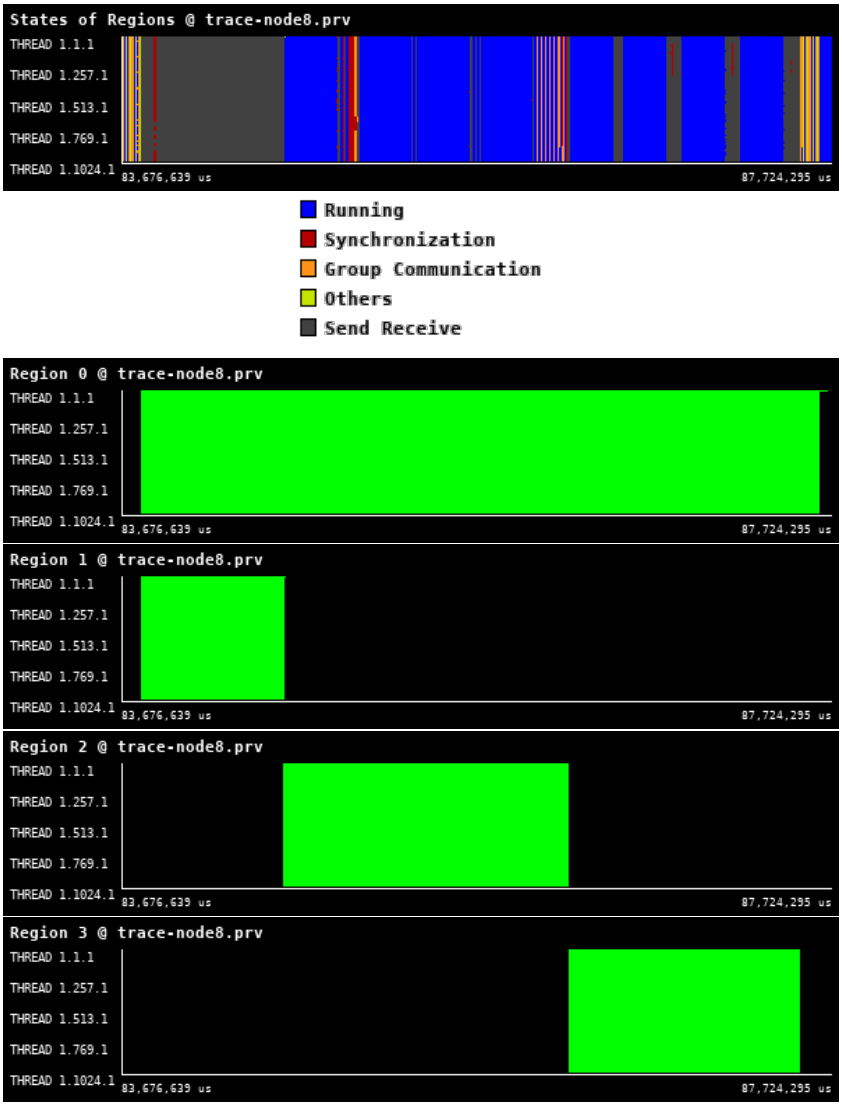


Figure 47: Traces for all regions in iPic3D from Paraver showing the high-level structure of the code.

The code structure is summarized in the main cycle of the code:

```
\label{code:ipic3d_codestructure}

int main(int argc, char **argv) {

  iPic3D::c_Solver KCode;
  bool b_err = false;

  /* ------------------------------ */
  /* 0- Initialize the solver class */
  /* ------------------------------ */

  KCode.Init(argc, argv);
  KCode.InjectBoundaryParticles();
  KCode.GatherMoments();

  /* ------------ */
  /* 1- Main loop */
  /* ------------ */


  for (int i = KCode.FirstCycle(); i <= KCode.LastCycle(); i++) {
Extrae_event(100001+i*100,1);
    if (KCode.get_myrank() == 0) cout << "␣=======␣Cycle␣" << i << "␣=======␣" << endl;

    /* ---------------------------------------------------- */
    /* 2- Calculate fields and move particles               */
```

```
    /*    Exit if there is a memory issue with the particles */
    /* ---------------------------------------------------- */

    KCode.UpdateCycleInfo(i);
Extrae_event(100002+i*100,1);
    clocks->start(2);
    KCode.CalculateField();
    clocks->stop(2);
Extrae_event(100002+i*100,0);

Extrae_event(100003+i*100,1);

    clocks->start(3);
    b_err = KCode.ParticlesMover();

Extrae_event(100003+i*100,0);

    if (!b_err) KCode.CalculateBField();
    clocks->stop(3);
Extrae_event(100004+i*100,1);
    clocks->start(1);
    if (!b_err) KCode.GatherMoments();
    clocks->stop(1);
Extrae_event(100004+i*100,0);
    if ( b_err) i = KCode.LastCycle() + 1;

    /* --------------- */
    /* 3- Output files */
    /* --------------- */

    clocks->start(4);
    KCode.WriteOutput(i);
    KCode.WriteConserved(i);
    KCode.WriteRestart(i);
    clocks->stop(4);

Extrae_event(100001+i*100, 0);
  }

int myrank;
myrank = KCode.get_myrank();

  KCode.Finalize();

  return 0;
}
```

5.3 Calculate the Fields - Region 1

This region of the calculation pertains the solution of the Maxwell equations closed by the plasma dielectric tensor produced by the implicit moment method. This is a linear but non-symmetric system that needs to be solved with GMRES or other methods not assuming a symmetric matrix. Experience tells us this is the region of the code that has the most trouble scaling efficiently on supercomputers due to the global communication needed in the reduction operations of the GMRES algorithm. Here we use in the scaling study our own implementation of the code. We have a PETSc version that will be investigated in future work and will be extended to GPUs. The field region is normally a relatively minority part of the cost compared with all particle operations, but once the particle operations are massively scaled up in processors and moved to GPUs this part becomes the bottleneck.

```
    void c_Solver::CalculateField() {

  // timeTasks.resetCycle();
  // interpolation
  // timeTasks.start(TimeTasks::MOMENTS);

  EMf->interpDensitiesN2C(vct, grid);        // calculate densities on centers from nodes
  EMf->calculateHatFunctions(grid, vct);     // calculate the hat quantities for the implicit method
  MPI_Barrier(MPI_COMM_WORLD);
  // timeTasks.end(TimeTasks::MOMENTS);

  // MAXWELL'S SOLVER
  // timeTasks.start(TimeTasks::FIELDS);
  EMf->calculateE(grid, vct, col);               // calculate the E field
  // timeTasks.end(TimeTasks::FIELDS);

}
```

This part of the iPic3D calculates the electric field. Here the electric field is calculated by using the current and density interpolated in region 4 and solves Maxwell's equations. Before we proceed with the main calculations, we need to perform an initial step that takes into account the results from the previous moment in time. This is a special characteristic of the method we are using, called the "implicit method."

The function `EMf-calculateHatFunctions(grid, vct)` is called to do this initial step. What this function does is calculate certain values that are necessary for the next part of our process, which is known as the "predictor-corrector method." This method is particularly used when we are moving particles in our simulation. Region 1 focuses on using the interpolated densities to calculate the electric field on the grid by solving Maxwell's equation formulated in the second order (i.e. eliminating B using Faraday's law) using the charge density computed earlier. The numerical solver used is GMRES to address the algebraic systems emerging from the spatial discretization of Maxwell's equations. The GMRES method, thus, acts as a numerical engine, enabling the precise calculation of electromagnetic fields, which in turn influence the particle dynamics in the simulation, establishing a cyclical interplay between fields and particles.

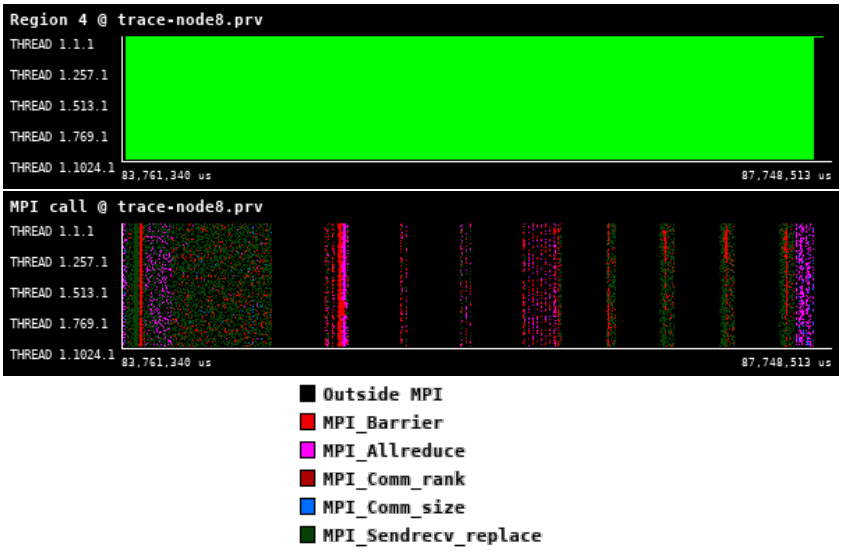


Figure 48: Zoomed traces for Region 1 of the iPic3D from Paraver showing the structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	3.910475	7.265658	2.35257	4.075049
Efficiency	1.0	0.269107	0.415554	0.119952
Speedup	1.0	0.538213	1.662214	0.959614
Average IPC	1.910302	1.901241	1.870737	1.896299
Average frequency (GHz)	2.718350	2.703843	2.674885	2.640611

Table 19: Overview of the key performance metrics of Region 1 of iPic3D.

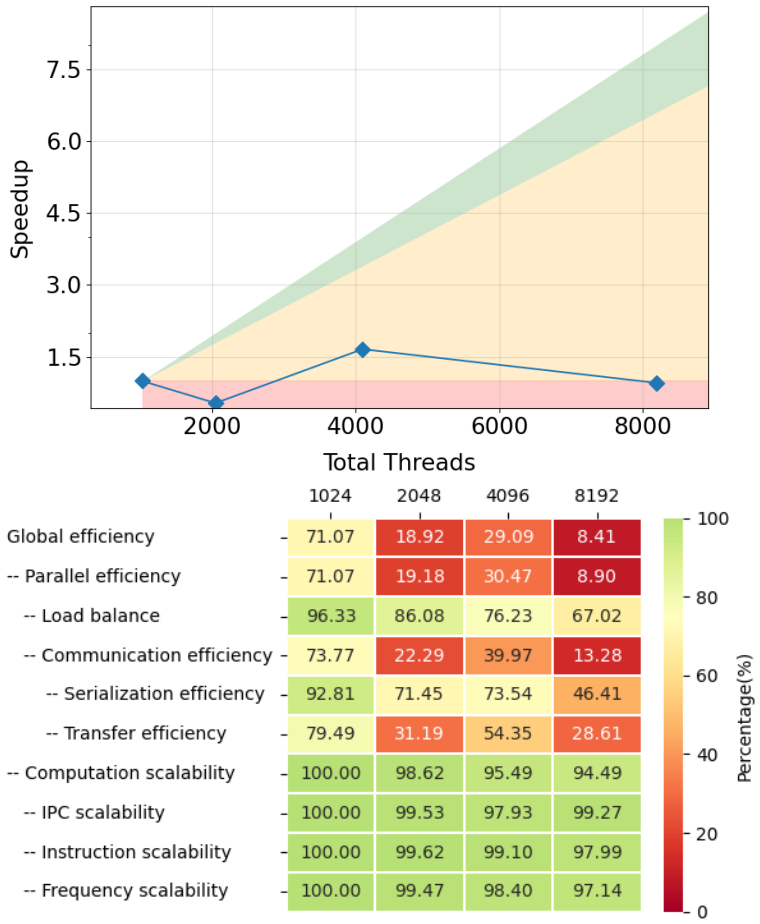


Figure 49: Strong scaling and POP efficiency metrics for Region 1 of iPic3D.

Based on the analysis (see Figure 49), the scalability of the code is limited due to communication efficiency. This is mainly due to the large amount of collective operations (All_reduce) and also synchronizations calls (Barrier) as shown in Figure 48. The scalability of the entire code is not well represented (in reality it is better) because due to an extremely large amount of small-size communications, the tracing is causing a non-negligible overhead. However, the problem remains and for the following optimization work, we recommend focusing on the analysis of the communication pattern and seeing whether communications can be grouped and hidden behind the calculations. On the algorithmic level one can explore the pipelined version of the GMRES solver, which provides even better communication hiding.

5.4 Particle Mover - Region 2

The *ParticlesMover* function in iPic3D manages the advancement of particle properties across discrete time steps utilizing a Predictor-Corrector (PC) scheme. The function iterates over each particle species, invoking mover-PC-sub(grid, vct, EMf) to compute an intermediate (predicted) particle state, subsequently utilizing updated electromagnetic fields (derived from the intermediate state) to correct the particle positions and velocities. Memory availability is verified post-particle movement; if insufficient, the simulation outputs an error message and signals termination. The PC method ensures numerically stable and accurate particle propagation by first predicting states using current fields, then correcting with newly computed fields, effectively mitigating numerical discrepancies introduced during particle pushing. The mover includes an operation of interpolation of the electric and magnetic fields to the particles.

```
  bool c_Solver::ParticlesMover() {

  /*  -------------- */
  /*  Particle mover */
  /*  -------------- */

  // timeTasks.start(TimeTasks::PARTICLES);
  for (int i = 0; i < ns; i++)  // move each species
  {
    // #pragma omp task inout(part[i]) in(grid) target_device(booster)
    mem_avail = part[i].mover_PC_sub(grid, vct, EMf); // use the Predictor Corrector scheme
  }
  // timeTasks.end(TimeTasks::PARTICLES);

  if (mem_avail < 0) {          // not enough memory space allocated for particles: stop the simulation
    if (myrank == 0) {
      cout << "************************************************************" << endl;
      cout << "Simulation stopped. Not enough memory allocated for particles" << endl;
      cout << "************************************************************" << endl;
    }
    return (true);              // exit from the time loop
  }
% \end{verbatim}
```

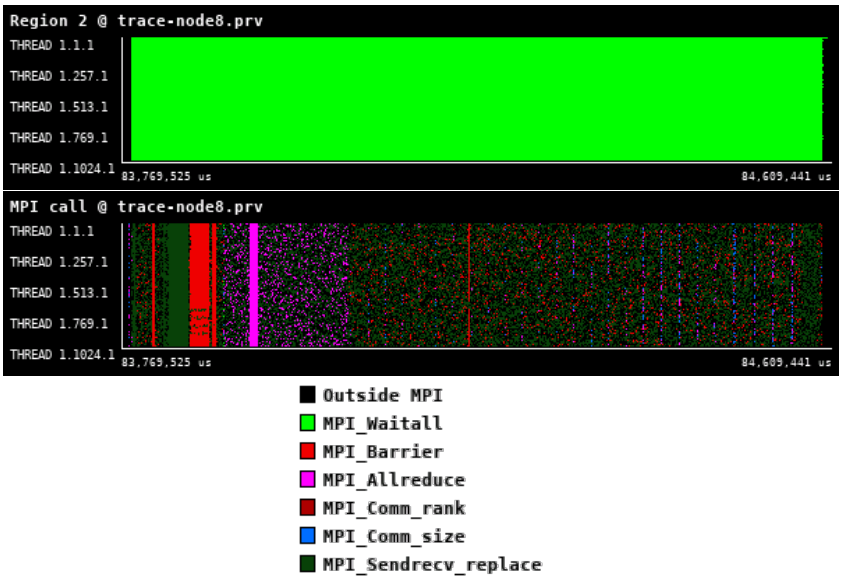


Figure 50: Zoomed traces for Region 2 of the iPic3D from Paraver showing the structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	0.818794	3.9449	0.90646	1.837977
Efficiency	1.0	0.103779	0.225822	0.055686
Speedup	1.0	0.207558	0.903288	0.445487
Average IPC	1.534331	1.640066	1.479532	1.292432
Average frequency (GHz)	1.814285	1.642642	1.504361	1.411053

Table 20: Overview of the key performance metrics of Region 2 of iPic3D.

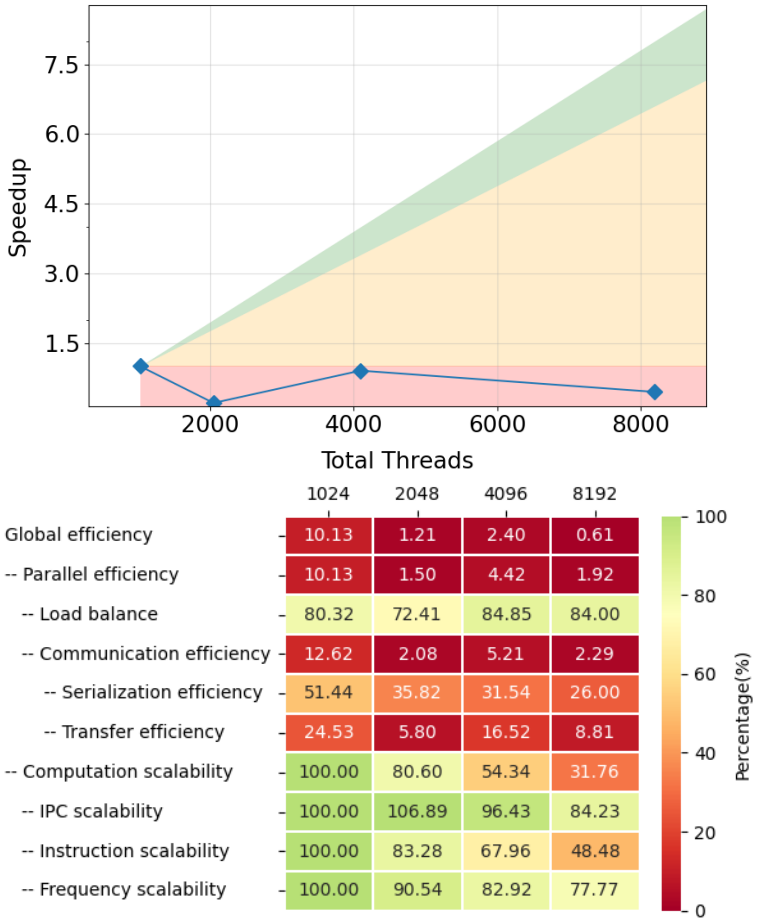


Figure 51: Strong scaling and POP efficiency metrics for Region 2 of iPic3D.

The basic analysis of POP metrics (see Figure 51) shows that, in this region, the problem that limits scalability also comes from the transfer efficiency [1, 2]. Figure 51 depicts significant amount of collective and synchronization MPI calls. We also see an increasing load imbalance which, when combined with collectives and barriers, causes the main reason for very low parallel efficiency. For this region, we also recommend a detailed analysis of the communication pattern and its implementation. Possible optimizations will involve study of the optimal algorithm for collectives, grouping of communications into larger messages, and reducing the frequency of communication.

Also, we see a decrease in instruction scaling, most probably caused by the problem size selected for initial performance analysis. In the near future, we will investigate how this is affected by the size of the problem. As in the case of the previous region, the limited scalability of the code as shown in this report is caused by the tracing overhead.

5.5 Calculate B field - Region 3

The `CalculateBField` function is tasked with computing the magnetic field, and uses Faraday's law. This is an explicit operation using the electric field computed in region 1 and the values of the previous time steps.

```
    void c_Solver::CalculateBField() {
  /* --------------------- */
  /* Calculate the B field */
  /* --------------------- */

  // timeTasks.start(TimeTasks::BFIELD);
  EMf->calculateB(grid, vct, col);   // calculate the B field
  // timeTasks.end(TimeTasks::BFIELD);

  // print out total time for all tasks
  // timeTasks.print_cycle_times();
}
```

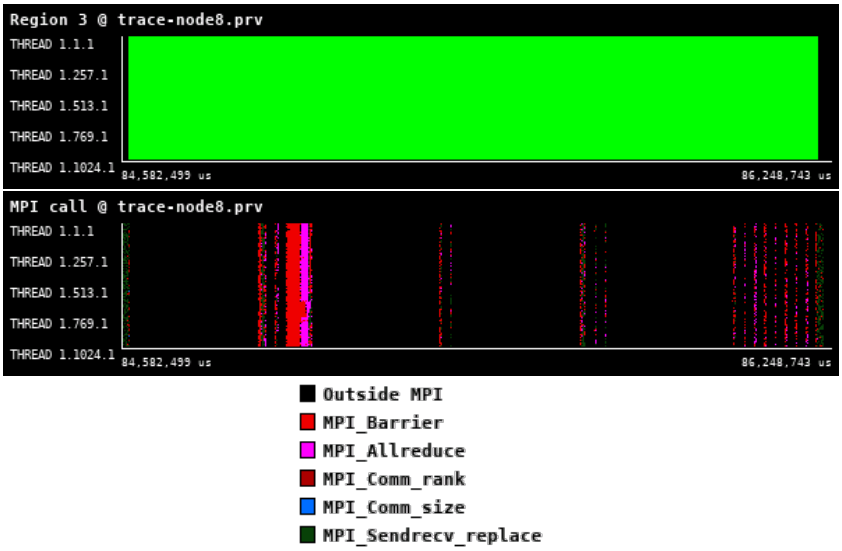


Figure 52: Zoomed traces for Region 3 of the iPic3D from Paraver showing the structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	1.315812	1.451175	0.833654	1.103825
Efficiency	1.0	0.453361	0.394592	0.149006
Speedup	1.0	0.906722	1.578367	1.192048
Average IPC	1.530675	1.503970	1.467013	1.563908
Average frequency (GHz)	2.068616	2.068002	2.066256	2.069223

Table 21: Overview of the key performance metrics of Region 3 of iPic3D.

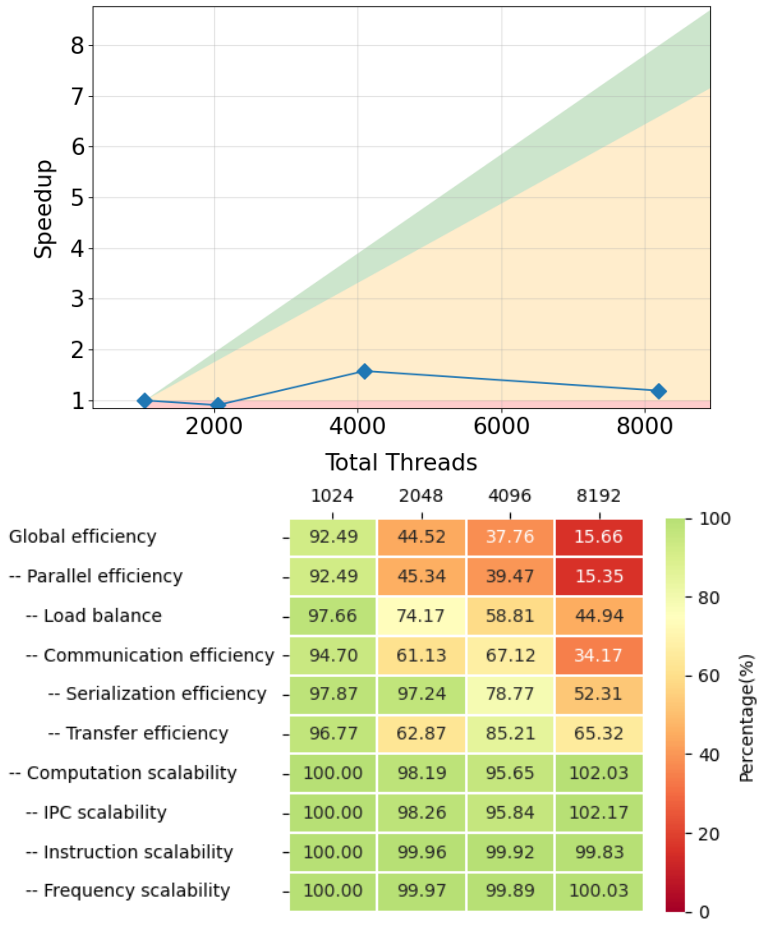


Figure 53: Strong scaling and POP efficiency metrics for Region 3 of iPic3D.

The analysis of this region, see Table 21 and Figure 53, again shows that there are limited transfer and serialization efficiencies. Compared to previous regions, Figure 52 shows that the frequency of collective operations is lower, but there are still sub-regions that exhibit a significant amount of these calls. Another key problem is the load imbalance. We recommend a detailed evaluation of the communication patterns. We also see that there is no issue with computation scalability, which means that the parallel region efficiently utilizes the resources when code is not in the communication calls. This part of the code has potential to be ported to GPU accelerator and target for single-core optimizations, like vectorization.

5.6 Gather Moments - Region 4

The `GatherMoments` function orchestrates the transition from discrete particle information to continuum grid. Initially, grid densities are reset, ensuring no residual data interferes with new calculations. Then, an interpolation of particle attributes, such as charge and velocity, to grid nodes (P2G) is performed. This step that can be computationally demanding due to the involvement of all particles and grid points. Subsequently, contributions from all particle species are accumulated to compute total charge and current densities on the grid, a vital step for electromagnetic field calculations.

```
  void c_Solver::GatherMoments(){
  // timeTasks.resetCycle();
  // interpolation
  // timeTasks.start(TimeTasks::MOMENTS);

  EMf->updateInfoFields(grid,vct,col);
  EMf->setZeroDensities();                    // set to zero the densities

  for (int i = 0; i < ns; i++)
    part[i].interpP2G(EMf, grid, vct);        // interpolate Particles to Grid(Nodes)

  EMf->sumOverSpecies(vct);                   // sum all over the species
  //
  // Fill with constant charge the planet
  if (col->getCase()=="Dipole") {
    EMf->ConstantChargePlanet(grid, vct, col->getL_square(),col->getx_center(),col->gety_center(),col->getz_center());
  }

  // EMf->ConstantChargeOpenBC(grid, vct);      // Set a constant charge in the OpenBC boundaries

}
```

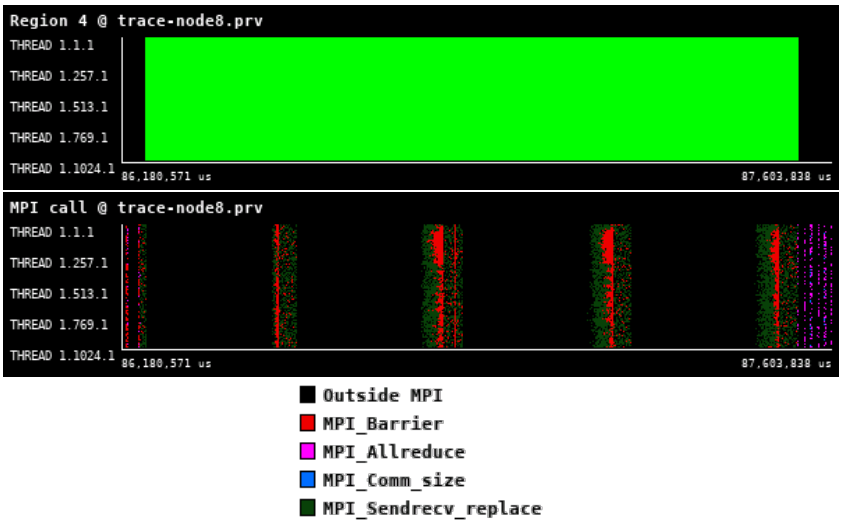


Figure 54: Zoomed traces for Region 4 of the iPic3D from Paraver showing the structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	1.054096	1.456185	0.425057	0.915114
Efficiency	1.0	0.361938	0.619973	0.143984
Speedup	1.0	0.723875	2.479893	1.151874
Average IPC	2.526970	2.593448	2.579069	2.571237
Average frequency (GHz)	2.068498	2.060184	2.048735	2.024946

Table 22: Overview of the key performance metrics of Region 4 of iPic3D.

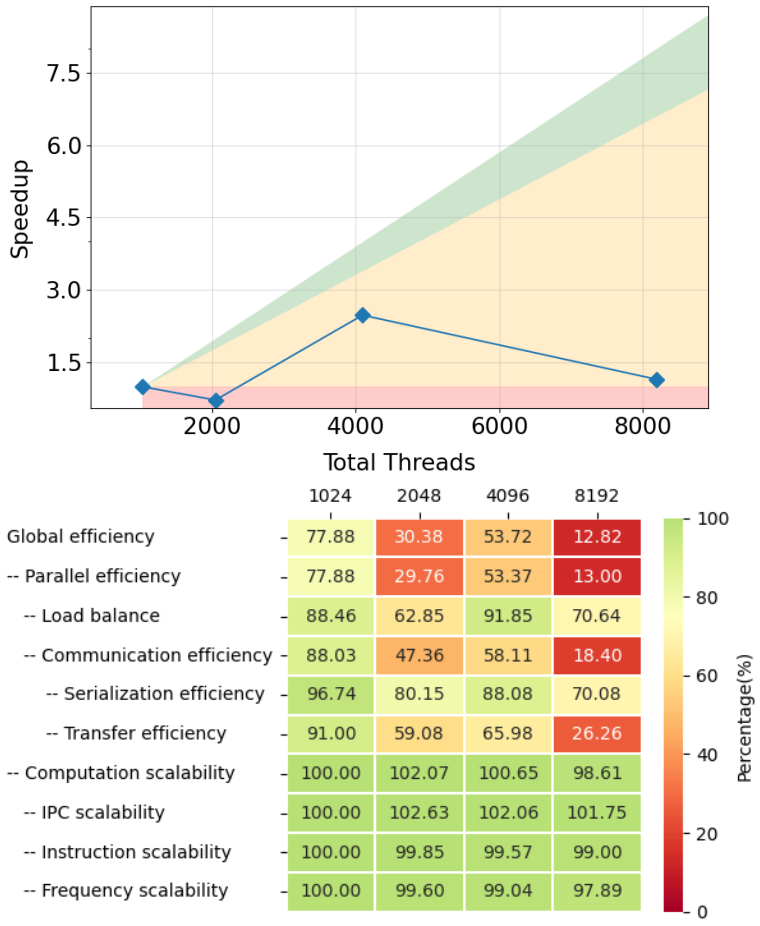


Figure 55: Strong scaling and POP efficiency metrics for Region 4 of iPic3D.

The analysis of this region, see Table 22 and Figure 55, again shows that there is limited parallel efficiency resulting from the transfer efficiency. Compared to previous regions, Figure 54, shows regions with Send/Receive pattern followed by barriers. In this case, the further evaluation should focus on the reduction of synchronization. The computation scalability is very good showing that this region is a good candidate for GPU acceleration.

5.7 Conclusions

The comprehensive execution pattern showcased in Figure 48 offers profound insights into the performance dynamics of the entire code. The trace predominantly displays a uniform task distribution across threads or processes, exemplifying optimal parallel efficiency. However, discernible disparities or irregularities in this pattern suggest potential load imbalances, hinting at areas where addressing these imbalances could elevate parallel performance.

Synchronized pauses or gaps, especially when observed across multiple threads or processes, serve as markers for synchronization points. These points often act as performance bottlenecks in parallel applications. Minimizing or efficiently handling these synchronization events could substantially boost execution efficiency. Concurrent task execution, illustrated by overlapping colors across different threads or processes, emphasizes efficient resource utilization. However, instances where a single task predominates for extended periods spotlight potential bottlenecks or synchronization challenges. Furthermore, significant gaps in the trace, particularly those synchronized across threads, highlight idle times that may stem from waiting on data, synchronization barriers, or other dependencies.

Diving into the "Calculate E Field" phase, as depicted in Figure 48, the trace reveals a strong influence of communication, especially MPI calls. This dominance suggests a shift towards being more communication-bound than computation-bound, indicating that performance may be more constrained by communication overhead than by the primary computational tasks.

Focusing on the "Particle Mover" region, the execution trace delineates periods of concentrated task execution through its green patches. Their pronounced presence underscores their computationally intensive nature. The accompanying MPI trace deciphers the inter-process communications intricacies, with synchronized patterns across processes pointing to global communication events. In particular, patterns corresponding to MPI-Barrier emphasize moments of synchronization among all processes, while patterns related to collective operations like MPI-Allreduce indicate global data distribution events. Prolonged instances of specific colors, especially those associated with communication calls, suggest potential communication bottlenecks.

In conclusion, the traces provide a detailed view of the code's performance characteristics, extending from overarching execution patterns to nuanced regional intricacies. While there are evident indicators of efficient parallelism and resource utilization, areas like communication overhead in certain phases and potential synchronization bottlenecks warrant closer examination. In particular, similar observations and patterns are also reflected in the other two regions, underscoring the consistency in performance dynamics across different segments of the code.

The observed performances confirm the previous experience and reaffirm our plan for code algorithmic improvements with focus to reduce communication and for compute bound regions their offloading to GPU.

It is important to note that the limited scalability of the code as shown in this section is also caused by the tracing overhead. We will investigate the options to reduce the tracing overhead for selected regions and optimize it for further analysis of regions selected for kernel extraction and further optimization.

6 RAMSES

RAMSES [18, 19] is an Adaptive-Mesh-Refinement (AMR) code which is used to study astrophysical fluid dynamics and the formation of structures in the Universe. It is based on an oct-tree structure, where parent cells are refined into children cells on a cell-by-cell basis following some user-defined criteria. RAMSES can deal with 1D, 2D and 3D Cartesian grids.

RAMSES integrates the equations of fluid dynamics in their conservative form

$$\left\{\begin{array}{lcccl} \partial_t \rho & + & \nabla \cdot [\rho \mathbf{u}] & = & 0, \\ \partial_t \rho \mathbf{u} & + & \nabla \cdot [\rho \mathbf{u} \otimes \mathbf{u} + P \mathbb{I}] & = & 0, \\ \partial_t E & + & \nabla \cdot [\mathbf{u}\,(E+P)] & = & 0, \end{array}\right. \tag{15}$$

where ρ is the density, $\mathbf{u}$ the velocity, E the total energy, i.e. $E = e + 1/2\rho u^2$, with $e = P/(\gamma - 1)$ the internal energy. This system can be written in the canonical form

$$\frac{\partial \mathbb{U}}{\partial t} + \nabla . \mathbb{F}(\mathbb{U}) = 0, \tag{16}$$

where the vector $\mathbb{U} = (\rho, \rho\mathbf{u}, E)$ contains the conservative variables. The flux vector $\mathbb{F}(\mathbb{U}) = (\rho\mathbf{u}, \rho\mathbf{u} \otimes \mathbf{u} + P\mathbb{I}, \mathbf{u}\,(\mathrm{E} + P))$ is a linear function of $\mathbb{U}$, and uses the primitive variables ρ, $\mathbf{u}$ and P. This system is closed using a perfect gas equation of state.

RAMSES uses an explicit second-order predictor-corrector finite-volume Godunov scheme to integrate the conservative system of equations. The hydrodynamic solver consists in computing flux at cell's interfaces, dealing with coarse-to-fine and fine-to-coarse interfaces. The discretized scheme writes (here in 1D for simplicity)

$$\frac{\mathbb{U}_i^{n+1} - \mathbb{U}_i^n}{\Delta t} \times V_i = \mathbb{F}_{i+1/2}^{n+1/2} S_{i+1/2} - \mathbb{F}_{i-1/2}^{n+1/2} \times S_{i-1/2}, \tag{17}$$

where $\mathbb{U}_i^n$ is the state variable $\mathbb{U}$ at time n and averaged in the cell i of volume V_i. $\mathbb{F}_{i-1/2}^{n+1/2}$ is the flux at the interface between cells i and $i-1$, computed using a linear Riemann solver. The initial values of the Riemann problems are obtained from the primitive variables of cells $i-1$ and i, extrapolated in time and space at the interface in the predictive step. Altogether, the scheme follows the same steps as illustrated in figure 12 of the PLUTO code.

The time integration can be also accelerated using the adaptive-time-step implementation, in which each AMR level evolve with its own time-step which satisfies a global synchronisation point at the end of the coarser time-step.

For this deliverable D2.1, we have chosen to isolate the hydrodynamic solver from the other modules of the code. We have selected the regions of the the predictor-corrector Godunov solver (equation 17), as well as two regions corresponding to the communications between MPI domains (update of one process from the others and global communication of the updated value from one process to all others).

6.1 Use-case description

The classic Sedov-blast test is the first use-case. This test is standard and used in most of the regular tests and benchmarks of astrophysics gas dynamics codes. The use-case is run in 3D with a uniform grid in order to focus on the three selected regions, without any overhead due to adaptive mesh refinement. The physical setup consists in a uniform density and pressure medium at rest (zero velocity) in which an internal energy pulse is put in the corners of the computational domain. The internal energy is then progressively converted into kinetic energy via total energy conservation. A time evolution sequence of the density field is shown in figure 56.

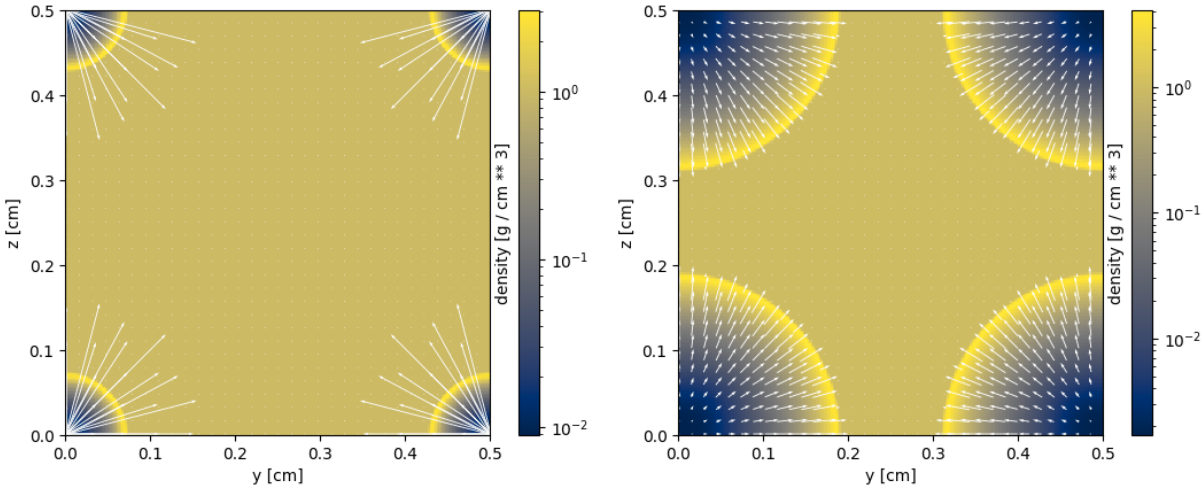
We use a 2048^3 uniform grid and the Lax-Friedrich Riemann solver. There is no load balancing done after the initialization of the simulation run. In this use-case all time steps (or iterations) are equivalent in terms of computational cost.

The code is compiled using the following options:

```
make NDIM=3 SOLVER=hydro MPI=1
```

The input file is:

```
&RUN_PARAMS
hydro=.true.
ncontrol=1
nrestart=0
nremap=0
nsubcycle=10*1
verbose=.false.
nstepmax=10
/

&AMR_PARAMS
levelmin=11
levelmax=11
ngridtot=2000000000
nexpand=1
boxlen=0.5
/

&INIT_PARAMS
nregion=2
region_type(1)='square'
region_type(2)='point'
x_center=0.5,0.0
y_center=0.5,0.0
z_center=0.5,0.0
length_x=10.0,1.0
length_y=10.0,1.0
length_z=10.0,1.0
exp_region=10.0,10.0
d_region=1.0,0.0
u_region=0.0,0.0
v_region=0.0,0.0
w_region=0.0,0.0
p_region=1e-5,0.4
/

&OUTPUT_PARAMS
noutput=3
foutput=0
/

&HYDRO_PARAMS
gamma=1.4
courant_factor=0.8
scheme='muscl'
slope_type=1
riemann='llf'
/

&REFINE_PARAMS
interpol_var=0
interpol_type=0
err_grad_p=0.1
/
```



(a) Time $t = 3.3 \times 10^{-3}$.

(b) Time $t = 3.9 \times 10^{-2}$.

Figure 56: 2D cut (plane $x = 0$) of the gas density at two different times for the Sedov blast test. The blast wave expands from the computational box corners with time. Note that for this visualisation, a resolution of only 256^3 has been used. Units are arbitrary.

6.2 High-level code structure

RAMSES first performs the initialisation of the grid, the load balancing into the MPI domains, and the initialisation of the hydrodynamics quantities. Each MPI process handles it own region but needs to know the hydrodynamical values of its neighboring MPI processes. This is done using ghost regions. Then it calls the main routine to perform the time integration.

The heart of the RAMSES code is the recursive routine `amr_step`, which handles the AMR hierarchy from the finer to the coarser levels. In this use-case, it is called only for a single level since we have chosen an uniform grid. Inside `amr_step`, the state of the vector $\mathbb{U}$ at time n is copied from the global vector `uold` into the temporary global vector `unew`. Then, the values of the vector $\mathbb{U}$ are updated to time $n+1$ by the second-order Godunov solver in the subroutine `godunov_fine`. After the hydrodynamic update, MPI domains need to share their updated values to their neighboring MPI domains. It is done in the `make_virtual_reverse` and `make_virtual_fine` routines, which employ asynchronous point to point MPI communications. After this, all MPI processes are synchronized and the new iteration can start with the updated `uold` vector.

```
program ramses
  istep=0
  call init_amr()
  call init_hydro()
  call load_balancing()
  do
    call amr_step(istep,levelmin,0)
    istep=istep+1
  end do
end ramses

------------------------------------------------

recursive subroutine amr_step(istep,ilevel,icount)

  EXTRAE_START(istep,ilevel,icount,0)

  ! some processing ...

  if ...
    call amr_step(istep,ilevel+1,0) ! 1 child amr_step
  else if ...
    call amr_step(istep,ilevel+1,0) ! 2 child amr_step
    call amr_step(istep,ilevel+1,1)
  else
    ! nothing, continue processing
  end if

  EXTRAE_START(istep,ilevel,icount,GODUNOV)
  call compute_godunov
  EXTRAE_STOP(istep,ilevel,icount,GODUNOV)

  EXTRAE_START(istep,ilevel,icount,REVERSE)
  call reverse_boundary
  EXTRAE_STOP(istep,ilevel,icount,REVERSE)

  EXTRAE_START(istep,ilevel,icount,UPDATE)
  call update_boundary
  EXTRAE_STOP(istep,ilevel,icount,UPDATE)

  ! some processing ...

  EXTRAE_STOP(istep,ilevel,icount,0)

  if ...
    stop ! exit application
  end if

end subroutine amr_step
```

The timeline of a single time step and the three internal regions is shown in Figure 57. There is a single region in which all computations are happening (Region 1) and two communication-heavy regions that follow. We can see a pattern similar to that in Pluto, where we have a long enough region where all the processing inside the domains takes place, followed by data exchanges between domains at the end of the time-step calculation.

This document is Public (PU)
and was produced under SPACE project EU GA 101093441

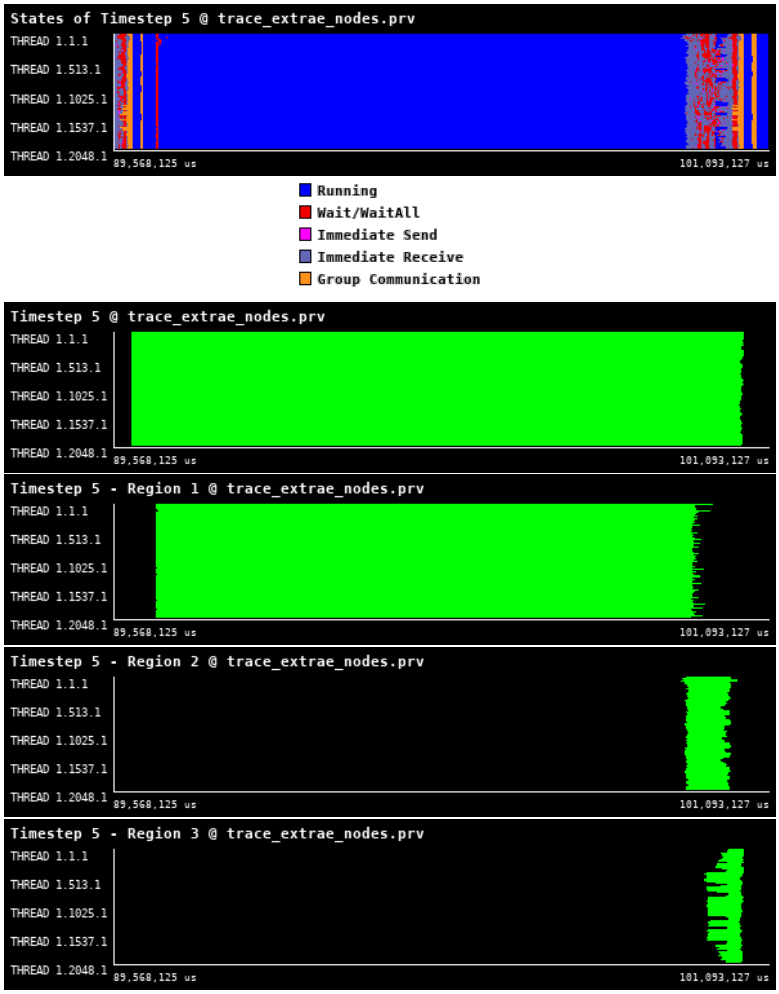


Figure 57: Traces for all regions in RAMSES from Paraver showing the high-level structure of the code.

6.3 Godunov solver - Region 1

Region 1 corresponds to the predictor-corrector Godunov solver (equation 17). This is a computational step, without I/O and MPI communication. It corresponds to the core of the hydrodynamical kernel, which can be extended to the MHD case. RAMSES uses an unsplit scheme, so that all directions are treated in a single call to the Godunov solver. Each cell of the computational domain is scanned, using loops with a low level of internal vectorizarion. Octs, i.e. groups of 2^{Ndim} cells, with $Ndim$ the number of dimensions, are first gathered by group of size `nvector` and then sent to the main routine of the Godunov kernel. The size `nvector` depends on each architectures and is the first basic optimisation we can perform.

```
! Hyperbolic predictor-corrector solver
call godunov_fine(ilevel)
```

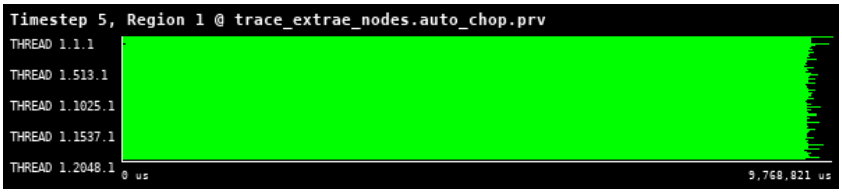


Figure 58: Zoomed traces for Region 1 of the RAMSES from Paraver showing the structure of the region.

Number of processes	2048	4096	16384
Elapsed time (sec)	9.760399	3.584701	1.037373
Efficiency	1.0	1.361396	1.176096
Speedup	1.0	2.722793	9.408765
Average IPC	1.623575	1.555882	1.535500
Average frequency (GHz)	2.094286	2.987629	2.997745

Table 23: Overview of the key performance metrics of Region 1 of RAMSES.

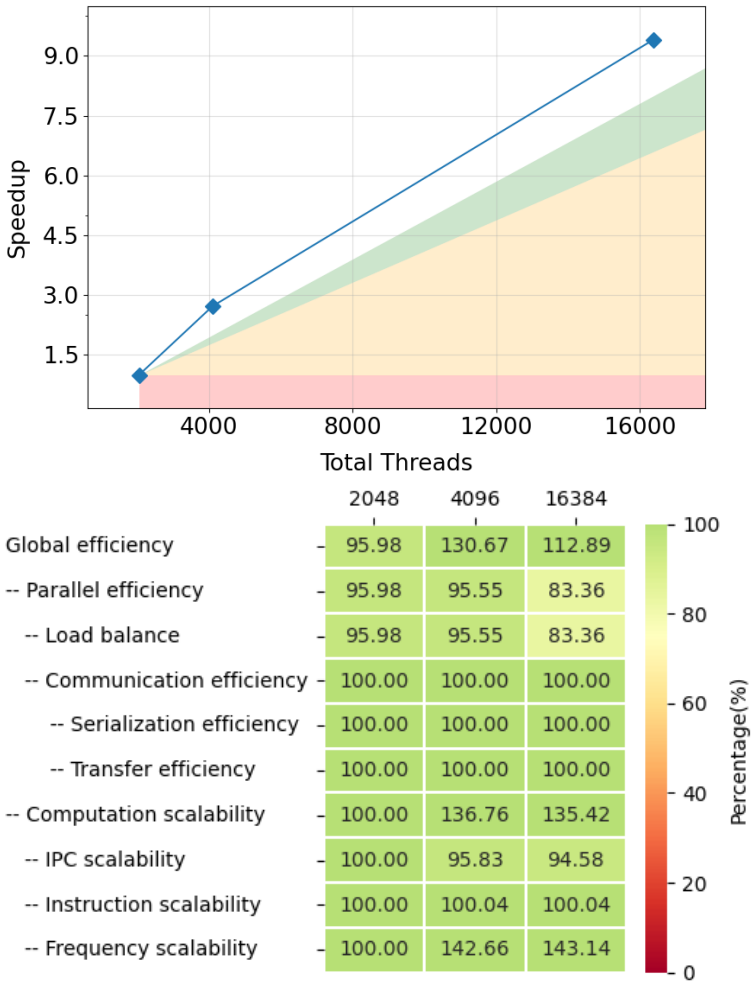


Figure 59: Strong scaling and POP efficiency metrics for Region 1 of RAMSES.

The zoomed-in trace for Region 1 is shown in Figure 59. From Table 23 and POP metrics shown in Figure 59 we can see that this region exhibits a super-linear scaling. In this case, the reason is the Frequency scaling. As this region is embarrassingly parallel (with no communication or synchronization point), the super-linear scaling is most probably due to the strong scaling procedure. Indeed, when the number of resources increases, we process a smaller amount of data per node, and a higher fraction of data probably fits into CPU caches. In this case, the region already looks well optimized on CPU architectures. The challenge might reside in porting this code on advanced hardware platforms, focusing on different types of GPU accelerators or ARM processors with potentially longer vector lanes (SVE).

6.4 Update from other MPI processes - Region 2

Region 2 corresponds to a first round of communication between MPI processes. Once all the hydrodynamical fluxes have been computed in Region 1, the hydrodynamical fluxes between neighboring MPI domains have to be communicated. Hence, Region 2 consists in updating one MPI process from its direct neighbors. It is currently the principal bottleneck regarding performance scaling when the number of MPI processes is too large: the ratio surface of the boundary region versus size of the MPI domain increases and too much time is spent in the MPI communication. The region contains asynchronous (Isend and Irecieve) communication operations and the `MPI_Waitall` operations which also highly depends on the load balancing between MPI domains. It has been selected for its potential optimisation in the MPI communication and in the load balancing.

```
! Reverse update boundaries
do ivar=1,nvar
  call make_virtual_reverse_dp(unew(1,ivar),ilevel)
end do

! Set uold equal to unew
call set_uold(ilevel)

! Restriction operator
call upload_fine(ilevel)
```

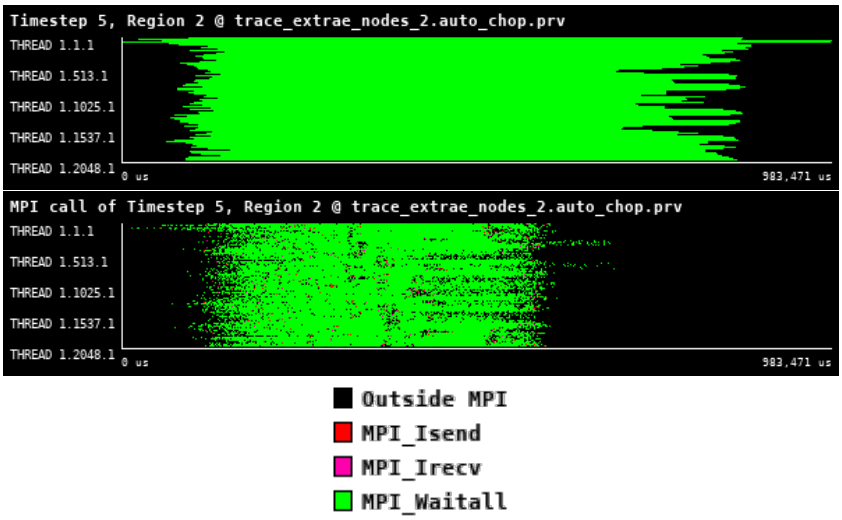


Figure 60: Zoomed traces for Region 2 of the RAMSES from Paraver showing the structure of the region.

Number of processes	2048	4096	16384
Elapsed time (sec)	0.813487	0.374387	0.234904
Efficiency	1.0	1.086426	0.432883
Speedup	1.0	2.172851	3.463062
Average IPC	0.704536	0.507323	0.588441
Average frequency (GHz)	0.250733	0.420925	0.866744

Table 24: Overview of the key performance metrics of Region 2 of RAMSES.

This document is Public (PU)
and was produced under SPACE project EU GA 101093441

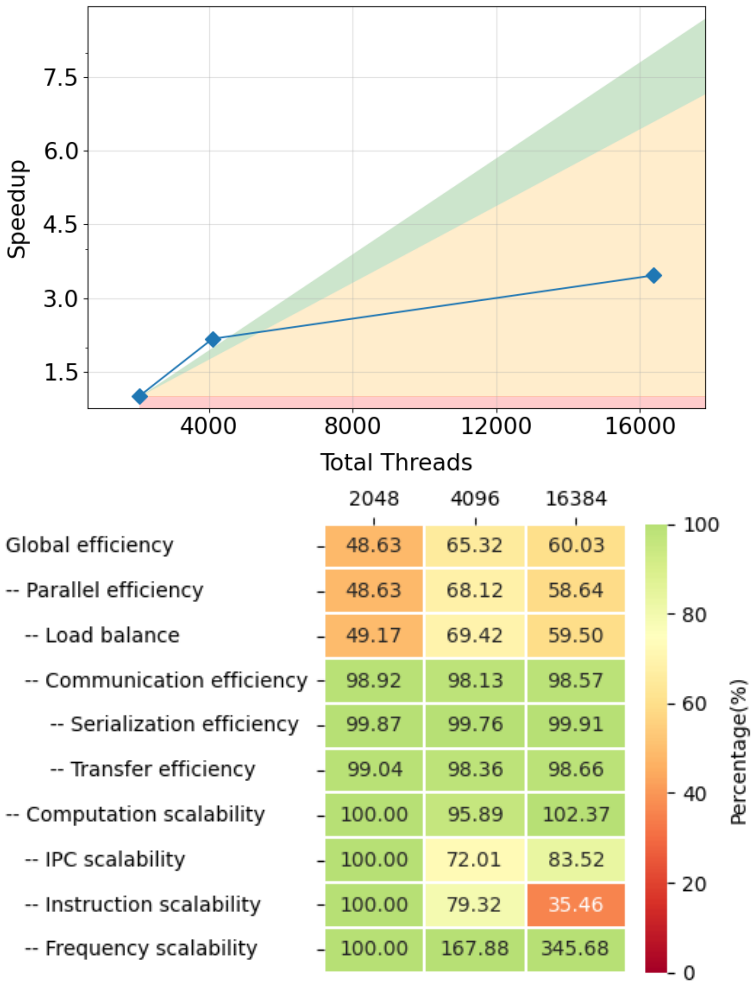


Figure 61: Strong scaling and POP efficiency metrics for Region 2 of RAMSES.

The timeline of the region in Figure 60 shows that the nearest neighbor communication pattern implemented in this function is purely based on non-blocking (`MPI_Isend` and `MPI_Ireceive`) communications synchronized by the Waitall operations. This approach already enables the overlap of communications and computation. We can see that the region, even though mostly communication bound, remains scalable due to fact the amount of data needed to be exchanged among neighboring regions is reduced as we scale. However this pattern as certain scale becomes a latency bound and stops scale.

The POP analysis of this region, see Table 24 and Figure 61, confirms the above observations and identifies key issues such as instruction scaling and load imbalance. We have already mentioned above that this region is the target for intra-node optimization of the communication layer. A first attempt will be to update the MPI communications to the MPI3 standard.

6.5 Update boundaries - Region 3

Region 3 is the final step corresponding to communication between MPI processes. Once the hydrodynamic update is completed and all values in the computational domain are synchronized in time, the new hydrodynamical states have to be communicated between the domains ghost regions in order to proceed to the next iteration. It represents a direct communication of the updated hydrodynamical values from one process to its neighbors. Only the ghost regions are affected by this communication. As for the previous region, it has been selected for its potential optimization in the MPI communications and in the load balancing.

```
!------------------------------------
! Update physical and virtual boundaries
!------------------------------------
do ivar=1,nvar
  call make_virtual_fine_dp(uold(1,ivar),ilevel)
end do
```

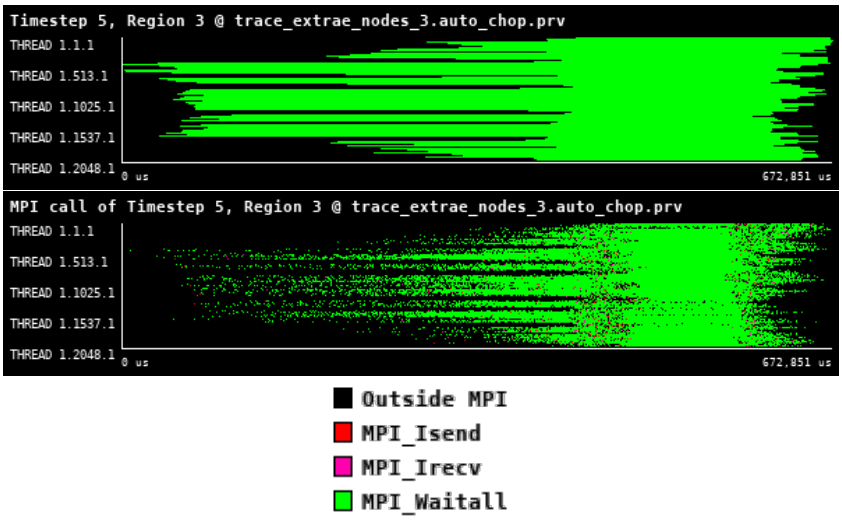


Figure 62: Zoomed traces for Region 3 of the RAMSES from Paraver showing the structure of the region.

Number of processes	2048	4096	16384
Elapsed time (sec)	0.625426	0.211412	0.199042
Efficiency	1.0	1.479164	0.392773
Speedup	1.0	2.958328	3.142181
Average IPC	0.755311	0.499686	0.600620
Average frequency (GHz)	2.077878	3.128745	3.013392

Table 25: Overview of the key performance metrics of Region 3 of RAMSES.

This document is Public (PU)
and was produced under SPACE project EU GA 101093441

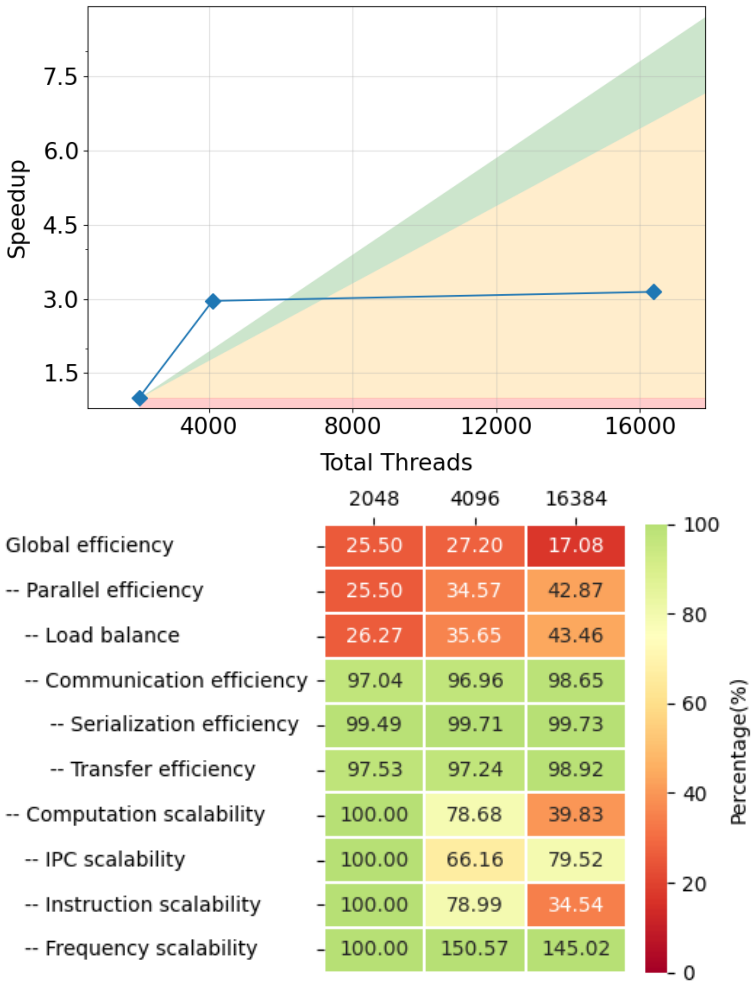


Figure 63: Strong scaling and POP efficiency metrics for Region 3 of RAMSES.

The timeline of the region is shown in Figure 62 and the analysis of this region is in Table 25 and Figure 63. The region exhibits the same behavior as the previous one, as it is also mainly communication based and it also contains the nearest-neighbor communication pattern. The proposed optimization is to further analyze the pattern and evaluate MPI3 standard to improve overlapping of communication and computation.

6.6 Conclusions

The performance analysis on this use-case for RAMSES shows a good scaling of the purely computational parts of the code (Region 1) but highlights the current limitation of the MPI decomposition and communications at a high number of MPI processes (Regions 2 and 3).

All the regions will thus be selected for optimisation using different strategies. Region 1 will be targeted for kernel optimisation. The first obvious step is to find the optimal `nvector` size on each architectures. This can be already done using our benchmark strategy. Then, Region 1 will be optimized to run on the next generation of hardware platforms (GPU accelerators, ARM processors). We will also consider the granularity of the AMR structure (using super-oct of size $< 2^{Ndim}$ for instance).

For regions 2 and 3, we will first investigate the strategy for ghost region communications. For instance, communications between domains can be handled by a single MPI using the MPI shared memory model. In addition, we will also investigate the possibility to enable the overlap of computation at the center of domains and communications at the boundaries. If successful, this optimisation will enable to run the current version of RAMSES (CPU only) more efficiently and have a important impact on all the RAMSES community. Second, we will consider the improvement of the load balancing strategy in RAMSES. We will use a simplified version of the code (only hydrodynamics and gravity), but which employs a hash table for domain decomposition. If successful, we will import the previously mentioned optimisation in this new version. This is a long-term goal.

This document is Public (PU)
and was produced under SPACE project EU GA 101093441

7 BHAC

The Black Hole Accretion Code (BHAC) [20, 21, 22, 23] is a multidimensional General Relativistic Magnetohydrodynamics (GRMHD) code that solves the equations of ideal GRMHD in one, two or three dimensions in order to perform (magneto)hydrodynamical simulations of accretion flows onto compact objects in arbitrary stationary space-times (Cowling approximation) using an efficient block based approach. BHAC is build upon the MPI-Adaptive Mesh Refinement-Versatile Advection Code (MPI-AMRVAC). MPI-AMRVAC [24, 25] is a parallel adaptive mesh refinement framework aimed at solving (primarily hyperbolic) partial differential equations (PDEs) by a number of different numerical schemes. The framework supports 1D to 3D simulations, in a number of different geometries (Cartesian, cylindrical, spherical). MPI-AMRVAC is written in Fortran 90 and uses MPI for parallelization.

BHAC is using the finite volume method to solve the equations of GRMHD in conservation form

$$\partial_t \mathbf{U} + \partial_i \mathbf{F}^i = \mathbf{S}, \tag{18}$$

where the vector of the conserved variables, the physical fluxes and the source read respectively:

$$\mathbf{U} = \sqrt{\gamma}\begin{bmatrix} D \\ S_j \\ \tau \\ B^j \end{bmatrix}, \quad \mathbf{F}^i = \sqrt{\gamma}\begin{bmatrix} V^i D \\ \alpha W^i_j - \beta^i S_j \\ \alpha(S^i - v^i D) - \beta^i \tau \\ V^i B^j - B^i V^j \end{bmatrix}, \quad \mathbf{S} = \sqrt{\gamma}\begin{bmatrix} 0 \\ \frac{1}{2}\alpha W^{ik}\partial_j \gamma_{ik} + S_i \partial_j \beta^i - U \partial_j \alpha \\ \frac{1}{2} W^{ik}\beta^j \partial_j \gamma_{ik} + W^j_i \partial_j \beta^i - S^j \partial_j \alpha \\ 0 \end{bmatrix}. \tag{19}$$

In addition to the density D, the covariant three-momentum S_j, the rescaled energy density τ, the Eulerian magnetic three-fields B^j, the lapse function α, the shift vector β^i, the 3-metric γ_{ij} and its trace γ, in the expressions above appear the transport velocity $V^i := \alpha v^i - \beta^i$, the fluid three-velocity v^i, the purely spatial part of the stress-energy tensor W^{ij} and the conserved energy density U. A number of different equations of state have been implemented in BHAC to close the system (18)-(19).

In addition to the conserved variables $\mathbf{U}$, knowledge of the primitive variables $\mathbf{P}(\mathbf{U})$ is also required for the calculation of fluxes and source terms. The primitives are the following

$$\mathbf{P} = [\rho, \Gamma v^i, p, B^i],$$

where ρ is the rest-mass density, Γ is the Lorentz factor and p is the pressure. While the transformation $\mathbf{U}(\mathbf{P})$ is straightforward, the inversion $\mathbf{P}(\mathbf{U})$ is highly non-trivial. Following the machinery of MPI-AMRVAC, we also extend here the conserved variables by the auxiliary variables $A = [\Gamma, \xi]$, where $\xi = \Gamma^2 \rho h$ with h the specific enthalpy. Knowledge of $\mathbf{A}$ enables the inversion $\mathbf{P}(\mathbf{U})$. The issue of inversion then becomes a matter of finding an $\mathbf{A}$ consistent with both $\mathbf{P}$ and $\mathbf{U}$.

BHAC is second-order accurate and employs a variety of multi-step Runge-Kutta schemes to temporally integrate the cell-average of the conserved variables through the computational grid.

7.1 Use-case description

As an use-case we simulate a 2D prograde weakly magnetised torus around a Kerr black hole using AMR. Such a torus solution for a fluid with constant angular momentum was first discussed in the eighties by Fishbone and Moncrief and is now a standard test for GRMHD simulations. Given the black hole mass and spin and the fluid equation of state, the solutions are defined by choosing the radius of the inner edge of the torus and the radius of the pressure maximum. The weak poloidal magnetic field that is used to introduce magnetorotational instability (MRI) to the solution is defined by the vector potential $A_\phi \propto \max(\rho/\rho_{max} - 0.2, 0)$, where ρ_{max} is the maximum initial density. The simulations are performed using horizon penetrating logarithmic Kerr-Schild coordinates.

Input file for the magnetised torus accretion test:

```
&filelist
    primnames       = 'rho u1 u2 u3 p b1 b2 b3 s tr1 lfac xi'
    autoconvert     = .true.
    saveprim        = .true.
    convert_type    = 'vtuBCCmpi'
    slice_type      = 'vtuCC'
    filenameini     = 'output/data'
    filenameout     = 'output/data'
    filenamelog     = 'output/amrvac'
```

```
   nwauxio          = 20
   typeparIO        = 0
&end

&savelist
   itsave(1,1)      = 0
   itsave(1,2)      = 0
   itsave(1,5)      = 0
   ditsave(1)       = 1
   dtsave(2)        = 1
   dtsave(5)        = 0.1

&end

&stoplist
   itmax            = 1000
   tmax             = 5000
   dtmin            = 1.d-6
&end

&methodlist
   wnames           = 'd␣s1␣s2␣s3␣tau␣b1␣b2␣b3␣Ds␣dtr1␣lfac␣xi'
   typeadvance      = 'twostep'
   typefull1        = 13*'tvdlf'
   typelimiter1     = 13*'ppm'
   typeemf          = 'uct2'
   tlow             = 1.0d-6
   strictgetaux     = F
   typeinversion    = '2D1DEntropy'
   typeaxial        = 'spherical'
&end

&boundlist
   dixB             = 4
   typeB            =
                    5*'noinflow',3*'cont',4*'noinflow'
                    5*'noinflow',3*'cont',4*'noinflow'
                    'symm','symm','asymm',2*'symm','symm','asymm', 5*'symm'
                    'symm','symm','asymm',2*'symm','symm','asymm', 5*'symm'
   primitiveB(1,1)  = .true.
   primitiveB(2,1)  = .true.
   primitiveB(1,2)  = .false.
   primitiveB(2,2)  = .false.
   internalboundary = .true.
&end

&amrlist
   mxnest             = 2
   nxlone1            = 192
   nxlone2            = 96
   xprobmin1          = 0.19325057145871735
   xprobmax1          = 7.824046010856292
   xprobmin2          = 0.0d0
   xprobmax2          = 0.5d0
   prolongprimitive   = T
   coarsenprimitive   = T
   restrictprimitive  = T
&end

&paramlist
   slowsteps         = 0
   courantpar        = 0.7d0
   typecourant       = 'maxsum'
&end
```

The number of AMR levels is controlled with mxnest and the resolution with nxlone1 and nxlone2.

7.2 High-level code structure

BHAC does both the initialization as well as the advancing of the variables of the governing PDEs. The main program is in the file amrvac.t and all time-advancing actually happens in advance.t. Input and output routines are in amrio.t and also in the postprocess conversion part collected in convert.t. Subroutines likely to be modified by the user are to be collected in amrvacusr.t. The explicit temporal discretizations are in the main advancing module advance.t. A rough scheme of BHAC's workflow follows:

```
START SIMULATION
   |
   |  <-------------- readcommandline, readparameters, initialize_vars
   |
READ SNAPSHOT  (only for restarts)
   |
INITGLOBALDATA_USR
INITGLOBALDATA
INITLEVELONE  (initialize grid level one)
SETTREE             (initialize finer grid levels)
   |
TIMEINTEGRATION <------------------
   |                              |
   |              --------------> |
   |              |               |
   |              |             SETDT
   |              |             SAVEAMRFILE
   |              |             RESETTREE
   |              |             ADVANCE  <---------|
   |              |               |                |
   |              ------ no -- | stop?           ADVECT <------- loop over grids ---- PROCES1_GRID
   |                           |                 ADD_SOURCE <--- loop over grids ---- ADDSOURCE1_GRID
   |                         yes
   |                          |
   |                          |
   |<--------------------------
   |
SAVEAMRFILE
   |
END SIMULATION
```

The main time cycle of BHAC looks like with the Regions 1 and 2 indicated:

```
program bhac
...
time_evol : do
   call extrae_event(base + region1 + iter2, event_start)   <----------------------------------
   ...                                                                                        |
   if (it>=itmax) exit time_evol                                                              |
   if (time_accurate .and. t>=tmax) exit time_evol                                            |
   ...                                                                                        |
   call extrae_event(base + region2 + iter2, event_start)         <-----------               |
   call advance(it) ! does time integration of all grids on all levels       | Region 2      | Region 1
   call extrae_event(base + region2 + iter2, event_stop)          <-----------               |
   ...                                                                                        |
   it = it + 1                                                                                |
   if (time_accurate) t = t + dt                                                              |
   ...                                                                                        |
   call extrae_event(base + region1 + iter2, event_stop)    <----------------------------------
 end do time_evol
end program bhac
```

`advance()` is called until a final time or a specific number of iterations has been reached. `advance()` calls `advect()` which integrates all grids by one full time-step using, in our case, the two stages predictor-corrector method.

```
subroutine advect()
...
case ("twostep")
   call advect1(...,t,...) ! predictor step
   call advect1(...,t+half*dt,...) ! corrector step
...
end subroutine advect
```

`advect1()` integrates all grids by one partial intermediate time-step. `advect1()` calls `advect1_grid()` which integrates one grid by one partial intermediate time-step. In `advect1_grid()` the primitive reconstruction happens, the sourses are added and the numerical fluxes are computed.

```
subroutine advect1_grid()
...
```

```
call primitive() ! primitive reconstruction
...
call tvdlf()
...
call addsource()
...
end subroutine advect1_grid
```

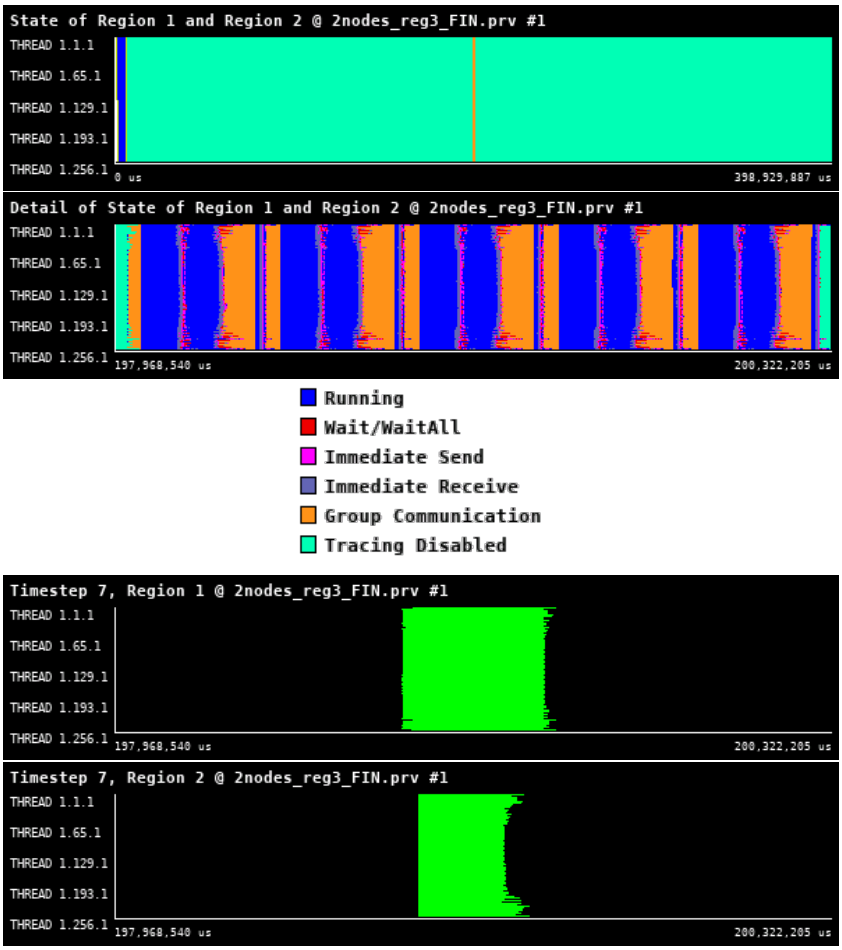


Figure 64: Traces for all regions in BHAC from Paraver showing the high-level structure of the code.

Figure 64 shows several iterations of the time integration region described above. There is a clear pattern showing iterations, which are combined of processing regions (blue) and communication regions. For tracing we have enabled tracing only for several iterations, as otherwise the traces became extremely large; see region "Tracing Dissabled". A single time step region is depicted as Region 1 and the most computationally demanding region is annotated as Region 2 (`advance()` routine). Both are analyzed below.

7.3 Single time step - Region 1

The `timeintegration()` routine is responsible for the temporal update of the semi-discrete form of (18). A multi-step Runge-Kutta scheme is employed to evolve the average state in the cell $\mathbf{U}_{i,j,k}$. At each sub-step, the point-wise interface fluxes $\mathbf{F}^i = (\mathbf{F}^x, \mathbf{F}^y, \mathbf{F}^z)$ are obtained by performing a limited reconstruction operation of the cell-averaged state $\mathbf{U}$ at the interfaces and employing approximate Riemann solvers.

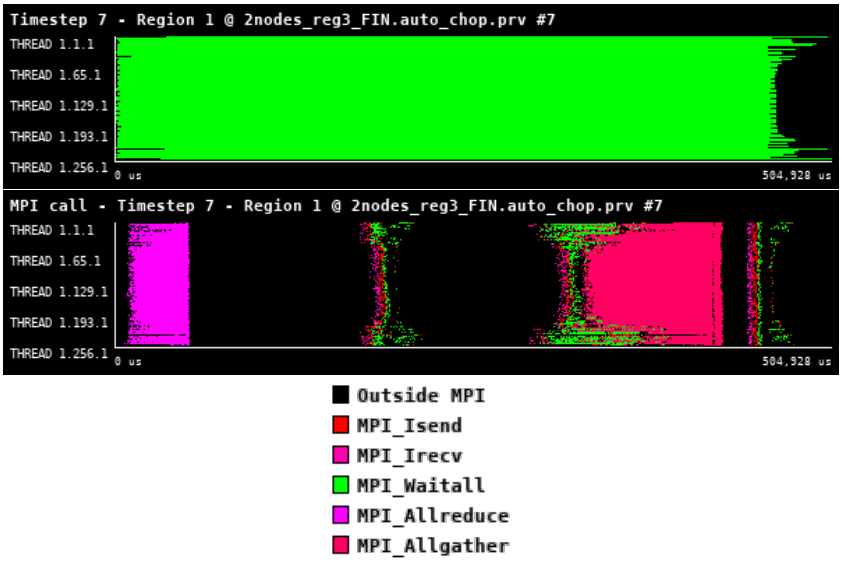


Figure 65: Zoomed traces for 7th timestep of Region 1 of the BHAC from Paraver showing the structure of the region.

Number of processes	256	512	1024	2048	4096
Elapsed time (sec)	0.453523	0.255785	0.199417	0.247043	1.831297
Efficiency	1.0	0.886532	0.568561	0.229476	0.015478
Speedup	1.0	1.773063	2.274244	1.835806	0.247651
Average IPC	2.161858	2.146241	2.149229	2.153679	3.016146
Average frequency (GHz)	2.978609	2.881363	2.646965	2.353549	1.555294

Table 26: Overview of the key performance metrics of Region 1 of BHAC.

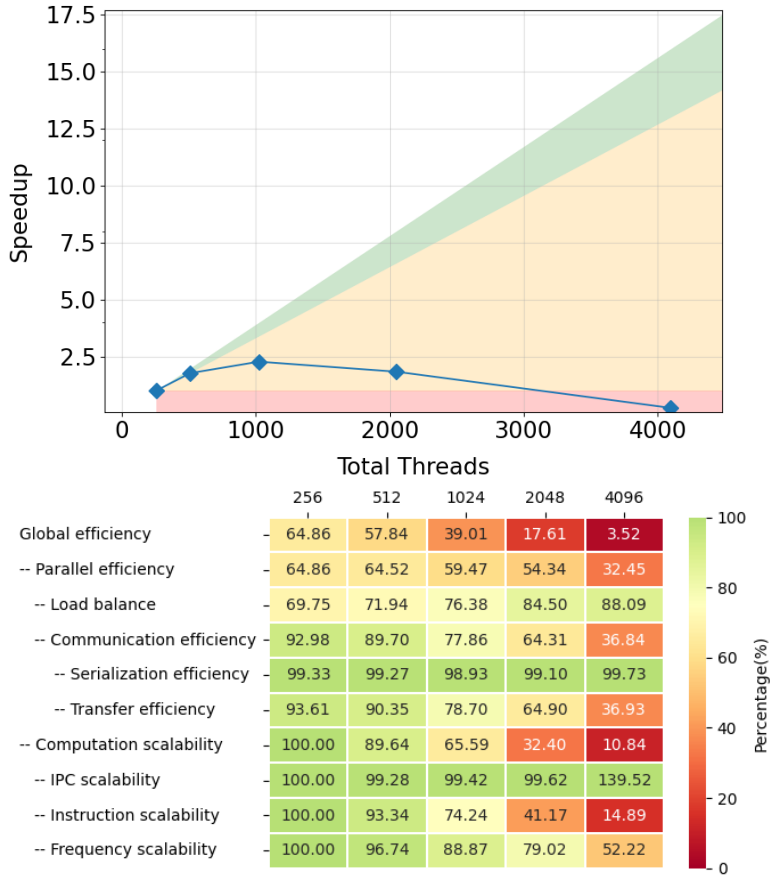


Figure 66: Strong scaling and POP efficiency metrics for Region 1 of BHAC.

Figure 65 shows that single time step consists of several phases. The first phase is a collective operation, `MPI_Allreduce`, which also synchronizes all the process. After this follows the Region 2 (`advance()` function (to be described in the next section)). After its execution there is another significant MPI collective call (`MPI_Allgather`), which has even higher impact on the execution time caused also by the load-imbalance of Region 2.

The analysis of this region, see Table 26 and Figure 66, shows two key factors limiting the scalability of the code. The first is the instruction scalability, which shows that as we scale the code, the amount of instruction needed to process this region remains almost the same. The second most significant limiting factor is the transfer efficiency, pointing out a large amount of communications. The ration between communication and computation is also caused by the fact that we choose a rather small test case in order to highlight the causes of the limited parallel efficiency.

7.4 `advance()` routine - Region 2, 7th Timestep

`advance()` is the core of BHAC as the entire time-advancing of the conserved variables **U** happens there. In addition, primitive recovery is nested in `advance()` which is the most resource demanding module of BHAC. This is because there are three inversion strategies implemented in BHAC, where the one acts as a backup for the other in the case that the former fails to converge. We have isolated `advance()` from the rest of the time-loop in order to study its efficiency with as few as possible communication overhead.

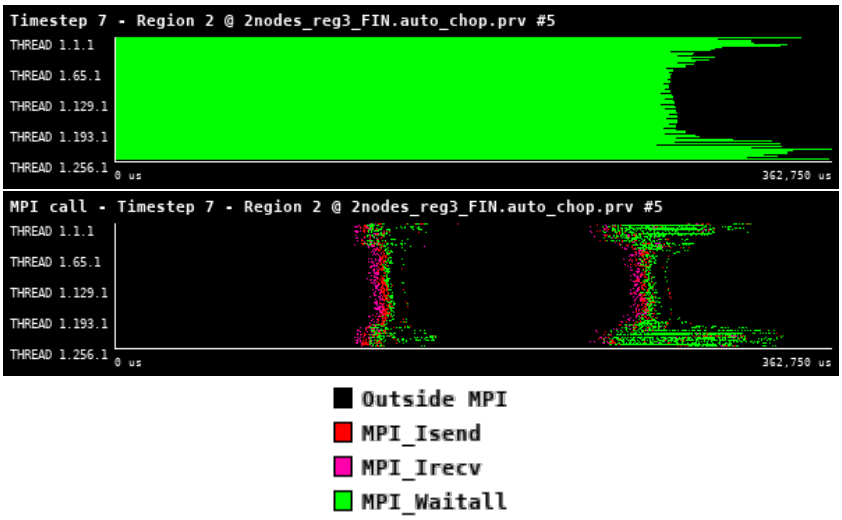


Figure 67: Zoomed traces for 7th timestep of Region 2 of the BHAC from Paraver showing the structure of the region.

Number of processes	256	512	1024	2048	4096
Elapsed time (sec)	0.235972	0.114527	0.05783	0.030077	0.082567
Efficiency	1.0	1.030202	1.020111	0.980699	0.178622
Speedup	1.0	2.060405	4.080443	7.845596	2.857946
Average IPC	2.312116	2.337873	2.367483	2.374932	2.385530
Average frequency (GHz)	1.957517	1.956692	1.954776	1.947031	1.896571

Table 27: Overview of the key performance metrics of Region 2 of BHAC.

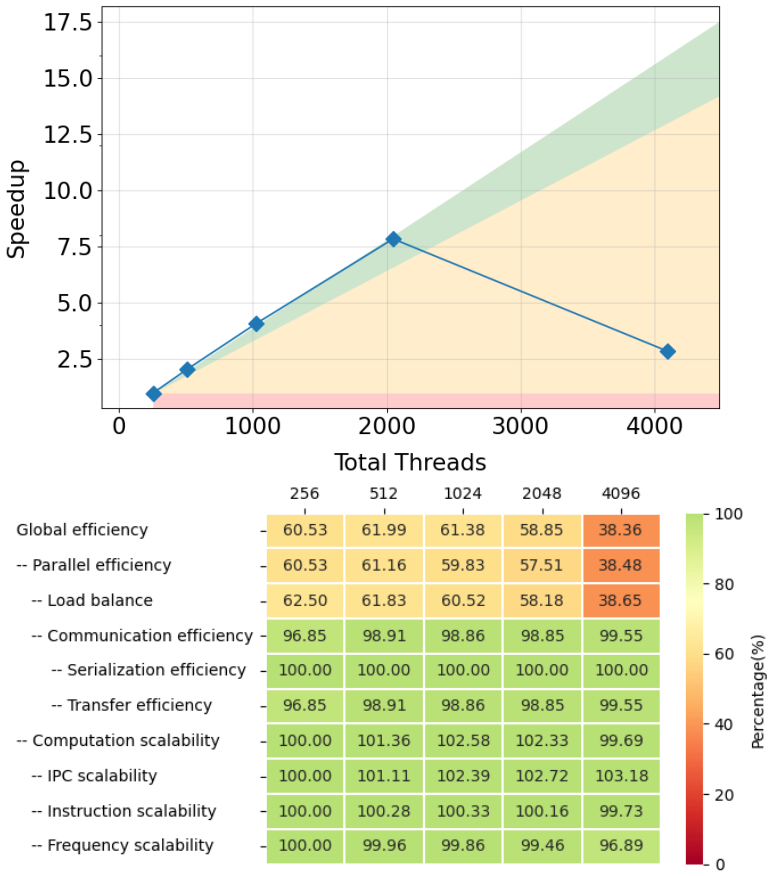


Figure 68: Strong scaling and POP efficiency metrics for Region 2 of BHAC.

The Region 2, see Figure 67, focuses on the evaluation of the `advance()` routine, which contains most of the local independent computations of the time integration step. It also contains two sets of point-to-point communications, but efficiently implemented in a non-blocking fashion and synchronized by the `MPI_Waitall` function.

From the analysis of this region, see Table 27 and Figure 68, one can see that both computation and communication efficiency are very good and the only cause of the limited parallel efficiency is the load imbalance, which is present even for a low number of processes. We recommend focusing the optimization mainly on this issue. Please note that for 4096 MPI processes there is a significant change in the trend speedup. The only indication we have is a significant change in load imbalance. But this does not entirely explain the behavior. As of now we consider this to be an outlier measurement sample.

7.5 Conclusions

There is an evident stark discrepancy between the efficiency and the scalability of the single time step (Region 1) and of one of its constituents `advance()` (Region 2).

Region 2 is scaling well and has good communication efficiency. The distribution of the computation load across the available computational resources is problematic even for a low number of processes, which caps the parallel efficiency. So we should focus on resolving this load imbalance in Region 2.

Once the issue with load balancing is tackled, this region has a potential to become a computational-focused kernel that has a potential to take advantage of single-core optimizations (vectorization, etc.) or porting to GPU accelerators. One should also evaluate the granularity (size of the problem per node or accelerator) of the problem to see if it is relevant to move to accelerators.

Region 1 on top of the Region 2 also adds several MPI collective operations, like `MPI_Allreduce` or `MPI_Allgather`, which, together with load imbalance resulting from Region 2, causes the main source of parallel inefficiency. Please note that a collective operation essentially behaves as a synchronization point and forces all processes to wait for the latest one to finish. Therefore, we recommend detailed evaluation of the communication patterns and implementation of the communication layer to find the potential for overlapping of communication and computation, by for instance using MPI-3 non-blocking collectives.

8 FIL

FIL is a module of the Einstein Toolkit (ET). The ET is a publicly available evolution framework that is designed, maintained and extended to enable numerical relativity simulations. Modules of the ET are known as Thorns. FIL is used in conjunction with other Thorns of the ET to solve the General Relativistic Magneto-Hydrodynamic (GRMHD) equations in 3-dimensions. FIL can therefor simulate magnetic fields in dynamical space-times allowing for the study of binary neutron star and black hole-neutron star mergers. Further information on both FIL and the ET can be found here [26] and here [27] respectively.

Einstein's field equation is given as

$$G^{\mu\nu} = 8\pi T^{\mu\nu}, \tag{20}$$

where $G^{\mu\nu}$ is the Einstein tensor, $T^{\mu\nu}$ is the stress-energy tensor and μ and ν are indices between 0-3 for the four dimension of space-time, 0 being the time dimension and 1-3 being the spatial dimensions. Note for the rest of this section Greek indices will span the whole 4D space whilst Latin indices will span only the 3 spatial dimensions. FIL evolves the RHS of (20), the stress energy tensor, using the system of ideal MHD equations which can be written in conservative form as

$$\partial_t \mathbf{U} + \partial_i \mathbf{F}^i = \mathbf{S}, \tag{21}$$

where the vector of the conserved variables, the physical fluxes and the source read respectively:

$$\mathbf{U} = \begin{bmatrix} \rho_* \\ \tilde{\tau} \\ \tilde{S}_i \\ \tilde{B}^i \end{bmatrix}, \quad \mathbf{F}^i = \begin{bmatrix} \rho_* \nu^j \\ \alpha^2\sqrt{\gamma}T^{0j} - \rho_* \nu^j \\ \alpha\sqrt{\gamma}T^j_i \\ V^j\tilde{B}^i - V^i\tilde{B}^j \end{bmatrix}, \quad \mathbf{S} = \begin{bmatrix} 0 \\ \frac{1}{2}\alpha W^{ik}\partial_j\gamma_{ik} + S_i\partial_j\beta^i - U\partial_j\alpha \\ \alpha\sqrt{\gamma}[(T^{00}\beta^i\beta^j + 2T^{0i}\beta^j + T^{ij})K_{ij} - (T^{00}\beta^i + T^{0i})\partial_i\alpha] \\ 0 \end{bmatrix}. \tag{22}$$

The conserved variables in $\mathbf{U}$ have no physical meaning they are constructed. In equation (22) the lapse function α; the shift vector β^i; the extrinsic curvature K_{ij}; the spatial 3 metric γ_{ik} and its determinant γ; the transport velocity $V^i := \alpha v^i - \beta^i$; the purley spatial part of the stress energy tensor W^{ik} and the fluid three-velocity ν^i appear. The vectors in equation (22) are dependent on the primitive variables, $\mathbf{P} = [\rho_0, P, v^i, B^i]$, those with physical meaning, where ρ_0 is the fluid rest mass density, P the pressure of the fluid, v^i the 3 velocity of the fluid and B^i the magnetic field as measured by Eulerian observers. The conservative variables can be written in terms of the primitive variables as

$$\mathbf{U} = \begin{bmatrix} \rho_* \\ \tilde{\tau} \\ \tilde{S}_i \\ \tilde{B}^i \end{bmatrix} = \begin{bmatrix} \alpha\sqrt{\gamma}\rho_0 u^0 \\ \alpha^2\sqrt{\gamma}T^{00} - \rho_* \\ (\rho_* h + \alpha u^0\sqrt{\gamma}b^2)u_i - \alpha\sqrt{\gamma}b^0 b_i \\ \sqrt{\gamma}B^i \end{bmatrix}, \tag{23}$$

where h is the specific enthalpy, u^i is the fluid four velocity and $b^\mu = B^\mu_{(u)}/\sqrt{4\pi}$ with $B^\mu_{(u)}$ being the magnetic field measured by an observer co-moving with the fluid.

During the simulations the conservative variables, not the primitives are evolved in time using a 4th order Runge Kutta method (RK4), which is provided by the Method of Lines (MoL) Thorn in the ET. Evolving the conservative variables has a number of numerical advantages; it insures the conservation of mass, energy and momentum to truncation error; it allows for the use of high order high-resolution shock capturing numerical schemes and it guarantees that the Rankine–Hugoniot shock jump conditions are satisfied.

To perform the time steps the source and the flux terms in equation (21) need to be evaluated. This is done by the `Illinois_GRMHD_driver_evaluate_MHD_RHS_type` function. During and at the end of a time step the primitive variables need to be recovered, this is done by inverting equation (23). The inversion of equation (23) is handled in FIL by the `illinoisGRMHD_conserv_to_prims` function which employs a Newton–Raphson-based root finder to calculate the inversion. As there are more primitive variables then conservatives an equation of state must be used to close the system of equations. Users are able to define their own EOS which is especially useful for the study of neutron stars where the EOS is still unknown. The Illinois driver and conservative to primitive functions from FIL were chosen to be profiled as they are the most computationally expensive functions in FIL. The whole of the main evolution time loop was also profiled, the time loop contains code from other Thorns in the ET, it is of considerable interest how the ET scales compared to FIL.

8.1 Use-case description

FIL has two important use cases that can be leveraged to ensure the numerical algorithms are properly stressed and check the accuracy of the numerical results. The first use case is that of head-on collisions where two compact objects either start at rest or have some initial linear velocity directed at the companion. The head-on use case will be the primary use case for the FIL code as it can both stress the algorithm implementation while still being able to complete a run in a reasonable amount of time. The second and most relevant to multi-messenger astronomy is the in-spiral, merger, and post-merger of two neutron stars. Although this test case is more relevant for research purposes, the evolution of the in-spiral is very long and does not stress the numerical implementation to the degree necessary to ensure future optimizations still yield reliable physical results. The parameter file for the head on collision test case can be seen bellow:

```
ActiveThorns = "AEILocalInterp␣Constants␣Fortran␣GSL␣Watchdog␣"
ActiveThorns = "SystemTopology"
################################################
# Termination
################################################
Cactus::terminate              = "iteration"
Cactus::cctk_itlast                          = 16
Cactus::cctk_full_warnings             = no
Cactus::highlight_warning_messages     = no

################################################
# Timer report
################################################
ActiveThorns           = "TimerReport"
TimerReport::out_every             = 200

################################################
# Carpet
################################################
ActiveThorns = "Carpet␣CarpetLib␣CarpetInterp␣LocalInterp2␣CarpetReduce␣LoopControl␣Slab"
Carpet::verbose                = no
Carpet::veryverbose            = no
Carpet::schedule_barriers      = no
Carpet::storage_verbose        = no
CarpetLib::output_bboxes       = no
Carpet::domain_from_coordbase        = yes
driver::ghost_size             = 4
Carpet::use_buffer_zones       = yes
Carpet::prolongation_order_space       = 5
Carpet::prolongation_order_time      = 2
Carpet::enable_all_storage                    = "no"
Carpet::convergence_level        = 0
Carpet::init_fill_timelevels       = yes
Carpet::poison_new_timelevels        = yes
CarpetLib::poison_new_memory         = yes
CarpetLib::combine_sends                   = "no"
CarpetLib::check_bboxes                    = "no"
CarpetLib::interleave_communications       = "yes"
CarpetLib::combine_recompose               = "no"

################################################
# Grid
################################################
ActiveThorns = "Boundary␣CartGrid3D␣CoordBase␣SymBase"
CoordBase::domainsize        = "minmax"
CoordBase::xmin          = -1024.0
CoordBase::ymin          = -1024.0
CoordBase::zmin          = -1024.0
CoordBase::xmax          = 1024.0
CoordBase::ymax          = 1024.0
CoordBase::zmax          = 1024.0

CoordBase::spacing                         = "numcells"
CoordBase::ncells_x                        = 128
CoordBase::ncells_y                        = 128
CoordBase::ncells_z                        = 64

CoordBase::boundary_size_x_lower           = 4
CoordBase::boundary_size_y_lower           = 4
CoordBase::boundary_size_z_lower           = 4
CoordBase::boundary_size_x_upper           = 4
CoordBase::boundary_size_y_upper           = 4
CoordBase::boundary_size_z_upper           = 4
CoordBase::boundary_shiftout_x_lower       = 0
CoordBase::boundary_shiftout_y_lower       = 0
CoordBase::boundary_shiftout_z_lower       = 0
```

```
CoordBase::boundary_shiftout_x_upper       = 0
CoordBase::boundary_shiftout_y_upper       = 0
CoordBase::boundary_shiftout_z_upper       = 0
ActiveThorns           = "ReflectionSymmetry"
ReflectionSymmetry::reflection_x     = "no"
ReflectionSymmetry::reflection_y     = "no"
ReflectionSymmetry::reflection_z     = "no"
ReflectionSymmetry::avoid_origin_x     = "no"
ReflectionSymmetry::avoid_origin_y     = "no"
ReflectionSymmetry::avoid_origin_z     = "no"
CartGrid3D::type         = "coordbase"
CartGrid3D::domain             = "full"
CartGrid3D::avoid_origin       = "no"

#=============== Refinement ====================
ActiveThorns = "CarpetRegrid2"
Carpet::max_refinement_levels        = 4
CarpetRegrid2::num_centres                 = 2
CarpetRegrid2::regrid_every                = 200
CarpetRegrid2::snap_to_coarse              = "yes"
CarpetRegrid2::freeze_unaligned_levels           = "yes"
CarpetRegrid2::freeze_unaligned_parent_levels    = "yes"

# ----------- Region 1 --------------------
CarpetRegrid2::active_1            = "yes"
CarpetRegrid2::num_levels_1              = 4
CarpetRegrid2::position_x_1              = 30.0
CarpetRegrid2::position_y_1              = 0.0
CarpetRegrid2::radius_1[1]               = 300.0
CarpetRegrid2::radius_1[2]               = 80
CarpetRegrid2::radius_1[3]               = 20.0
CarpetRegrid2::radius_1[4]               = 40.0

ActiveThorns="BNSTrackerGen"
BNSTrackerGen::sym_pi       = "no"
BNSTrackerGen::analysis_reflevel    = 5
BNSTrackerGen::analyze_every      = 200


################################################
# Time integrator
################################################
ActiveThorns = "MoL Time"
MoL::ODE_Method                  = "Euler"
MoL::MoL_Intermediate_Steps      = 1
MoL::MoL_Num_Scratch_Levels      = 1
MoL::initial_data_is_crap       = "yes"
InitBase::initial_data_setup_method       = "init_all_levels"
Time::dtfac           = 0.2

################################################
# Illinois GRMHD
################################################
ActiveThorns = "IllinoisGRMHD"
ActiveThorns = "ID_converter_ILGRMHD"
#============= Initial Data Import============================
ID_converter_ILGRMHD::pure_hydro_run      = "no"
ID_converter_ILGRMHD::rho_threshold     = 1.0e-16
ID_converter_ILGRMHD::adjust_mb           = no
#============= IllinoisGRMHD evolution  ================================
CarpetLib::support_staggered_operators    = "yes"
IllinoisGRMHD::damp_lorenz                = 1.0

################################################
# Equation of state
################################################
ActiveThorns = "Margherita_EOS Margherita_ID EOS_Base"

Margherita::eos_type        = "polytropic"
Margherita::K_EOS                = 100
Margherita::Gamma_EOS[0]           = 2.0

################################################
# spacetime
################################################
ActiveThorns = "ADMBase ADMCoupling ADMMacros CoordGauge SpaceMask StaticConformal TmunuBase InitBase"
ActiveThorns = "GenericFD NewRad"
ActiveThorns = "Antelope Dissipation"

SpaceMask::use_mask         = yes
TmunuBase::stress_energy_storage    = yes
TmunuBase::stress_energy_at_RHS     = yes
TmunuBase::timelevels               = 1
TmunuBase::prolongation_type        = "none"
```

```
TmunuBase::support_old_CalcTmunu_mechanism  = "no"
ADMMacros::spatial_order      = 4
HydroBase::prolongation_type      = "ENO"

ADMBase::metric_prolongation_type     = "none"
ADMBase::evolution_method     = "Antelope"
ADMBase::lapse_evolution_method     = "Antelope"
ADMBase::shift_evolution_method     = "Antelope"
ADMBase::dtlapse_evolution_method   = "Antelope"
ADMBase::dtshift_evolution_method   = "Antelope"

ADMBase::lapse_timelevels       = 3
ADMBase::shift_timelevels       = 3
ADMBase::metric_timelevels      = 3

Antelope::eta           = 1.4
Antelope::kappa1        = 0.02
Antelope::non_shifting_shift      = yes
Antelope::import_Tmunu        = no

Antelope::eta_damp_radius       = 50.
Antelope::theta_damp_radius       = 1.e99
Antelope::kappa_damp_radius       = 50.

Boundary::radpower                      = 2

Dissipation::order                      = 5
Dissipation::epsdis                     = 0.2
Dissipation::vars                       = "
␣␣Antelope::conformal_factor
␣␣Antelope::conformal_metric
␣␣Antelope::curvature_scalar
␣␣Antelope::Theta
␣␣Antelope::curvature_tensor
␣␣Antelope::Gamma
␣␣Antelope::Gammadriver
␣␣ADMBase::lapse
␣␣ADMBase::shift
"
################################################
# Initial data
################################################
ActiveThorns ="HydroBase␣SummationByParts␣tovsolver"
ActiveThorns="Seed_Magnetic_Fields_BNS"

Seed_Magnetic_Fields_BNS::enable_IllinoisGRMHD_staggered_A_fields = "yes"
Seed_Magnetic_Fields_BNS::P_cut = 1.e-7 #8.e-8 # 0.04% of maximum initial pressure
Seed_Magnetic_Fields_BNS::n_s = 1.0

Seed_Magnetic_Fields_BNS::have_two_NSs_along_x_axis = "yes"
Seed_Magnetic_Fields_BNS::r_NS1 = 10.0
Seed_Magnetic_Fields_BNS::r_NS1_low = 5.0
Seed_Magnetic_Fields_BNS::x_c1=30
Seed_Magnetic_Fields_BNS::x_c2=-30

Seed_Magnetic_Fields_BNS::A_field_type = "poloidal_A_interior"
Seed_Magnetic_Fields_BNS::A_b = [28.0*0.8]

Seed_Magnetic_Fields_BNS::g_width = 15.
Seed_Magnetic_Fields_BNS::g_r0 = 5.66


ADMBase::metric_type      = "physical"
ADMBase::initial_data       = "tov"
ADMBase::initial_lapse      = "tov"
ADMBase::initial_shift      = "tov"
ADMBase::initial_dtlapse      = "zero"
ADMBase::initial_dtshift      = "zero"

HydroBase::timelevels           = 3
HydroBase::initial_hydro          = "tov"
HydroBase::initial_Y_e            = "one"
HydroBase::initial_temperature    = "zero"
HydroBase::initial_entropy      = "zero"
HydroBase::initial_Bvec       = "zero"
HydroBase::initial_Avec       = "zero"
HydroBase::initial_Aphi       = "zero"

tovsolver::TOV_Num_TOVs           = 2
tovsolver::TOV_Num_Radial         = 200000
tovsolver::TOV_Combine_Method     = "average"
tovsolver::TOV_Gamma            = 2.0

tovsolver::TOV_K                = 100.0
```

```
#tovsolver::TOV_Velocity_x[0]     = 0.
tovsolver::TOV_Rho_Central[0]     = 1.28e-3
tovsolver::TOV_Position_x[0]      = -30.0

#tovsolver::TOV_Velocity_x[1]     = -0.1
tovsolver::TOV_Rho_Central[1]     = 1.28e-3
tovsolver::TOV_Position_x[1]      = 30.0

################################################
# NaN checker
################################################
ActiveThorns = "WatchDog NaNChecker"

NaNChecker::check_every         = 100
NaNChecker::verbose             = "standard"
NaNChecker::action_if_found     = "terminate"
NaNChecker::check_vars          = "
        ADMBase::metric
        ADMBase::curv
        ADMBase::lapse
        ADMBase::shift
    IllinoisGRMHD::u0
    IllinoisGRMHD::rho_star
"

################################################
# Spherical surfaces
################################################
ActiveThorns = "SphericalSurface"
SphericalSurface::nsurfaces    = 1
SphericalSurface::maxntheta    = 140
SphericalSurface::maxnphi    = 240

SphericalSurface::ntheta[0]    = 55
SphericalSurface::nphi[0]    = 96
SphericalSurface::nghoststheta[0] = 2
SphericalSurface::nghostsphi[0]   = 2

################################################
# Apparent horizon finder
################################################
ActiveThorns = "CarpetMask AHFinderDirect"
AHFinderDirect::N_horizons        = 1
AHFinderDirect::find_every        = 100
AHFinderDirect::output_h_every        = 100
AHFinderDirect::max_Newton_iterations__initial    = 50
AHFinderDirect::max_Newton_iterations__subsequent = 50
AHFinderDirect::max_allowable_Theta_growth_iterations = 10
AHFinderDirect::max_allowable_Theta_nonshrink_iterations= 10
AHFinderDirect::geometry_interpolator_name    = "Lagrange polynomial interpolation"
AHFinderDirect::geometry_interpolator_pars    = "order=4"
AHFinderDirect::surface_interpolator_name   = "Lagrange polynomial interpolation"
AHFinderDirect::surface_interpolator_pars   = "order=4"
AHFinderDirect::verbose_level       = "physics details"
AHFinderDirect::move_origins        = "yes"

AHFinderDirect::origin_x[1]       = 0.0
AHFinderDirect::initial_guess__coord_sphere__x_center[1]= 0.0
AHFinderDirect::initial_guess__coord_sphere__y_center[1]= 0.0
AHFinderDirect::initial_guess__coord_sphere__z_center[1]= 0.0
AHFinderDirect::initial_guess__coord_sphere__radius[1]  = 1.0
AHFinderDirect::which_surface_to_store_info[1]    = 0
AHFinderDirect::set_mask_for_individual_horizon[1]  = "no"
AHFinderDirect::reset_horizon_after_not_finding[1]  = "no"
AHFinderDirect::find_after_individual_time[1]   = 100.0
AHFinderDirect::max_allowable_horizon_radius[1]   = 5.0
AHFinderDirect::reshape_while_moving      = "yes"
AHFinderDirect::predict_origin_movement     = "yes"

CarpetMask::excluded_surface[0]       = 0
CarpetMask::excluded_surface_factor[0]      = 1
################################################
# Output
################################################
ActiveThorns = "SphericalSurface"
ActiveThorns = "CarpetIOASCII CarpetIOScalar CarpetIOHDF5 CarpetIOBasic"

IO::out_single_precision                  = "yes"
IOUtil::strict_io_parameter_check     = "yes"
IOUtil::parfile_write               = "copy"
CarpetIOScalar::all_reductions_in_one_file  = "yes"
CarpetIOScalar::one_file_per_group          = "yes"
CarpetIOASCII::one_file_per_group           = "yes"
CarpetIOHDF5::one_file_per_group            = "yes"
```

```
CarpetIOBasic::outInfo_criterion                = "divisor"
CarpetIOBasic::outInfo_every      = 10
CarpetIOBasic::outInfo_reductions= "maximum␣minimum␣norm1␣norm2"
IOBasic::outInfo_vars= "ADMBase::lapse␣IllinoisGRMHD::rho_b␣Antelope::H␣Carpet::physical_time_per_hour"

CarpetIOScalar::outScalar_criterion             = "divisor"
#CarpetIOScalar::outScalar_dir                   = "data_Scalar"
CarpetIOScalar::outScalar_every                 = 10
CarpetIOScalar::outScalar_reductions            = "minimum␣maximum␣norm1␣norm2"
CarpetIOScalar::outScalar_vars      = "HydroBase::rho␣Antelope::H"

CarpetIOASCII::out0D_criterion                  = "divisor"
#CarpetIOASCII::out0D_dir                        = "data_asc_0D"
CarpetIOASCII::out0D_every                      = 100
CarpetIOASCII::out0D_vars       = "Carpet::timing"

CarpetIOHDF5::out1D_criterion                   = "divisor"
#CarpetIOHDF5::out1D_dir                         = "data_asc_1D"
CarpetIOHDF5::out1D_every                       = 512
CarpetIOHDF5::out1D_vars      = "HydroBase::rho␣Antelope::H"

CarpetIOHDF5::out2D_criterion       = "divisor"
#CarpetIOHDF5::out2D_dir             = "data_hdf5_2D"
CarpetIOHDF5::out2D_every           = 512
CarpetIOHDF5::out2D_vars      = "HydroBase::rho␣Antelope::H"
```

8.2 High-level code structure

The ET is built around the Cactus framework a general framework for the development of portable modular applications, where programs are split into modules known as Thorns. As mentioned above FIL is a Thorn of the ET. The Cactus framework acts as the interface between different Thorns. Some important Thorns are; Carpet, responsible for the AMR grid simulations take place on; Antelope, the space-time evolution Thorn; MoL, provides the numerical time integrators; TOVSolver, integrates the TOV equations to provide the initial data for the neutron stars; and the Einstein base Thorns, AMDBase, Hydrobase and TmunuBase define the grid functions[2] for the basic space time variable. Respectively they define the 3 metric, the hydrodynamic variables and the stress energy tensor. Having these variables and more defined by the Einstein base Thorns is key to the Cactus framework, it allows for a clearly defined interface between different Thorns. Analysis, initial data and evolution Thorns will all be working on the same well defined grid functions. Antelope, the space-time evolution Thorn, can be used to solve the BSSN, CCZ4 and Z4c formulations of the Einstein's equations. It is important to understand that FIL evaluates the MHD variables and Antelope evaluates the equivalent space-time variables, both sets of variables are passed to the MoL Thorn which evolves both sets forward in time. Each Thorn has the following directory structure:

```
/Cactus/arrangements/
└── ArrangementName/
    └── ThornName/
        ├── COPYRIGHT
        ├── README
        ├── test/
        ├── doc/
        ├── par/
        ├── configuration.ccl
        ├── interface.ccl
        ├── schedule.ccl
        ├── param.ccl
        └── src/
            └── make.code.defn
```

The COPYRIGHT, README, doc/, test/ and par/ are optional; doc/ will typically be used to store documentation related to the Thorn, and par/ can have example parameter files for the Thorn. The src/ directory will store the code of the Thorn as will as the make.code.defn file which tells the ET which files to compile. FILs source code can be found in the Relastro_dev/IllinoisGRMHD/ directory. The .ccl files are the files which directly interact with the cactus framework. The configuration.ccl file contains inter-Thorn build dependencies and is optional; interface.ccl defines Thron-wide variables, grid functions and shared functions; param.ccl defines all parameters and sets their default values and schedule.ccl takes care of all the function scheduling and controls the global storage of all grid functions. At the start of each run of FIL the Cactus schedule is outputted: a complete list of every function, the order it is called in and some of the program structure such as loops. Bellow is an edited version of the scheduler output where effort has been made to reduce the size considerably and slight modifications have been made for the ease of reading. Included is the main time loop; details on the main time loop can be found in section 7.3.

```
################################################
#Initialisation of grid
################################################
Carpet::MultiModel_Startup                          #Init AMR grids
Antelope::MultiModel_Startup                        #Register Antelope with CoordGauge
Antelope::Antelope_Select_System                    #Register evolution system for the space-time
CartGrid3D::RegisterCartGrid3DCoords                #Register coordinates for the Cartesian grid

################################################
#Calculate initial data
################################################
Margherita_EOS::Margherita_setup_polytrope          #Setup piecewise polytropic EOS.
ADMBase_InitialData                                 #ADMBase initial data
Hydrobase::HydroBase_Initial                        #HydroBase initial data
TOVSolver::TOV_C_Integrate_RHS                      #Integrate the 1d equations for the TOV star
TOVSolver::TOV_C_Exact                              #Set up the 3d quantities for the TOV starr
```

[2]Grid functions are functions that are discretized and stored at every point on the grid.

```
Seed_Magnetic_Fields_BNS::Seed_Magnetic_Fields_Privt    #Set up binary neutron star seed magnetic fields.
TmunuBase::SetTmunu                                     #Calculate the stress-energy tensor
Antelope::Antelope_MoLRegister                          #Register the evolution variables with MoL

Boundary::BoundaryConditions                            #Set up boundary conditions

################################################
Main time loop
################################################


BNSTrackerGen::BNSTrackerGen_Track_Stars                #Track the stars position

do loop over time steps:
    BNSTrackerGen::BNSTrackerGen_Move_Grids             #Update positions of refined regions
    CartGrid3D::SpatialCoordinates                      #Set Coordinates after regridding

    do loop over refinement levels:
        Extrae_event(Region_0, 1)
        do loop over MoL substeps                       #The RK4 method has 4 sub steps
            MoL::MoL_StartStep: MoL                     #Internal setup for the evolution step
            MoL::MoL_AllocateScratch                    #Allocate sufficient space for scratch variables
            MoL::MoL_InitRHS                            #Initialise the RHS functions
            Antelope::Antelope_CalcFD                   #Computing RHS for spacetime evolution

            Extrae_event(region_1, 1)
            IllinoisGRMHD::IllinoisGRMHD_RHS_eval       #Evaluate RHSs of GR Hydro & GRMHD equations
            Extrae_event(region_1, 0)

            MoL::MoL_PostRHS                            #Modify RHS functions
            MoL::MoL_Add                                #Updates calculated with the Runge-Kutta 4 method
            MoL::MoL_DecrementCounter                   #Alter the counter number

            Boundary::BoundaryConditions                #Execute all boundary conditions
            IllinoisGRMHD::IllinoisGRMHD_compute_B_and_Bstagger_from_A
                                                        #Compute B and B_stagger from A
            Extrae_event(region_2, 1)
            IllinoisGRMHD::IllinoisGRMHD_conserv_to_prims
            Extrae_event(region_2, 0)

            IllinoisGRMHD::IllinoisGRMHD_outer_boundaries_on_P_rho_b_vx_vy_vz
            #Apply outflow-only, flat BCs on {P,rho_b,vx,vy,vz}.
        Extrae_event(Region_0, 0)
```

The Figure 69 captures integration step of two different refinement levels. Although, simulation domain consist of four levels in total, integration step in the all refinement levels can be categorized into two distinct groups based on behaviour. Therefore, we selected representative refinement level form each category for further study. Due to issues with the Extrae trace file size an Euler evolution scheme had to be used instead of the usual RK4 method. This is why there is only a single call of the driver and conservative to primitive functions within the main time loop instead of 4. For performance profiling an code optimisation this should not be an issue as improvements to a single time step will carry forward to the 4 sub time steps of the RK4 method. It is also clear that a time step for the lower refinement level is taking longer to complete whilst the driver and conservative to primitive functions seem to be taking the same amount of time. Whilst this could be due to more calculations that need to be done on a more refined but smaller grid it could also be due to poor optimisation. Isolating which part of the ET are responsible for the slower run times as the refinement level increases will help to answer this problem.

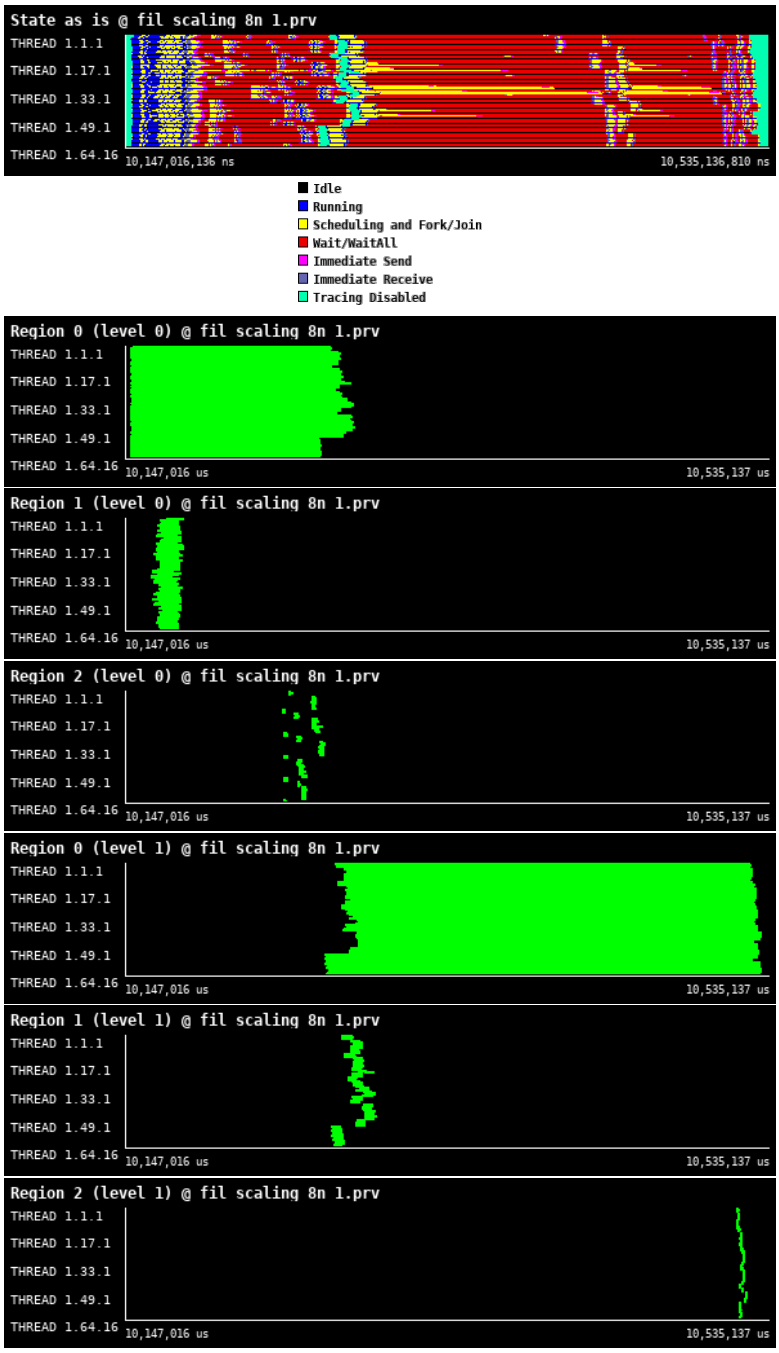


Figure 69: Traces for all regions in FIL from Paraver showing the high-level structure of the code.

8.3 Single time step structure

Here the basic algorithm that FIL uses is outlined, all tasks are performed by FIL unless otherwise stated. At the beginning of the time step the variables $[P, \rho_0, v^i, B^i, A_\mu, \phi]$ must be defined everywhere, where A_μ is the purely spatial components of a 4 vector potential and ϕ is the EM scalar potential, i.e. the temporal component of the 4 vector potential. These two variables are needed to evolve the magnetic field whilst insuring that the magnetic field's divergence remains zero everywhere. $[P, \rho_0, v^i]$ are defined everywhere using the the TOVSolver and HydroBase Thorns and $[B^i, A_\mu, \phi]$ are defined using the Seed Magnetic Fields BNS Thorn.

1. The first step in the evolution is to calculate the conservative variables from the primitive, unlike the inverse this is trivial. With the special treatment of the magnetic field the set of evolution variables are $E = [\rho_*, \tilde{\tau}, \tilde{S}_i, A^i, \sqrt{\gamma}\phi]$.

2. The evolution variables all obey (21). To evolve them in time the flux and the source terms need to be evaluated. For $\rho_*, \tilde{\tau}, \tilde{S}_i$ the derivatives of the space time field that appear in the source term, equation (22), are evaluated using a 4th order finite difference method and the fluxes are computed using a fourth order high resolutions shock capturing (HRSC) finite volume method. $[A^i, \sqrt{\gamma}\phi]$ are evaluated using a Vector-potential-based constrained transport scheme. Both steps (1) and (2) are performed by the `add_fluxes_and_source_terms_to_hydro_rhss` function within the `Illinois_GRMHD_driver_evaluate_MHD_RHS_type` function.

3. Time derivative data from all the evolved GRMHD variables outlined above and coming from Antelope are passed to the MoL Thorn which applies a RK4 method to iteratively evolve the variables forward in time.

4. The RK4 method requires three ghost zones at the outer boundary of each AMR grid. The ghost zones need to be updated at the end of every substep, this is done by the `Boundary::BoundaryConditions` function.

5. Once the outer boundaries are calculated and updated the B field can be calculated from A_i and $\sqrt{\gamma}\phi$.

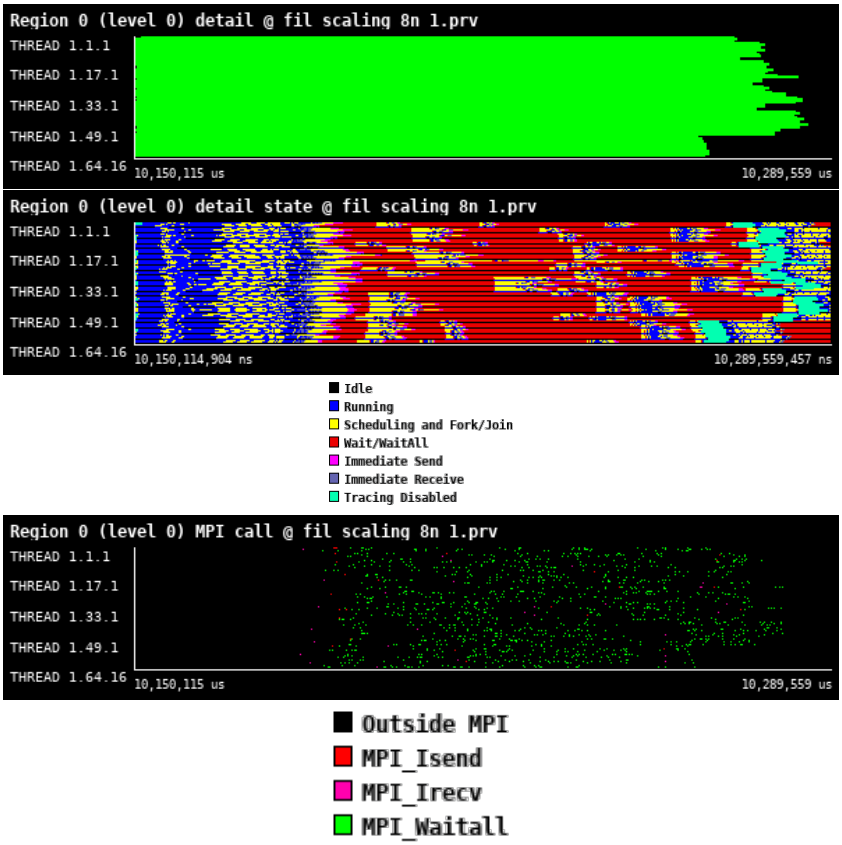
6. The conservative variable $[\rho_*, \tilde{\tau}, \tilde{S}_i, \tilde{B}^i]$ have all been calculated at this point. The primitive variables need to be recovered for the next substep to take place and so the `IllinoisGRMHD_conserv_to_prims` function is used to invert equation (23).

7. Finally zero outflow boundary conditions need to be applied to the primitive variables $[P, \rho_0, v^i]$, this is done by the `IllinoisGRMHD_outer_boundaries_on_P_rho_b_vx_vy_vz` function.

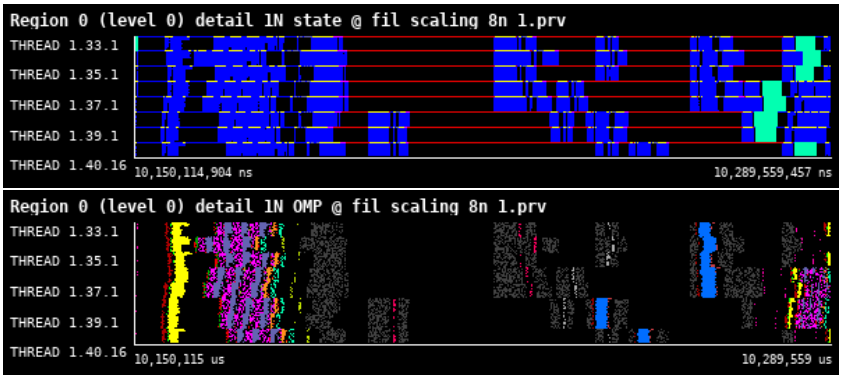
This algorithm is then applied recursively to every refinement level with special care taken to interpolate over the boundaries of the inner AMR grids.

8.4 ScheduleTraverse - Region 0

The main time loop of the code is called from the Carpet Thorn, specifically in the Evolve.cc file by the `ScheduleTraverse` function. The whole of the time loop outlined in the pseudo code of Section 8.2 is profiled, results can be seen in Figure 70. Unlike regions 2 and 3 the profiling of region 1 includes significant part of the ET. Comparisons of how FIL scales compared to the rest of the ET is of particular interest as it needs t odetermined if continuing to use the ET evolution framework is the best route forward for FIL. Currently the ET developers are developing CarpetX: the next generation of Carpet that should be GPU compatible. However current progress has not shown the speed ups one would expect. Other evolution frameworks that are compatible with GPUs are being developed concurrently and the decision to continue using the ET as FIL's evolution framework is undecided.

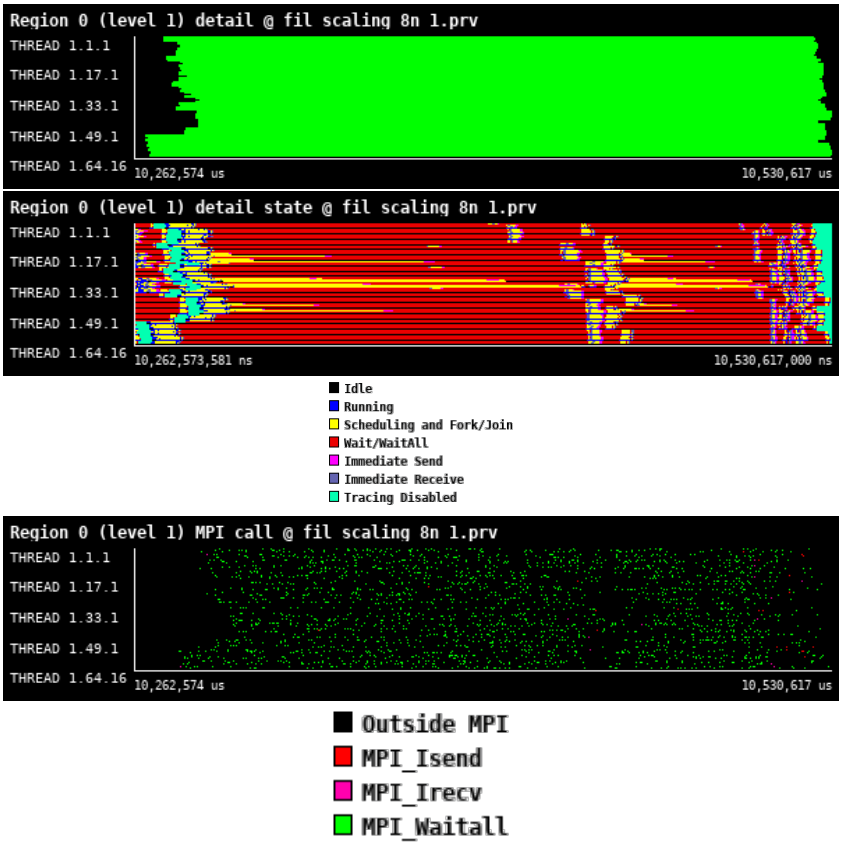


(a) Zoomed traces for Region 0 - level 0 showing MPI structure of the region.

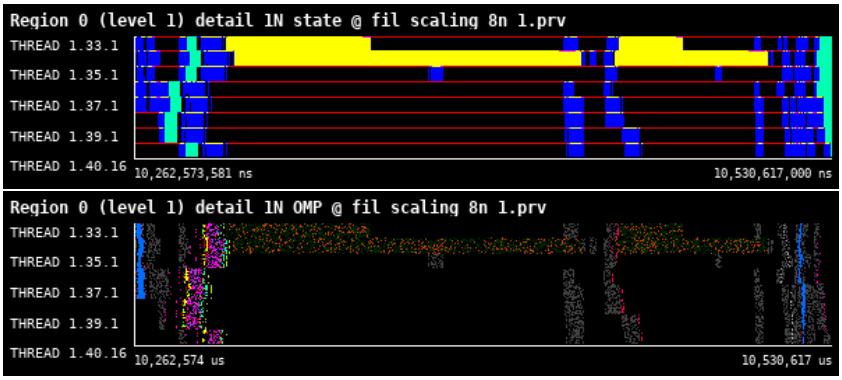


(b) Zoomed traces for selected threads of Region 0 - level 0 howing OpenMP structure of the region.

Figure 70: Zoomed traces for Region 0 - level 0 region of the FIL from Paraver showing the structure of the region.



(a) Zoomed traces for Region 0 - level 1 showing MPI structure of the region.



(b) Zoomed traces for selected threads of Region 0 - level 1 howing OpenMP structure of the region.

Figure 71: Zoomed traces for Region 0 - level 1 region of the FIL from Paraver showing the structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	0.134374	0.098542	0.185117	0.111446
Efficiency	1.0	0.681811	0.181472	0.150717
Speedup	1.0	1.363622	0.725887	1.205732
Average IPC	1.561925	1.465750	1.290249	1.149880
Average frequency (GHz)	2.848977	2.721164	2.497150	2.406346

Table 28: Overview of the key performance metrics of Region 0 - level 0 of FIL.

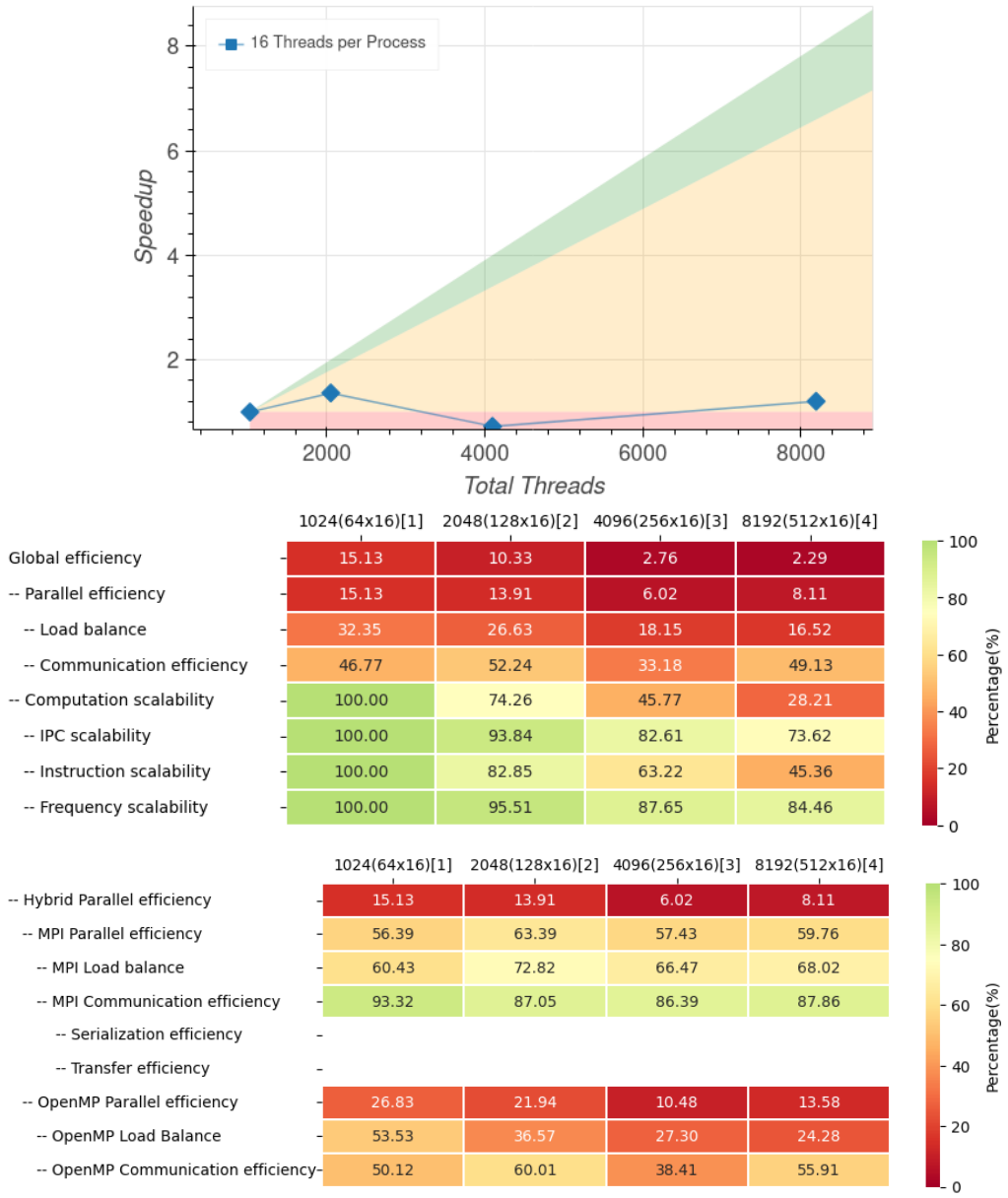


Figure 72: Strong scaling and POP efficiency metrics for Region 0 - level 0 of FIL.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	0.262365	0.25944	0.301456	0.201831
Efficiency	1.0	0.505637	0.217582	0.162491
Speedup	1.0	1.011274	0.870326	1.299924
Average IPC	1.214093	1.084490	0.956516	0.861502
Average frequency (GHz)	2.696634	2.565550	2.365347	2.352145

Table 29: Overview of the key performance metrics of Region 0 - level 1 of FIL.

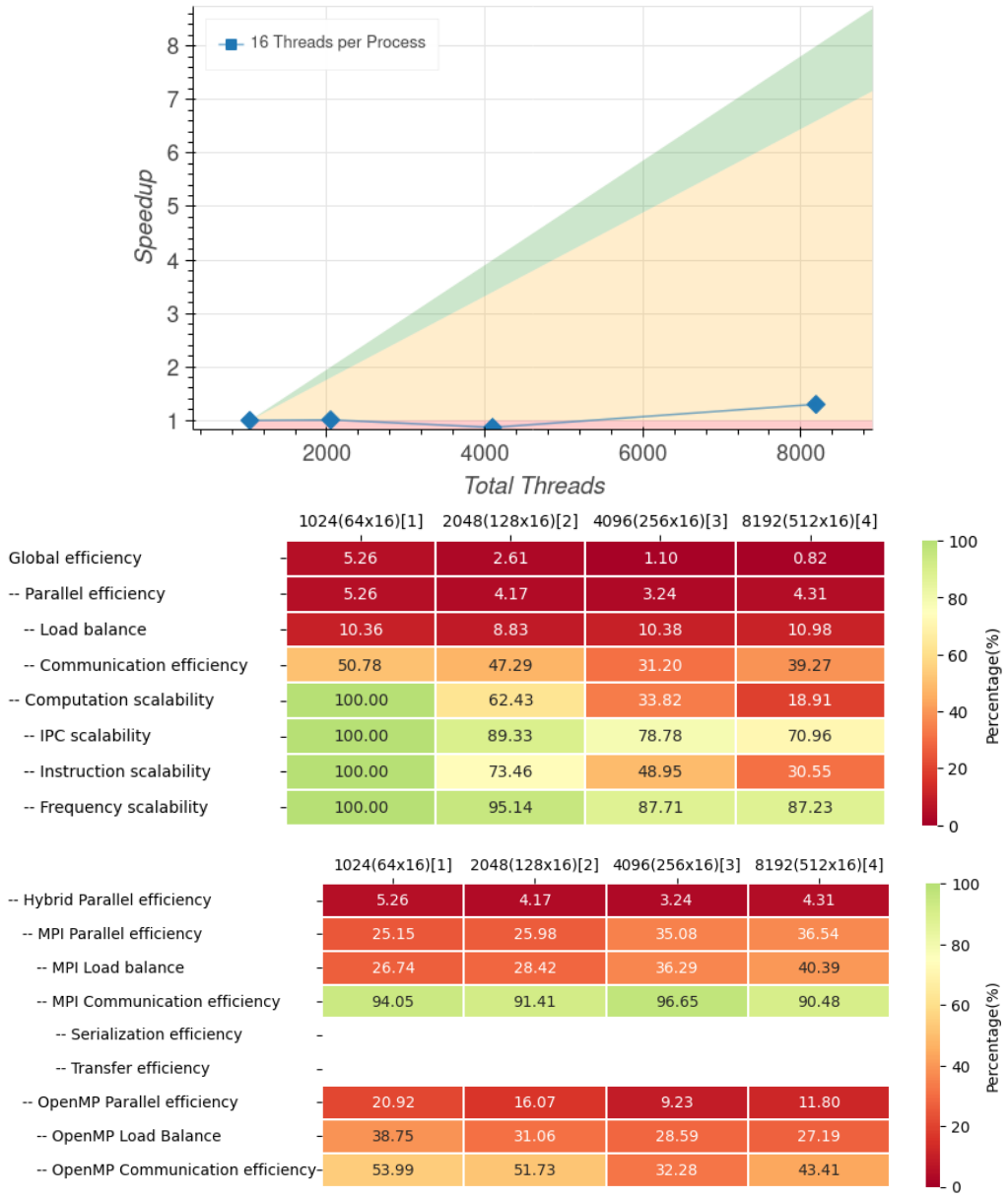


Figure 73: Strong scaling and POP efficiency metrics for Region 0 - level 1 of FIL.

The overall performance of this particular region is very poor. According to data of different refinement levels of the region in Table 28 and Table 29, adding more resources to the simulation affects overall runtime minimaly. Poor scaling is especially visible in Figures 72 and 73 where speedup essentially hewers around value of 1.3. Figures 70 and 71 depicts traces of the integration step into refinement levels 0 and 1, respectively. Each figure consists of two sub-plots: (a) aimed at assessing system-wide performance, and (b) focused on several threads to show OpenMP behaviour.

The Figure 70 clearly shows that most of the time in this level of the region is spent in `MPI_Wait` or `MPI_WaitAll` operations. This hints an issue with load balance as supported by POP metrics presented in Figure 72. Moreover, it can be seen that in the region, there is a relatively large amount of MPI communication calls. This can be potentially mitigated by message aggregation optimization. With respect to OpenMP parallelization of the first level, this region exhibits no apparent issues. One possible cause of poor OpenMP metrics in Figure 72 can be the relatively large sequential portion of the runtime at the beginning. Looking at the Figure 70 (b) there is an evident gap (represented by black color) in between to distinct OpenMP regions depicted in yellow and pink/gray color at the beginning. This gap is precisely the sequential part of the algorithm where only the

master thread is active.

Assessing next refinement level, depicted in Figure 71, we can conclude that the same problems with load balance and excessive MPI communications are found here as well. These problems are more pronounced in this refinement level. On top of that, there is an additional issue with OpenMP parallelization. In section (b) of Figure 71, there is significant portion of the time spent into Scheduling and Fork/Join operations (depicted in yellow). This means that there is large number of small OpenMP parallel regions or work-sharing constructs called repeatedly. This results in the worsening of OpenMP metrics in Figure 73 compared to previous level. The problem is twofold. Firstly, it is evident that this is happening only on several processes which points at load balance issue. Secondly, there is significant overhead in repeated execution of small parallel or work-sharing regions. We suggest to optimize the computation by both improving the load balance and investigating the aggregation of several OpenMP regions to reduce the overhead.

8.5 Driver evaluate MHD RHS - Region 1

The `Illinois_GRMHD_driver_evaluate_MHD_RHS_type` function is called at each sub step of the RK4 method. The driver is responsible for evaluating the flux (MHD) and the source (RHS) in equation (21). To this end, the driver performs successive calls to several functions. The most time consuming functions are:

- `compute_tau_rhs_extrinsic_curvature_terms_and_TUPmunu`, which calculates the extrinsic curvature used to evaluate the source term;
- `reconstruct_set_of_prims_WENO5`, which reconstructs the primitive variables at the cell faces to calculate the primitive variables vector flux in each direction;
- and `add_fluxes_and_source_terms_to_hydro_rhss`, which takes the calculations from the other two functions and evaluates the flux and source terms.

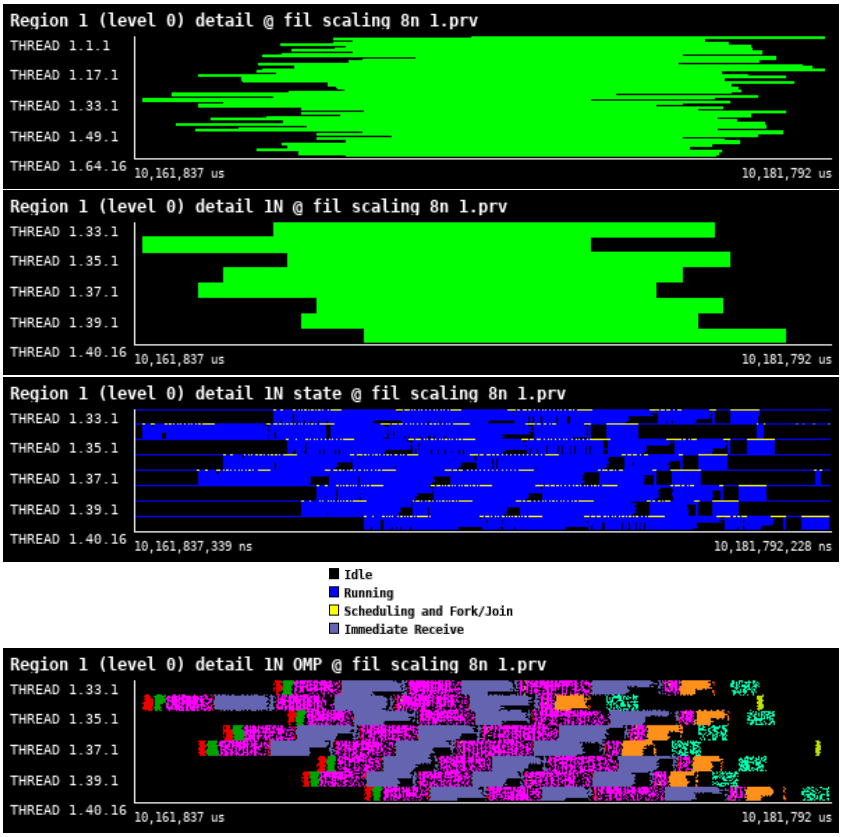


Figure 74: Zoomed traces for selected threads of Region 1 - level 0 of the FIL from Paraver showing OpenMP structure of the region.

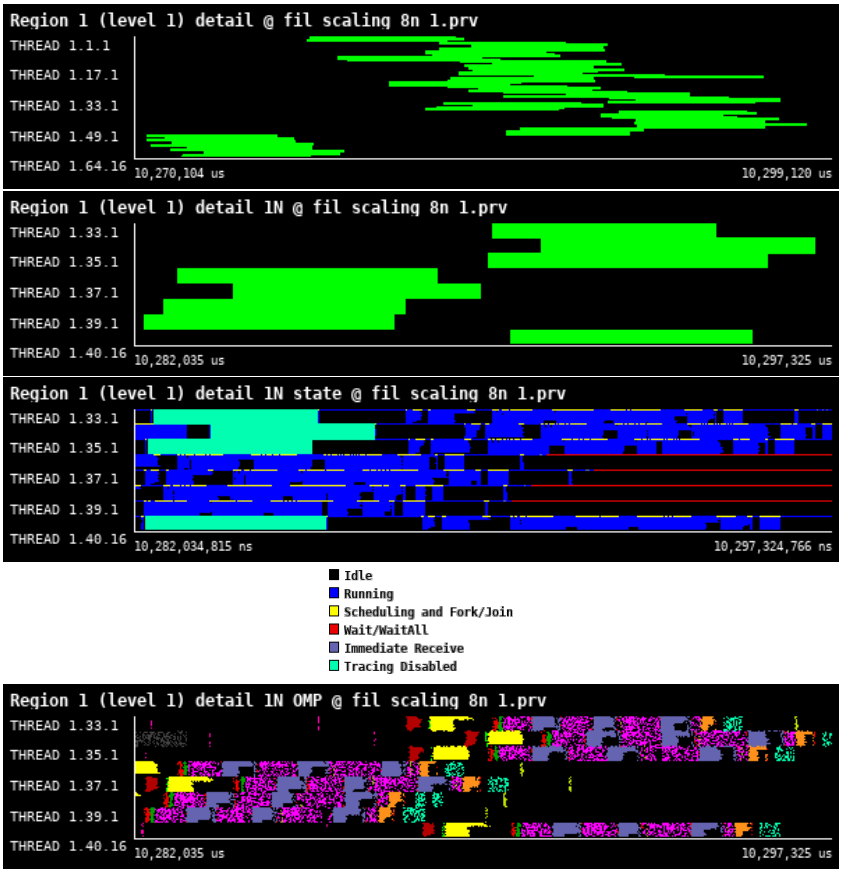


Figure 75: Zoomed traces for selected threads of Region 1 - level 1 of the FIL from Paraver showing OpenMP structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	0.016069	0.031484	0.017035	0.010838
Efficiency	1.0	0.255193	0.235823	0.185332
Speedup	1.0	0.510386	0.943293	1.482654
Average IPC	2.138470	2.111914	2.021546	1.984923
Average frequency (GHz)	3.039873	3.034593	2.883009	2.894161

Table 30: Overview of the key performance metrics of Region 1 - level 0 of FIL.

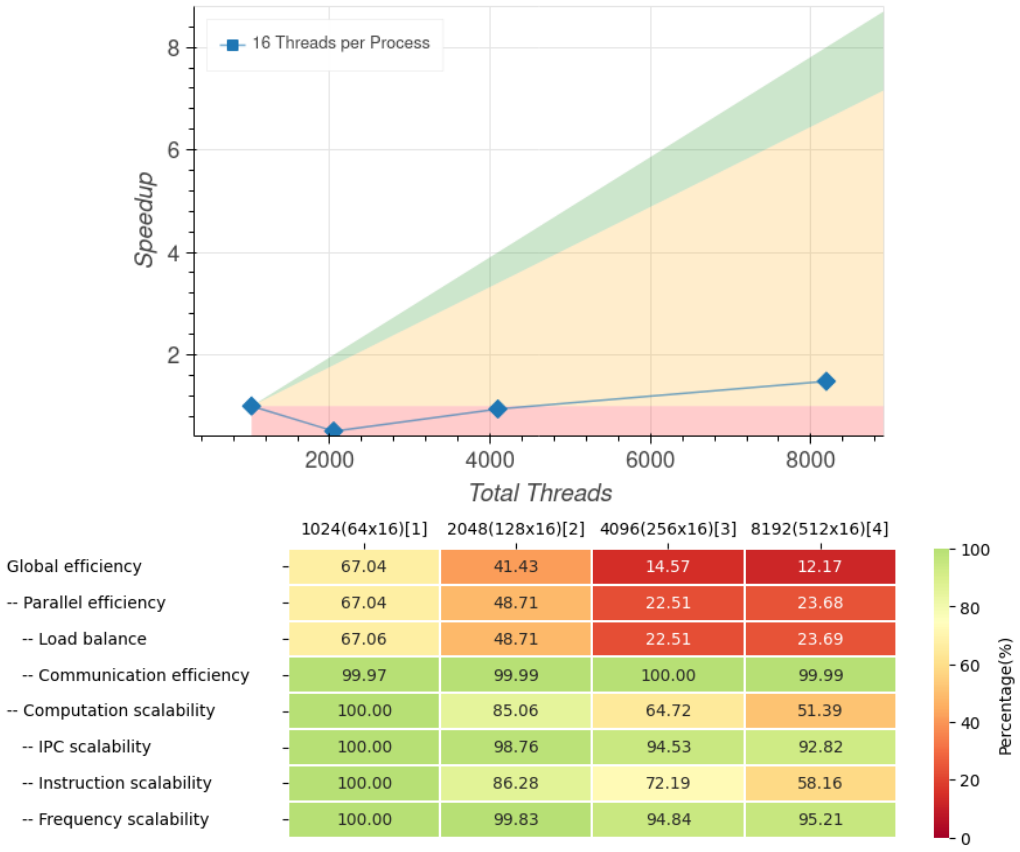


Figure 76: Strong scaling and POP efficiency metrics for Region 1 - level 0 of FIL.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	0.006314	0.004637	0.007423	0.005496
Efficiency	1.0	0.680828	0.21265	0.143604
Speedup	1.0	1.361656	0.850599	1.148836
Average IPC	2.031888	1.982323	1.860381	1.766386
Average frequency (GHz)	3.101163	3.047925	2.906245	2.875670

Table 31: Overview of the key performance metrics of Region 1 - level 1 of FIL. For purely computational kernel (no MPI communication), Parallel efficiency metrics are identical to OpenMP efficiency, therefore, hybrid metrics are omitted.

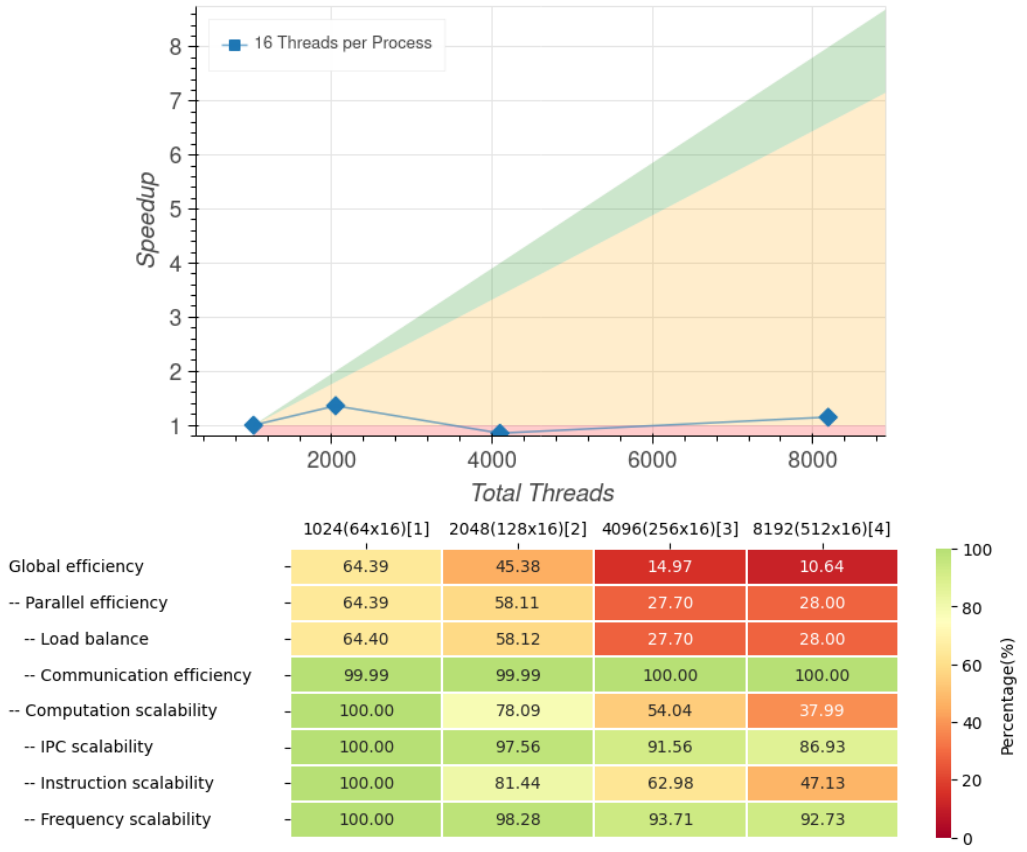


Figure 77: Strong scaling and POP efficiency metrics for Region 1 - level 1 of FIL. For purely computational kernel (no MPI communication), Parallel efficiency metrics are identical to OpenMP efficiency, therefore, hybrid metrics are omitted.

Figures 74 and 75 depicts the behaviour of two different representative regions belonging to different refinement levels. As these regions are compute kernels only, we focus here on the OpenMP parallelization. The regions starts at different times on different processes. This is caused by imbalances in previous computation. From a system-wide performance point of view, there is significant issue with load balance which prohibits the scaling. The metrics as well as scaling plots can be seen in Figures 76 and 77. Again, maximum speedup is, according to Tables 30 and 31, only around 1.5 and 1.3 respectively. Because the load-balancing is done inside the Cactus framework rather than into the FIL code it self, we suggest to investigate the possibilities to improve this via internal settings.

Overall, the performance of OpenMP is good. There are only minor issues. Firstly, there is (again) a relatively large number of Scheduling and Fork/Join operations (indicated by yellow color in middle plot) and as such this kernel could benefit from OpenMP region aggregation. Also, there is a slight OpenMP imbalance, visible predominantly in gray regions in last sub-plots. Relatively small sequential-only part of the algorithm (section between orange and mint regions) can be seen at the end of the region. The expected impact on performance should be minimal. This is a purely computational kernel, which makes a good candidate for SIMD vectorization. Due to potentially complex control flow, the possibility of GPU acceleration should be investigated.

8.6 Conservative to primitive solver - Region 2

After the conservative variables and the boundary conditions have been updated the primitive must be calculated by inverting (23). This is not a trivial task as the relationship between the conservative variables and the primitives is non-linear, requiring the use of a root-finding algorithm. For FIL, a 2D Newton-Raphson solver is used, implemented in the `IllinoisGRMHD_conserv_to_prims` function. Because of truncation and interpolation errors, calling the 2D Newton-Raphson solver naively may incur unphysical results. To prevent this, the `IllinoisGRMHD_conserv_to_prims` function does a series of checks to make sure the conservative variables are in a valid range before the root finder is called. If the conservative values are outside of this range they are minimally modified. Sometimes the root finder will still fail to find a root, this is very rare and almost always occurs in a low-density environment, such as the atmosphere of a neutron star or the interior of a black hole. In such cases the pressure is manually set to guarantee inversion. Once the primitives are successfully calculated, they are checked for physicality (make sure fluid speed remain sub-liminal for example) and the evolution algorithm can continue.

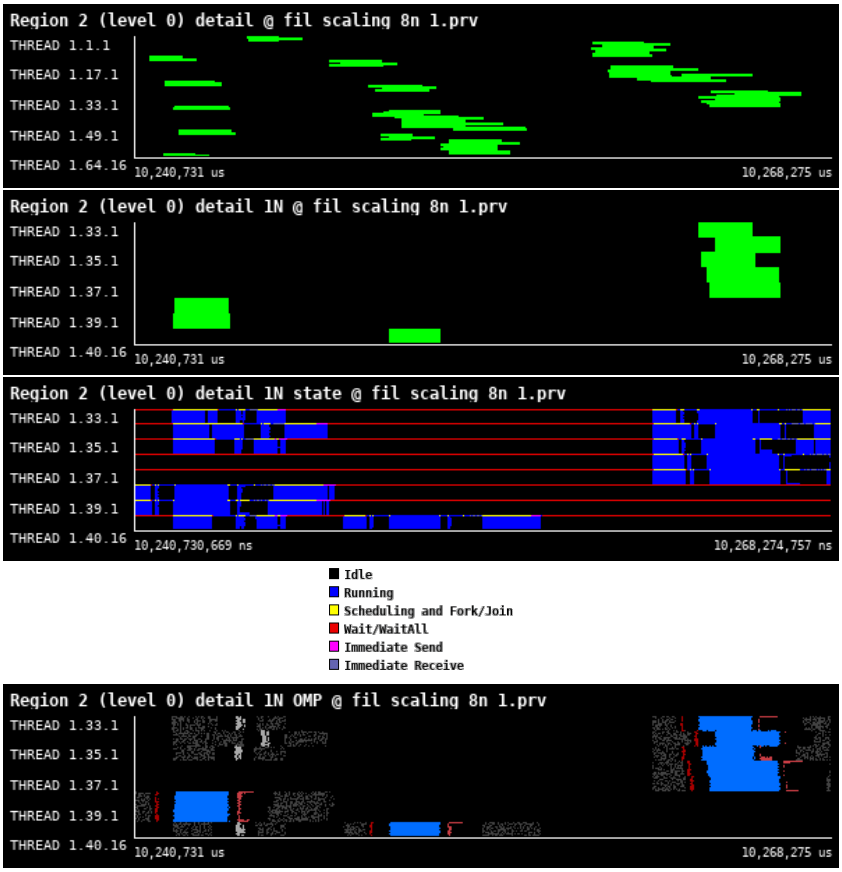


Figure 78: Zoomed traces for selected threads of Region 2 - level 0 of the FIL from Paraver showing OpenMP structure of the region.

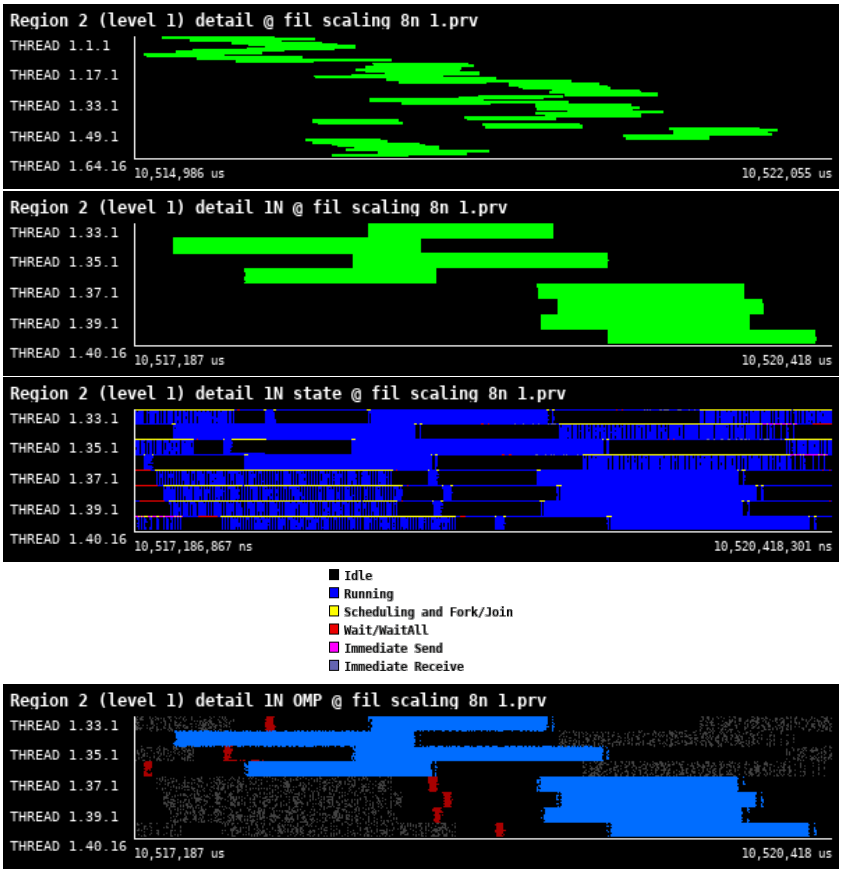


Figure 79: Zoomed traces for selected threads of Region 2 - level 1 of the FIL from Paraver showing OpenMP structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	0.002936	0.00178	0.00319	0.00226
Efficiency	1.0	0.824719	0.230094	0.162389
Speedup	1.0	1.649438	0.920376	1.299115
Average IPC	1.261826	1.252671	1.172994	1.175525
Average frequency (GHz)	3.172177	3.163214	3.039527	3.099582

Table 32: Overview of the key performance metrics of Region 2 - level 0 of FIL.

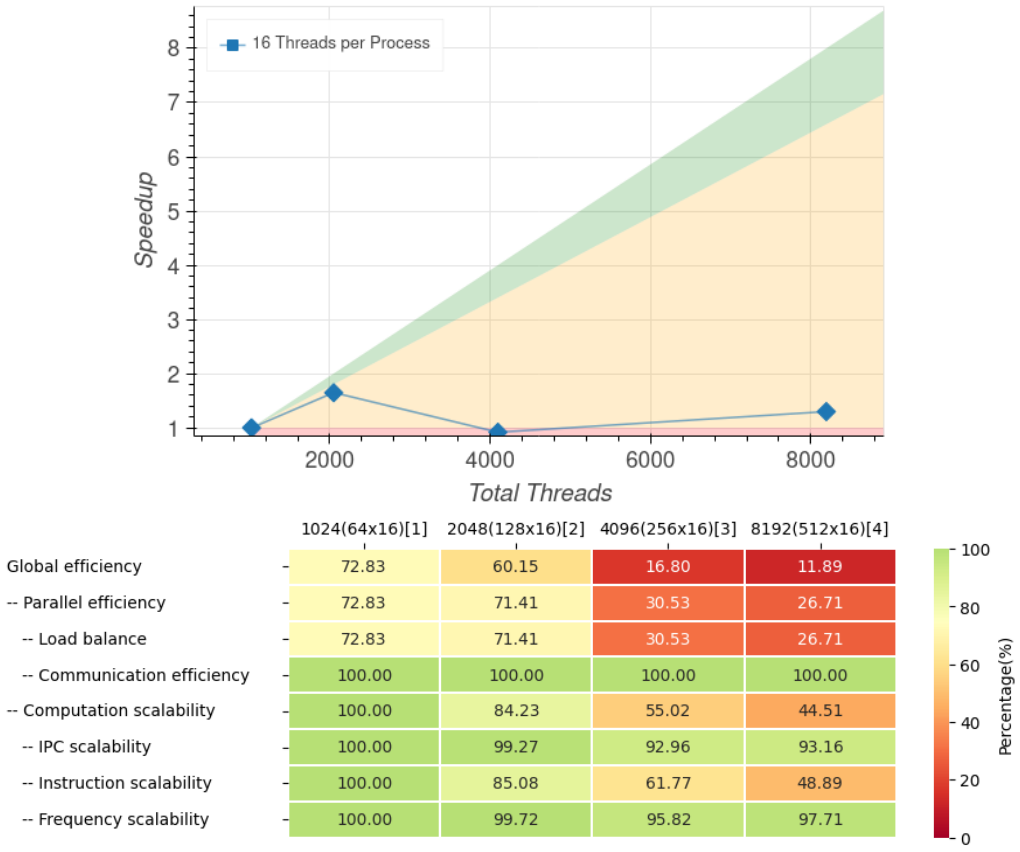


Figure 80: Strong scaling and POP efficiency metrics for Region 2 - level 0 of FIL.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	0.001181	0.000937	0.001614	0.001153
Efficiency	1.0	0.630203	0.18293	0.128035
Speedup	1.0	1.260406	0.731722	1.024284
Average IPC	1.266156	1.153518	1.142556	1.189507
Average frequency (GHz)	3.195190	3.170568	3.068834	3.101389

Table 33: Overview of the key performance metrics of Region 2 - level 1 of FIL. For purely computational kernel (no MPI communication), Parallel efficiency metrics are identical to OpenMP efficiency, therefore, hybrid metrics are omitted

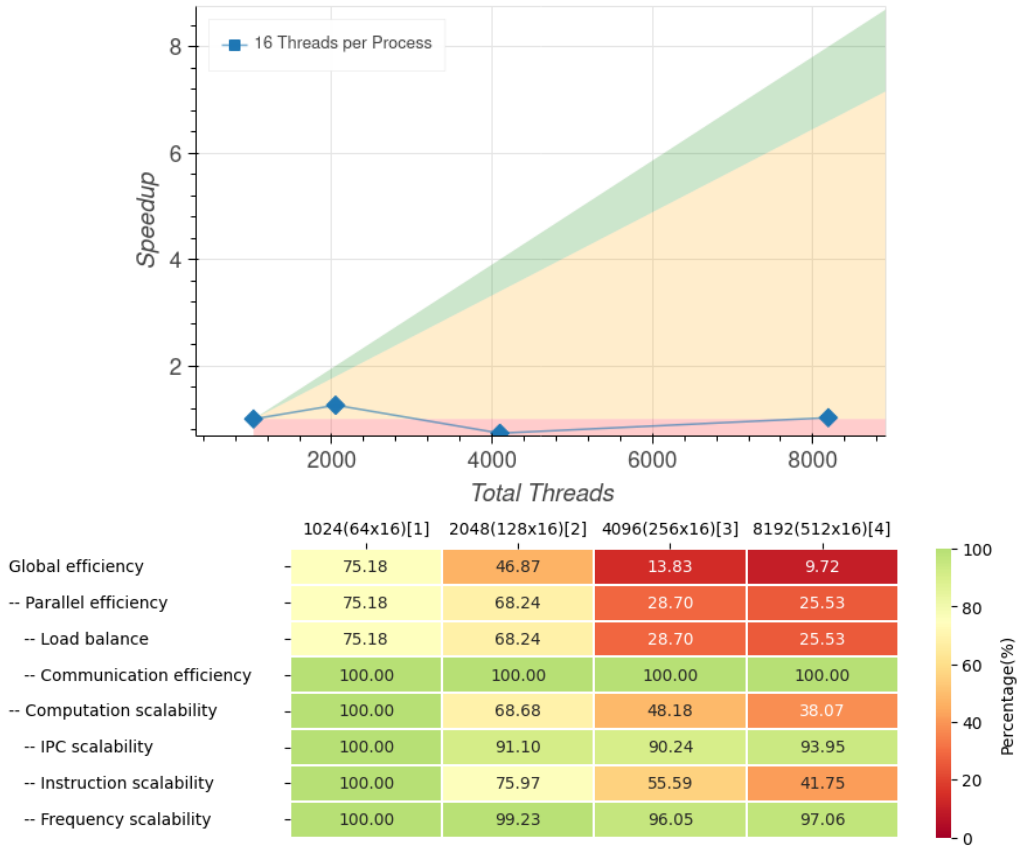


Figure 81: Strong scaling and POP efficiency metrics for Region 2 - level 1 of FIL. For purely computational kernel (no MPI communication), Parallel efficiency metrics are identical to OpenMP efficiency, therefore, hybrid metrics are omitted.

Again, Figures 78 and 79 represents two different refinement level. Our main focus here is the OpenMP parallelization as the region is purely computational. According to timeline, the performance of OpenMP is close to ideal. The region consists of a single OpenMP region with relatively good load balance. However, as scaling and POP metrics in Figures 80 and 81 show, the performance degrades with the growth of the number of total threads. The metric which have the most significant impact on overall performance is load balance. This is most probably caused by inefficiencies in the Cactus framework rather than implied by the implementation of the FIL physics module. As this region is purely computational, it is a potential candidate for GPU acceleration as well as SIMD vectorization. Due to comparatively short runtime visible in Tables 32 and 33, the impact on overall performance might be minimal. We suggest further investigation into whether it would be profitable to accelerate this kernel.

This document is Public (PU)
and was produced under SPACE project EU GA 101093441

8.7 Conclusions

From the profiling of region 0, it was found that a significant portion of the region run time was taken up by Wait/Wait all operations, suggesting load balancing problems. The issue is more pronounced at the second refinement level. With physics runs of FIL often having 6 refinement levels, if this trend continues, it could be a serious unexplored cause of overhead. More refinement levels could not be traced due to issues with Extrae trace file size. However, future work could be focused on seeing if this behaviour continues.

The load balancing issue in region 0 have knock on effects in region 1 where the driver is called at different times. Further work needs to be done to assess what parts of the ET are causing the load balancing issues. An assessment of whether these issues will be fixed by message aggregation optimization, future versions of the ET, creation of specialist Thorns to optimise the ET for FIL or using a different evolution scheme entirely needs to be conducted.

For regions 1 and 2, few load balancing issues were found with minor optimisations of region 1 being suggested with the caveat that overall affect on performance would be minimal. Both regions were suggested for SIMD vectorization with the region 1 being the main focus, due to the short run time and therefor minimal performance gains vectorizing region 2 would bring.

A qualifying note has to be made here however: the conservative to primitive function takes longer to run in highly magnetized regions before collision. Future profiling endeavours may benefit from starting the simulation with the 2 neutron star closer together to profile the code in this more extreme environment.

9 ChaNGa

ChaNGa [28][29][4] is an N-body and smoothed particle magneto-hydrodynamics (SPMHD) code which is used to study a wide array of astrophysical systems. While the gravity and SPMHD algorithms are based on the gasoline [30] and pkdgrav [31] codes, the unique feature of ChaNGa is it's implementation of the Charm++ framework, which enables highly efficient parallel scaling. Charm++ employs overdecomposition to achieve this. That is to divide the work into many more pieces (chares/tree pieces) than you have processors and let the Charm++ runtime system load balance by appropriately assigning pieces to real processors (see Figure 82 for an overview). During runtime, Charm++ applies dynamic load rebalancing strategies, to determine which tree pieces should be migrated to new processors for better load balance. In addition, the SMP mode of Charm++ leverages the shared-memory systems found in high-performance computing environments. Within an SMP process/node, one thread is assigned as the communication thread, responsible for internode communication, while the remaining threads, act as worker threads in charge of the processing elements. Multiple SMP processes can be initiated per network node, the number of which should be based on the CPU architecture and communication load. Charm++ also provides support to execute CUDA kernels on the GPU asynchronously and to manage data transfers between the CPU and GPU.

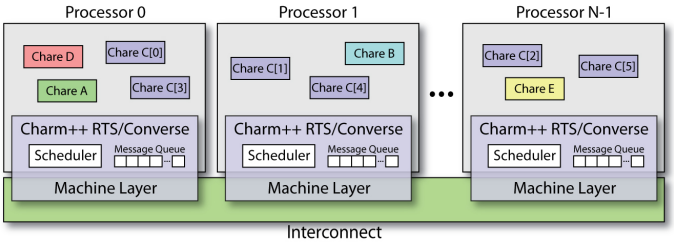


Figure 82: System's View of a Charm++ Application

The central feature of ChaNGa is a tree-based gravity solver using a variant of the Barnes-Hut algorithm. This solver is combined with several other features which include: gravitational interaction between dark matter, stars and gas, Ewald summation to handle cosmological boundary conditions, multi-time stepping, hydrodynamics, magnetic field and several subgrid physics such as, star formation, feedback from supernova, stellar winds, black holes and the radiative cooling of gas. We can summarize these into 4 major parts: gravity, Ewald/boundary, SPMHD and subgrid physics. We begin by looking at gravity. To compute the gravitational force at any point in space given a specific mass distribution, we want to solve Poisson's equation for gravity:

$$\nabla^2\Phi = 4\pi G \int f dv = 4\pi G\rho \tag{24}$$

Here ∇^2 is the Laplace operator, Φ is the gravitational potential, G is the gravitational constant, ρ is the mass density at each point in space. In an N-body code, the continuous mass distribution is discretized into a finite number of particles. Instead of directly calculating the gravitational influence that each particle has on eachother, the Barnes-Hut tree approximation is used, where distant particles are combined into center of mass of tree cells. This allows one to separate the force calculation into subdivisions of long-range components (tree/multipole) and a short-range component (direct) to the degree which is required by the desired force accuracy. For ChaNGa the global tree is then split into N number of tree pieces (depending on the given overdecomposition) and distributed among all the processors (including different nodes). To achieve effective load balancing, it's recommended that each processor handles a minimum of 8 tree pieces, with an optimal particle count ranging between 500 and 1000 within each tree piece. Particles, or more specifically, groups of particles contained in the tree leaves (often referred to as "buckets"), traverse the tree to compute the gravitational force. A representation of the tree structure can be seen in Figure 83.

The governing equations of smoothed particle magneto-hydrodynamics is given by:

$$\frac{dv_a^i}{dt} = \sum_b \frac{m_b}{\rho_a \rho_b}\left(S_a^{ij} + S_b^{ij}\right)\nabla_a^j \overline{W_{ab}} + F_{v,diss}, \tag{25}$$

$$\frac{du_a}{dt} = \frac{P_a}{\rho_a}\sum_b \frac{m_b}{\rho_b}\left(v_b - v_a\right)\cdot \nabla_a \overline{W_{ab}} + F_{u,diss}, \tag{26}$$

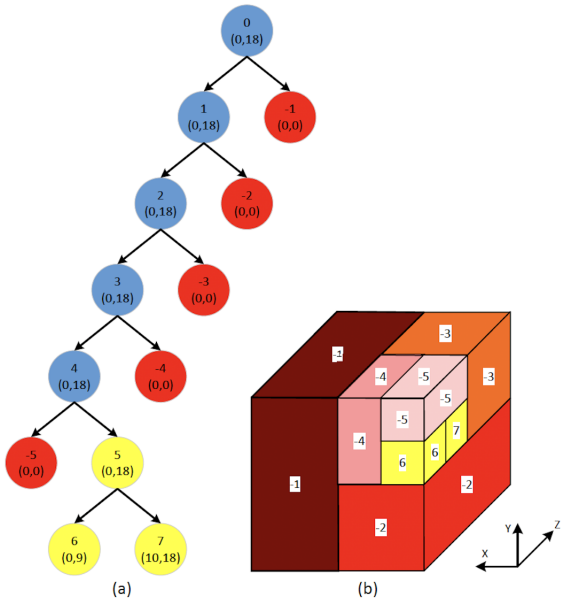


Figure 83: a) Tree structure, here we can see nodes within local tree piece (yellow and blue) and nodes within remote tree pieces(red) b) Spatial representation of the tree build(the blue nodes have been represented as their remote children nodes). Darker color indicates a further away node from the local nodes.

$$\frac{dB_a}{dt} = \sum_b \frac{m_b}{\rho_b} \left[B_a(v_{ab} \cdot \nabla_a \overline{W_{ab}}) - v_{ab}(B_a \cdot \nabla_a \overline{W_{ab}}) \right] + F_{B,diss} + F_{B,clean} \tag{27}$$

Here a and b are particle indices, v is the velocity, m is the mass, ρ is the density, S is the stress tensor($S = -\delta^{ij}\left(P + \frac{B^2}{2}\right) + B^i B^j$), P is the pressure, $F_{v,diss}$ is the momentum term of artificial viscosity, $F_{u,diss}$ is the heat dissipation term of artificial viscosity, resistivity and thermal diffusion, $F_{B,diss}$ is the magnetic resistivity and $F_{B,clean}$ is the divergence cleaning function. Where the density (ρ) is given by the distribution of particles (interpolation points) and the smoothing kernel:

$$\rho_a = \sum_b^{N_{smooth}} m_b W_{ab} \tag{28}$$

The term N_{smooth} represents the number of neighbors that each particle uses for interpolation and is user-defined (default 64). Timestepping in ChaNGa is done using the "Kick Drift Kick" integrator, using individual timesteps for each particle. An arbitrary number of sub-steps with factors of two smaller can be used to integrate gas with different timestep criteria.

9.1 Use-case description

For our performance analysis we look at a simulation of a major merger between two Milky-Way type galaxies. This simulation include a plethora of physics(gravity, hydrodynamics, magnetic fields, star formation, feedback, radiative cooling) and presents close to the toughest conditions for our code, in terms of parallelisation. The regions that we have decided to focus on, pertains to the gravity and hydrodynamics calculations, which are the two most expensive computations. They also present different challenges: gravity is a long-range force that needs to calculate the contribution from all particles (or regions as explained in previous section), while hydrodynamics require information only from it's nearest neighbours(usually 64 nearest). The hydrodynamic part is more communication intensive compared to gravity, which on the other hand is more computation intensive. The use-case is dynamically not evolved much in this benchmark test as we focus on the performance profiling of the global timestepping. The state of the merger during the simulation run occur during the approach of the two galaxies, as the circumgalactic-medium of each galaxy start to mix.

```
################ Run and Output Related
```

```
achInFile       =  LOW_64XLARGEMHDVERTDENSWend64FBSB1mergb0.00410
achOutName      = LOW_64XLARGEMHDVERTDENSWend64FBSB1mergb0b
dESN = 1.0e51
dInitGasMass = 1.34268e-06  #64X
iCheckInterval  = 50
iOutInterval    = 50
iLogInterval    = 10
bDoGravity      = 1
bBenchmark = 1
bDohOutput = 1

################Timesteps
dEta            = 0.185
dEtaCourant     = 0.4   #leave this alone

################Tree Options
dTheta          = 0.7
dDelta          = 0.00000670763 # Forcing single timestepping for benchmark 0.00001 Myr
nSteps          = 10


################Performance Options
dFracNoDomainDecomp = 0.1
bConcurrentSph = 1
nDomainDecompose = 1 #OCT-TREE

################Boundary/Cosmology
bComove         = 0
bPeriodic       = 0


################Gas Options
dMsolUnit       = 1e9
dKpcUnit        = 1
dConstGamma     = 1.666667
dMeanMolWeight  = 0.59259
dThermalDiffusionCoeff = 0.03
bDoGas = 1

################SPH
bVDetails       = 1
bViscosityLimiter = 1
bViscosityLimitdt = 1
nSmooth         = 64
nSmoothFeedback = 1
dConstAlpha     = 2
bFastGas        = 1
dhMinOverSoft = 0.25 #2017,  was 0.5

################SF Parameters
bStarForm = 1
bFeedBack       = 1
nSmoothFeedback = 1
dInitStarMass = 1.34268e-05 #10*gasmass --> always used full gas particle
dCStar          = 0.05
dPhysDenMin     = 100
dTempMax        = 1e3
dDeltaStarForm = 1e5

################Gas Physics with Metals/Photo Electric
bGasCooling                             = 1
bUV             = 1

################What Arrays to Write
iBinaryOutput   = 1
bDoIOrderOutput = 1 # writes .iord, .igasorder, .rhoform, .Tform
bDoDensity      = 1

############### Jeans Floor
dResolveJeans = 14.1372

.
```

9.2 High-level code structure

After initialization we begin our integration and start stepping our simulation.

```
doSimulation() //Start of simulation
{
...
for(int iStep = param.iStartStep+1; iStep <= param.nSteps; iStep++){
...
advanceBigStep(iStep-1);
...
}
} //End of simulation
```

We allow for multi-timestepping and follow the "Kick Drift Kick" method as outlined in eq. 29-32.

```
advanceBigStep(int iStep) {
  int currentStep = 0; // the current timestep within the big step
  while (currentStep < MAXSUBSTEPS) {
  ...
     * Form stars
     domainDecomp(PHASE_FEEDBACK);
     loadBalance(PHASE_FEEDBACK);
      if(param.bStarForm)
        FormStars(dTime, param.stfm->dDeltaStarForm);
      if(param.bFeedback)
        StellarFeedback(dTime, param.stfm->dDeltaStarForm);
...
    /***** Resorting of particles and Domain Decomposition *****/
    domainDecomp(activeRung);
        /********* Load balancer ********/
    loadBalance(activeRung);
        /******** Tree Build *******/
    buildTree(activeRung);
...
/******** Start Gravity *******/
    startGravity(cbGravity, activeRung, &gravStartTime);
/******** Start SPH *******/
    doSph(activeRung);
...
/******** Function to re-sync/wait for previous calculations to be done (as these are done concurrently) **/
    waitForGravity(cbGravity, gravStartTime, activeRung);
}
} // END OF 1 BIG STEP
```

We start the projection tracing within `startGravity()`. Which launches the tree walk and gravity calculation on each of the tree pieces.

```
startGravity(const CkCallback& cbGravity, int iActiveRung,
                        double *startTime){
...
#ifdef SELECTIVE_TRACING
        turnProjectionsOn(iActiveRung);
#endif
...
/// This method starts the tree walk and gravity calculation.
treeProxy.startGravity(iActiveRung, theta, cbGravity);
...
}
```

The `treeProxy.startGravity()` first registers with the node and particle caches. It initializes the particle acceleration by calling `initBucket()`. It initializes treewalk bookkeeping, and starts a prefetch walk which will eventually call a remote walk (a walk on non-local nodes). Finally it starts the local gravity walk.

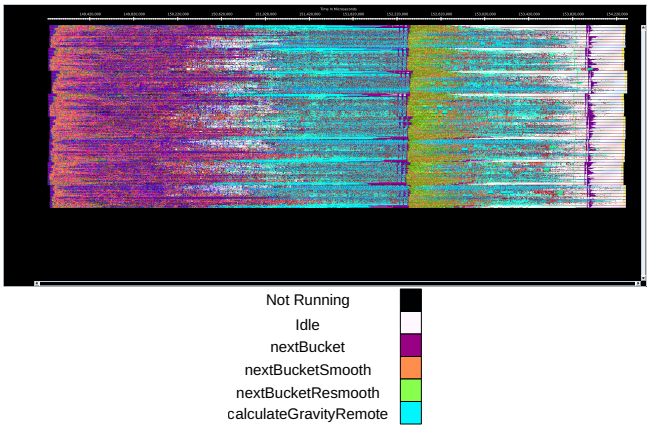
```
TreePiece::startGravity(int am,double myTheta,const CkCallback& cb) {
...
/// This calls the continueStartRemoteChunk(sPrefetchState->currentBucket) which
/// starts CalculateGravityRemote when prefetching is done.
/// Prefetches are done in chunks so that remote computation can start before all remote prefetches finishes.
initiatePrefetch(sPrefetchState->currentBucket);
...
/// This calls doAllBuckets() which then calls
/// nextBucket() and represents the local gravity calculation region.
thisProxy[thisIndex].commenceCalculateGravityLocal();
...
}
```

For the SPMHD part we have:

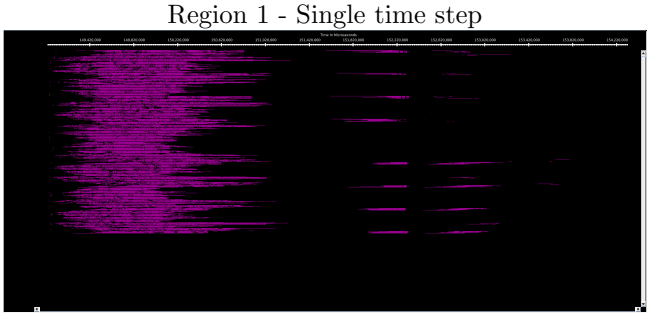
```
doSph(int activeRung, int bNeedDensity) {
...
  if(bNeedDensity) {
  ...
/// Neighbour finding + density calculation, executes KNearestSmoothCompute() and then
/// calculateSmoothLocal() which execute nextBucketSmooth()
treeProxy.startSmooth(&pDen, 1, param.nSmooth, dfBall2OverSoft2, CkCallbackResumeThread());
...
  }
  ...
  treeProxy.getCoolingGasPressure() //Calculate pressure and other non-gradient terms
  ...
  /// Calculate SPH gradients, this calls calculateReSmoothLocal() which executes nextBucketReSmooth()
  treeProxy.startReSmooth(&pPressure, CkCallbackResumeThread());
  ...
}
```

Both gravity and SPH is run concurrently. The tracing stops during the `waitForGravity()` call:

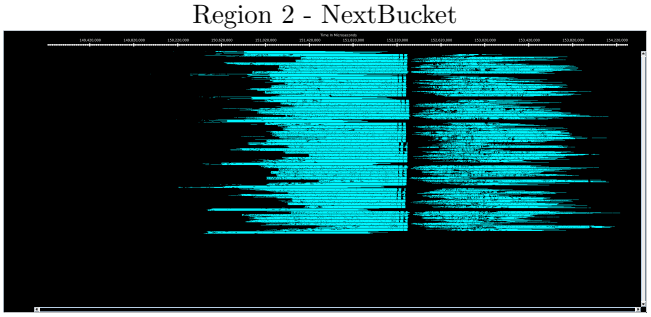
```
waitForGravity(const CkCallback &cb, double startTime,
                                int activeRung)
{...
CkFreeMsg(cb.thread_delay()); // Wait for gravity+sph to finish, which is running concurrently
...
#ifdef SELECTIVE_TRACING
        turnProjectionsOff();
#endif
}
```



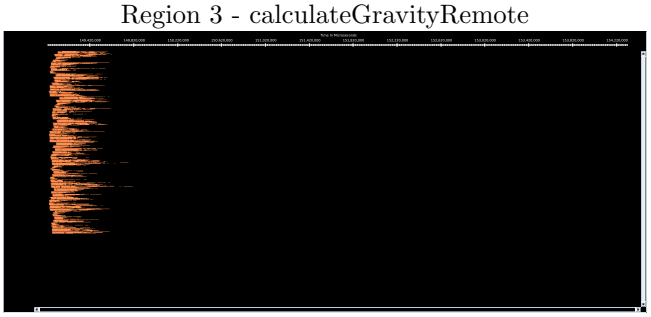
Region 1 - Single time step



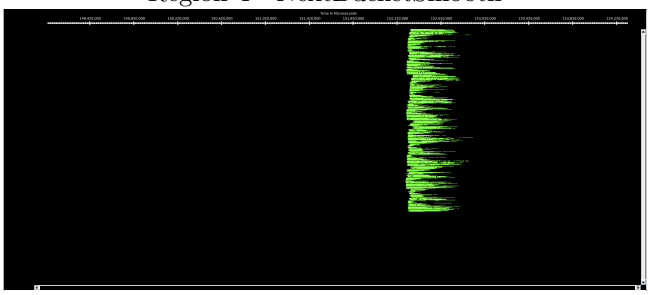
Region 2 - NextBucket



Region 3 - calculateGravityRemote



Region 4 - NextBucketSmooth



Region 5 - NextBucketResmooth

Figure 84: Traces for all regions in ChaNGa from Projections showing the high-level structure of the code.

9.3 Single time step structure

`advanceBigStep()` advances the equations according to the "Kick Drift Kick" integration method:

$$v^{n+\frac{1}{2}} = v^n + \frac{1}{2}\nabla t a^n \tag{29}$$

$$r^{n+1} = r^n + \nabla t v^{n+\frac{1}{2}} \tag{30}$$

$$a^{n+1} = a(r^{n+1}) \tag{31}$$

$$v^{n+1} = v^{n+\frac{1}{2}} + \frac{1}{2}\nabla t a^{n+1} \tag{32}$$

Here v is the velocity, r is the position, a is the acceleration, ∇t is the size of the timestep and n is the nth timestep. For this use-case we take 10 global timesteps and record the traces of the 9th timestep.

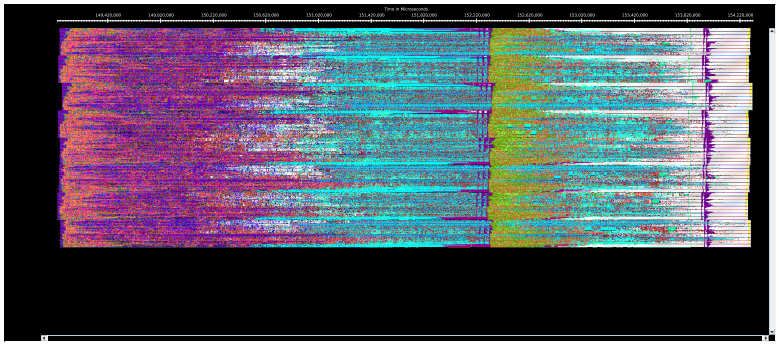
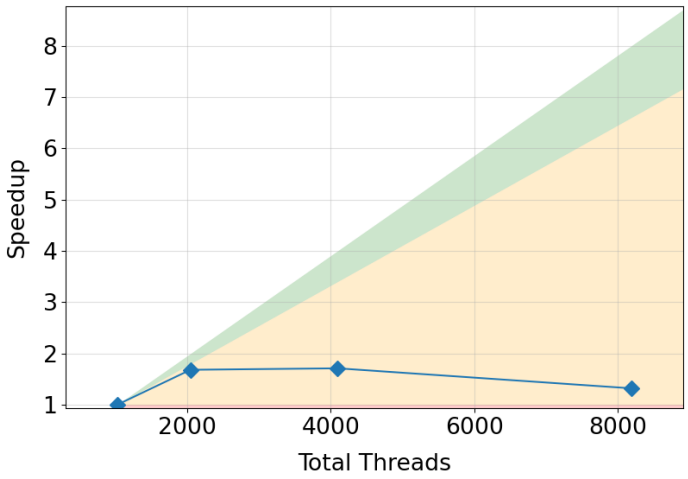


Figure 85: Zoomed traces for Region 1 of the ChaNGa from Projections showing the structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	5.278756	3.136457	3.083357	3.994412
Efficiency	1.0	0.841516	0.428004	0.165192
Speedup	1.0	1.683032	1.712016	1.321535

Table 34: Overview of the key performance metrics of Region 1 of ChaNGa.



Number of Processes	1024	2048	4096	8192
Global Efficiency	0.74	0.62	0.32	0.12
↪ Parallel Efficiency	0.74	0.63	0.33	0.15
↪ Load balance	0.85	0.81	0.72	0.75
↪ Communication Efficiency	0.87	0.78	0.46	0.20
↪ Computation Scaling	1.00	0.98	0.96	0.82

Figure 86: Strong scaling and POP efficiency metrics for Region 1 of ChaNGa.

The asynchronous nature of the Charm++ is clearly visible in Figure 84 as well as Figure 85, where different investigated functions do not form well-defined regions. This single step region contains all the computation and communication within the selected time step. It also includes the kernel regions 2 to 5. The time step region exhibits very poor scaling. As Figure 86 suggest, this is mostly due to inefficiency in communication scaling together with deteriorating load balance. As a result the runtime for 8192 processes is longer than runtime for 2048 or 4096 (see Table 34). We suggest to explore possibilities to optimize the communication in Charm++ or improving the cache manager of ChaNGa.

9.4 Gravity (local tree-traversal) - Region 2

As described in the introduction, the tree is traversed to calculate the gravitational force for all the particles. The `nextBucket()` function represents the local-tree traversal calculations, where particles in a tree piece interact with other particles in the same tree piece. However, rather than each particle performing its own tree walk, particles are grouped into buckets to traverse the tree. The local walk is done at the same time as the remote walk is waiting for the response from distant nodes. Enabling overlap between the communication and computation.

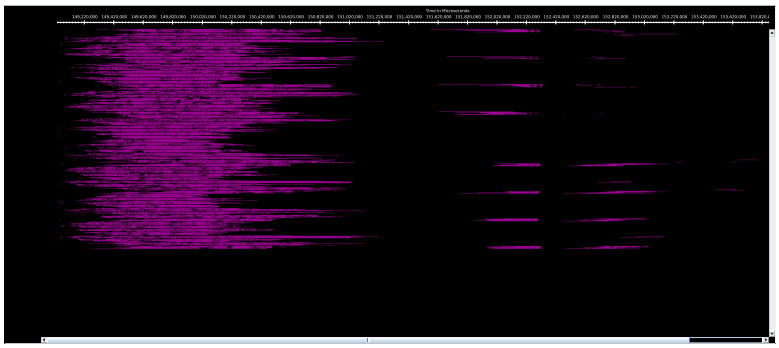


Figure 87: Zoomed traces for Region 2 of the ChaNGa from Projections showing the structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	2.059258	1.012433	0.454667	0.239186
Efficiency	1.0	1.016985	1.132289	1.076180
Speedup	1.0	2.033969	4.529156	8.609442

Table 35: Overview of the key performance metrics of Region 2 of ChaNGa.

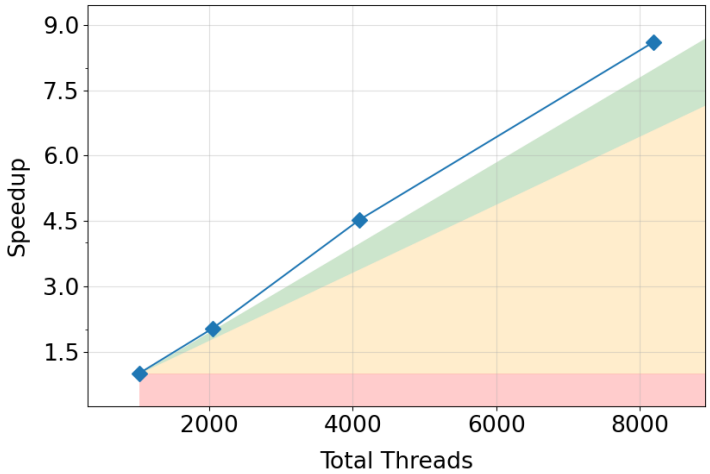


Figure 88: Strong scaling for Region 2 of ChaNGa.

Figure 87 show that work in this region is mostly concentrated at the beginning. However we can see same additional calls to this function later on. The distribution of the workload seems to be uneven, with some processing elements having significantly more work later in the timeline and some having only initial portion of work. We suggest to investigate reason for additional calls to this function after initial work. The super-linear scaling, visible in Figure 88 and Table 35 is probably caused by better data locality as a result of smaller data size per processing element. As the metrics are calculated from aggregated runtime of this function and

therefore doesn't include idle times caused by communication in between consecutive function calls, the perfect scaling can be highly misleading. The region mainly consists of computation, so it is a good candidate for SIMD vectorization and, potentially, GPU offloading.

9.5 Gravity (remote tree-traversal) - Region 3

The tree traversal also requires remotely accessing tree pieces that are part of other processors. Requests for information on the remote walk is done before the local walk is started to facilitate overlap between communication and computation. When messages have been fully received a remote tree walk can begin and `calculateGravityRemote()` is called. To reduce the number of messages communicated over compute nodes a CacheManager is used, which ensures that two or more tree pieces do not request the same remote information.

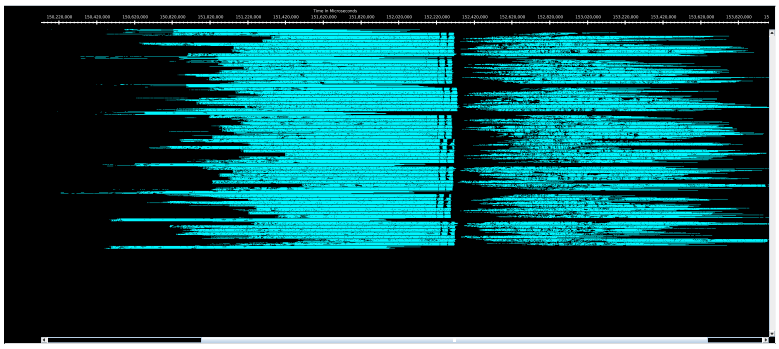


Figure 89: Zoomed traces for Region 3 of the ChaNGa from Projections showing the structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	2.808874	1.527062	0.824135	0.413444
Efficiency	1.0	0.919699	0.852067	0.849230
Speedup	1.0	1.839397	3.408269	6.793844

Table 36: Overview of the key performance metrics of Region 3 of ChaNGa.

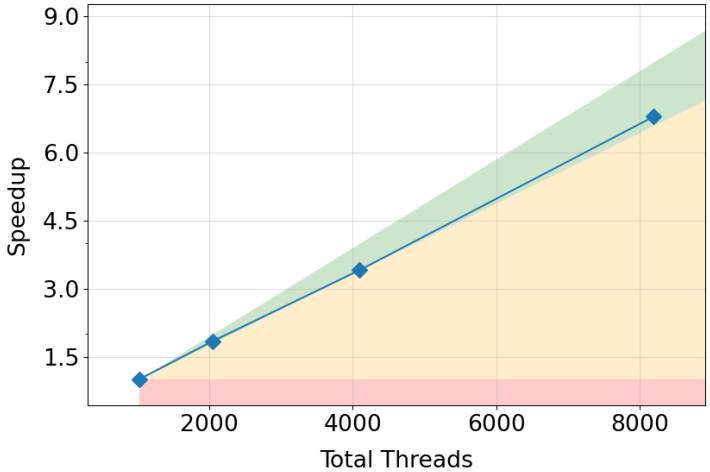


Figure 90: Strong scaling for Region 3 of ChaNGa.

The traces, presented in Figure 89, show two distinct parts of this region. The Figure 90 and Table 36 show the scaling and efficiency of the computation (again omitting any communication or idle times in between calls), which is in acceptable range. This region, in general, contributes the majority to the runtime. Therefore, the most of computation performance optimization should be applied to this compute kernel. Judging from the timeline the load balance of the computation can be improved as well.

9.6 Hydrodynamics (start of step) - Region 4

The `startsmooth()` function includes the initial neighbour-finding algorithm, tasked with identifying the N_{smooth} nearest neighbors of each particle ($N_{smooth} = 64$ in this case). The neighbour-finding operation is facilitated by the already spatially constructed tree build. Upon identifying the neighbors, this function proceeds to call the `nextBucketSmooth()` function, which primarily calculates the density from the density estimate:

$$\rho_a = \sum_{b}^{N_{smooth}} m_b W_{ab} \tag{33}$$

Here a and b are particle indices, m is the mass and W_{ab} is the smoothing kernel.

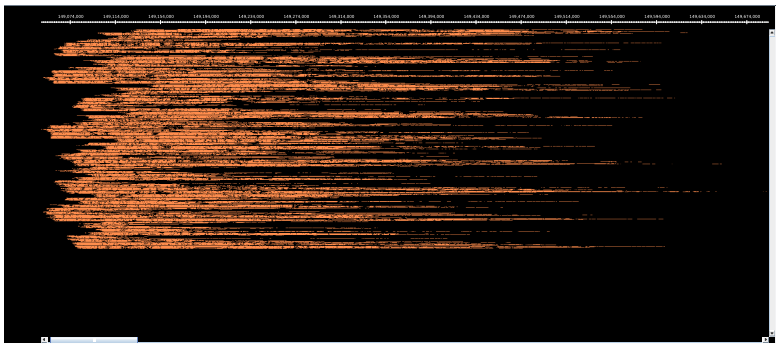


Figure 91: Zoomed traces for Region 4 of the ChaNGa from Projections showing the structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	0.310082	0.163483	0.097464	0.042256
Efficiency	1.0	0.948355	0.795370	0.917266
Speedup	1.0	1.896710	3.181482	7.338129

Table 37: Overview of the key performance metrics of Region 4 of ChaNGa.

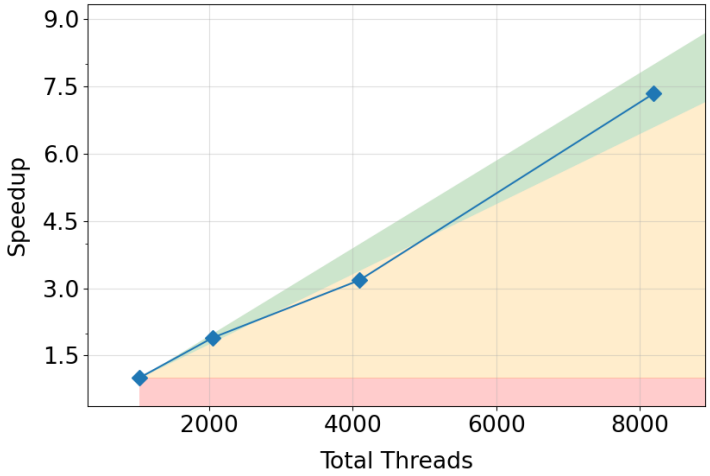


Figure 92: Strong scaling for Region 4 of ChaNGa.

Judging from Table 37, this region constitutes relatively small portion of the runtime. However, it directly precedes the region 2 and as such extra attention has to be paid to the load balance of this particular kernel, which can be improved according to Figure 91. The region exhibits no issues with respect to the computation scaling and efficiency depicted in Figure 92.

9.7 Hydrodynamics (rest of step) - Region 5

The `NextBucketResmooth()` works very similar to `NextBucketSmooth()` function call, where the main difference is that for `NextBuckerResmooth()` the nearest neighbours have already been collected. This function call calculates the gradients of the right hand side of the magneto-hydrodynamic equations:

$$\frac{dv_a^i}{dt} = \sum_b \frac{m_b}{\rho_a \rho_b} \left(S_a^{ij} + S_b^{ij} \right) \nabla_a^j \overline{W_{ab}} + F_{v,diss}, \tag{34}$$

$$\frac{du_a}{dt} = \frac{P_a}{\rho_a} \sum_b \frac{m_b}{\rho_b} (v_b - v_a) \cdot \nabla_a \overline{W_{ab}} + F_{u,diss}, \tag{35}$$

$$\frac{dB_a}{dt} = \sum_b \frac{m_b}{\rho_b} \left[B_a (v_{ab} \cdot \nabla_a \overline{W_{ab}}) - v_{ab} (B_a \cdot \nabla_a \overline{W_{ab}}) \right] + F_{B,diss} + F_{B,clean} \tag{36}$$

Where, B is the magnetic field vector, ρ is the density, S is the stress tensor($S = -\delta^{ij} \left(P + \frac{B^2}{2} \right) + B^i B^j$), P is the pressure, $\overline{W_{ab}}$ is the smoothing kernel, $F_{v,diss}$ is the momentum term of artificial viscosity, $F_{u,diss}$ is the heat dissipation term of artificial viscosity, resistivity and thermal diffusion, $F_{B,diss}$ is the magnetic resistivity and $F_{B,clean}$ is the divergence cleaning function.

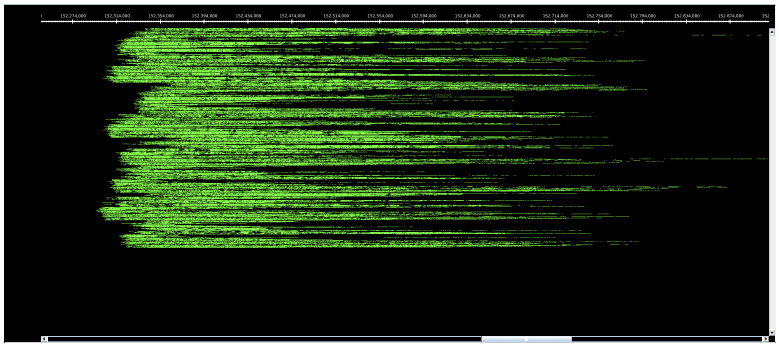


Figure 93: Zoomed traces for Region 5 of the ChaNGa from Projections showing the structure of the region.

Number of processes	1024	2048	4096	8192
Elapsed time (sec)	0.360767	0.201167	0.123375	0.067879
Efficiency	1.0	0.896685	0.731037	0.664357
Speedup	1.0	1.793371	2.924150	5.314854

Table 38: Overview of the key performance metrics of Region 5 of ChaNGa.

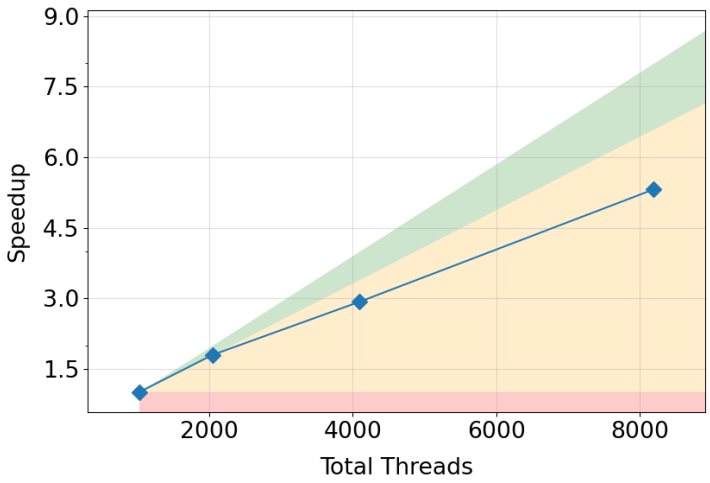


Figure 94: Strong scaling for Region 5 of ChaNGa.

This region separates region 3 in two distinct parts. Again, we suggest focusing mainly on load balance rather than performance optimization, judging from the traces in Figure 93. According to data in Figure 94 the scaling is not optimal. Although, based on runtime in Tab 38, this region constitutes only a small fraction of runtime of entire timestep, it is only a region with sub-optimal computational scaling. In exascale simulations this region can potentially domiante the total runtime. Therefore, improving this scaling of this region will translate to improvements in scaling of entire simulation.

9.8 Conclusions

The most apparent aspect of the performance assessment is drastic difference in efficiency and scaling of respective regions and entire time step. While regions exhibits super-linear scaling at times the entire time step scales extremely poorly. This is the result of several factors. Most importantly Projections tool provides only aggregated runtime information of respective functions thus omitting communication and idle times. This information is only available in unfiltered traces of entire time step. Due to asynchronous nature the Charm++ there are no well-defined boundaries of region. Functions belonging to different regions are usually scheduled asynchronously by the runtime and thus different regions usually blend together. Therefore any information about runtime of particular region is only relevant when accumulated. See more about Projections in Section 2.3.

The conclusion from the discrepancy in scaling between the regions and time step is that the most effort should be invested into optimizing communication and minimizing overhead of the Charm++ runtime rather than to improve the performance of respective compute kernels.

The poor parallel scaling seen in the overall time step is likely due to the communication imbalance produced by the highly clustered dataset. As we mentioned in section 8.4 and 8.5, during the tree traversal stage, remote requests are sent to nodes outside of the local cache. In a highly clustered dataset, some tree nodes receive much more requests than others. This significantly delays the receiving of messages for the remote tree walk. Even though there is overlap between communication and computation, there is insufficient local computation to overlap with the extended delay in receiving messages. This effect is exacerbated as we increase the node count, as this increases the communication imbalance. One potential solution to the communication issues is to investigate the capabilities of Charm++ module TRAM. This module is designed to automatically aggregate messages and detect collective communication patterns to improve communication within Charm++ application.

Apart from improving the Charm++ runtime to address the communication imbalance, the next step would be to improve the cache manager. One potential solution would be to introduce a node-wide cache. Another solution could be to replicate the requested information, to ensure that no single processor becomes overloaded with messages.

This document is Public (PU)
and was produced under SPACE project EU GA 101093441

10 Conclusions

This document presents the initial work on SPACE code performance analysis using the POP CoE methodology and tools. In order to strengthen the upcomming collaboration with POP3 CoE, all codes were instrumented and traced using state-of-the-art European performance analysis tools also used by POP. Furthermore, we are immediately prepared to collaborate with POP3 CoE when it starts with in-depth analysis very efficiently, because we can directly provide the traces produced in the frame of this study to POP experts.

Code name	Section	Issues detected from POP metrics: load balance	comm. efficiency	computation scalability	Region type	Recommended optimization	Target for GPU offload
Pluto	3						
	3.3 Region 1	x	N/A	good	compute	LB, INR	yes
	3.4 Region 3	x	N/A	good	compute	LB, INR	yes
	3.5 Region 6	x	N/A	good	compute	LB, INR	yes
	3.6 Region 9	x	N/A	good	compute	LB, INR	yes
	3.7 Region 10	x	transfer eff.	x	comm.	INA/comm.	no
Gadget	4						
	4.3 Region 0	x	x	good	compute/comm.	LB, INA/comm.	yes
	4.4 Region 1	x	x	good	comm.	LB, INA/comm.	no
	4.5 Region 2	x	x	good	comm.	LB, INA/comm.	no
	4.6 Region 3	x	good	good	compute	INR	yes
	4.7 Region 4	good	x	good	comm.	INR	yes
	4.8 Region 5	good	x	good	compute	INA/comm.	yes
	4.9 Region 6	x	good	good	compute	INR	yes
iPic3D	5						
	5.3 Region 1	x	x	good	compute/comm.	INA/comm	no
	5.4 Region 2	good	x	x	compute/comm.	INA/comm	no
	5.5 Region 3	x	x	good	compute/comm.	INA/comm	no
	5.6 Region 4	x	x	good	compute/comm.	INA/comm	no
RAMSES	6						
	6.3 Region 1	x	good	good	compute	INR	yes
	6.4 Region 2	x	good	good	comm.	INA/comm	no
	6.5 Region 3	x	good	x	comm.	INA/comm	no
BHAC	7						
	7.3 Region 1	x	x	x	compute	INR/comm	no
	7.4 Region 2	x	good	good	compute	LB	yes
FIL	8						
	8.4 Region 0	x	x	x	compute/comm.	LB, INA/comm	no
	8.5 Region 1	x	N/A	x	compute	INR	yes
	8.6 Region 2	x	N/A	x	compute	INR	yes
Changa	9						
	9.4 Region 2	x	N/A	good	compute	LB, INR	yes
	9.5 Region 3	x	N/A	good	compute/comm	LB, INA/comm	no
	9.6 Region 4	x	N/A	good	compute	LB	no
	9.7 Region 5	x	N/A	good	compute	LB	no

Table 39: A summary table of POP analysis of the selected regions of all codes including the region type and recommendations for the future optimization (time and energy) and co-design tasks in WP2. (Abbreviations: comm. - communication, LB - load balance, INA/comm - IntrA-Node/communication optimizations, INR - InteR-Node optimizations including single-core optimization like vectorization

There are several key points that this deliverable achieved:

1. we have used unified and standardized metrics to quantify the scalability properties of the codes and setup the baseline for further improvements;
2. the performance evaluation was done at scale; some codes were analyzed on up to 16,384 CPU cores (128 nodes of the Karolina cluster), allowing us to see bottlenecks that are not necessarily present at lower scales;
3. we have worked very closely with code owners to identify regions of interest and focused the work of the analysis of these particular regions;

4. for every annotated region, and when possible, we have evaluated the POP metrics and highlighted the potential reasons behind the observed limited scalability and parallel efficiency in general;

5. we have identified possible optimization actions for every annotated region.

We summarize the observations and recommendations in Table 39 in a compact way. At the same time, the table gives a feeling about the very large amount of work that has been done to prepare the final document. In addition, we have also identified several other performance issues that are not reported in this deliverable. But these issues were presented to the code owners.

We are confident that the results achieved during this first stage of the proposal will serve as a solid starting point for the next set of efforts and activities planned for the following months.

References

[1] "Pop performance optimisation and productivity," https://pop-coe.eu/, accessed: 28.10.2023.

[2] "Pop standard metrics for parallel performance analysis," https://pop-coe.eu/node/69, accessed: 28.10.2023.

[3] "Karolina supercomputer at it4innovation - documentation," https://docs.it4i.cz/karolina/introduction/, accessed: 28.10.2023.

[4] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato, "Adaptive techniques for clustered N-body cosmological simulations," *Computational Astrophysics and Cosmology*, vol. 2, p. 1, Mar. 2015.

[5] "Bsc tools - extrae," https://tools.bsc.es/extrae, accessed: 28.10.2023.

[6] "Bsc tools - paraver," https://tools.bsc.es/paraver, accessed: 28.10.2023.

[7] "Pypop - documentation," https://numericalalgorithmsgroup.github.io/pypop/doc.html, accessed: 28.10.2023.

[8] "Projections - performance analysis/visualization framework for charm++," https://charm.readthedocs.io/en/latest/projections/manual.html, accessed: 28.10.2023.

[9] "The pluto code for astrophysical gasdynamics," http://plutocode.ph.unito.it/, accessed: 28.10.2023.

[10] V. Springel, "The cosmological simulation code GADGET-2," , vol. 364, no. 4, pp. 1105–1134, Dec. 2005.

[11] A. M. Beck, G. Murante, A. Arth, R. S. Remus, A. F. Teklu, J. M. F. Donnert, S. Planelles, M. C. Beck, P. Förster, M. Imgrund, K. Dolag, and S. Borgani, "An improved SPH scheme for cosmological simulations," , vol. 455, no. 2, pp. 2110–2130, Jan. 2016.

[12] F. Groth, U. P. Steinwandel, M. Valentini, and K. Dolag, "The cosmological simulation code OPENGADGET3 - implementation of meshless finite mass," , vol. 526, no. 1, pp. 616–644, Nov. 2023.

[13] A. Ragagnin, K. Dolag, M. Wagner, C. Gheller, C. Roffler, D. Goz, D. Hubber, and A. Arth, "Gadget3 on GPUs with OpenACC," *arXiv e-prints*, p. arXiv:2003.10850, Mar. 2020.

[14] G. Lapenta, "ipic3d," 2015, accessed: 2023-10-29. [Online]. Available: https://github.com/CmPA/iPic3D.git

[15] S. Markidis, G. Lapenta *et al.*, "Multi-scale simulations of plasma with ipic3d," *Mathematics and Computers in Simulation*, vol. 80, no. 7, pp. 1509–1519, 2010.

[16] G. Lapenta, "Exactly energy conserving semi-implicit particle in cell formulation," *Journal of Computational Physics*, vol. 334, pp. 349–366, 2017.

[17] D. Tskhakaya, K. Matyash, R. Schneider, and F. Taccogna, "The particle-in-cell method," *Contributions to Plasma Physics*, vol. 47, no. 8-9, pp. 563–594, 2007.

[18] R. Teyssier, "Cosmological hydrodynamics with adaptive mesh refinement. A new high resolution code called RAMSES," , vol. 385, pp. 337–364, Apr. 2002.

[19] "Reposiroty of the ramses code on bitbucket," https://bitbucket.org/rteyssie/ramses/src/master/, accessed: 29.10.2023.

[20] O. Porth, H. Olivares, Y. Mizuno, Z. Younsi, L. Rezzolla, M. Moscibrodzka, H. Falcke, and M. Kramer, "The Black Hole Accretion Code," *Computational Astrophysics and Cosmology*, vol. 4, 2017.

[21] H. Olivares, O. Porth, J. Davelaar, E. R. Most, C. M. Fromm, Y. Mizuno, Z. Younsi, and L. Rezzolla, "Constrained transport and adaptive mesh refinement in the black hole accretion code," *Astronomy & Astrophysics*, vol. 629, p. A61, 2019.

[22] "The Black Hole Accretion Code – Documentation," https://bhac.science, accessed: 2023-11-01.

[23] "Repository of BHAC on GitLab," https://gitlab.itp.uni-frankfurt.de/BHAC-release/bhac, accessed: 2023-11-01.

[24] "The MPI - Adaptive Mesh Refinement - Versatile Advection Code – Documentation," https://amrvac.org/, accessed: 2023-11-01.

[25] "Repository of MPI-AMRVAC on GitHub," https://github.com/amrvac/amrvac, accessed: 2023-11-01.

[26] Z. B. Etienne, V. Paschalidis, R. Haas, P. Mösta, and S. L. Shapiro, "IllinoisGRMHD: An Open-Source, User-Friendly GRMHD Code for Dynamical Spacetimes," *Class. Quant. Grav.*, vol. 32, p. 175009, 2015.

[27] M. Zilhão and F. Löffler, "An Introduction to the Einstein Toolkit," *Int. J. Mod. Phys. A*, vol. 28, p. 1340014, 2013.

[28] "Repository of the changa code on github," https://github.com/N-BodyShop/changa, accessed: 28.10.2023.

[29] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[30] J. W. Wadsley, B. W. Keller, and T. R. Quinn, "Gasoline2: a modern smoothed particle hydrodynamics code," , vol. 471, no. 2, pp. 2357–2369, Oct. 2017.

[31] J. G. Stadel, "Cosmological N-body simulations and their analysis," Ph.D. dissertation, University of Washington, Seattle, Jan. 2001.