



SCALABLE PARALLEL ASTROPHYSICAL CODES FOR EXASCALE

First report on optimization

Deliverable number: D2.3

Version 1.0/1.0



Co-funded by
the European Union



EuroHPC
Joint Undertaking

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:	Benoît Commerçon – CNRS
Deliverable nature:	Report (R)
Dissemination level:	Public (PU)
Contractual Delivery Date:	31.12.2024
Actual Delivery Date	31.12.2024
Number of pages:	60
Keywords:	performance, optimization
Authors:	Andrea Mignone, Marco Rossazza – UniTo Pranab J Deka, Fabio Bacchini – KU Leuven Lubomir Riha, Kristian Kadlubiak, Joao Barbosa – IT4I@VSB Tine Colman, Benoît Commerçon – CRAL CNRS Matthieu Kuhn, Marc Sergent – Eviden Geray Karademir – LMU Khalil Pierre, Georgios Doulis – GUF Robert Wissing – UiO Luca Tornatore, Giuliano Taffoni – INAF Vasilis Flouris – FORTH
Peer review:	Guillermo Marin – BSC, Gino Perna – ENGINSOFT

History of Changes

Release	Date	Author, Organization	Description of changes
0.1	01.10.2024	Lubomir Riha, IT4I@VSB; Gino Perna, ENGINSOFT	Started the document and setup of the document structure.
0.2	5.12.2024	all authors	Version for review
0.3	19.12.2024	Guillermo Marin – BSC, Gino Perna – ENGINSOFT	Internal reviewers comments
0.4	20.12.2024	all authors	Version after review
1.0	27.12.2024	Benoît Commerçon, CRAL CNRS, Lubomir Riha, IT4I@VSB	Final version

Scalable Parallel Astrophysical Codes for Exascale

DISCLAIMER

Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European High Performance Computing Joint Undertaking (JU) and Belgium, Czech Republic, France, Germany, Greece, Italy, Norway, and Spain. Neither the European Union nor the granting authority can be held responsible for them.



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

Executive Summary

This deliverable reports on the optimization activities performed by month M24 for the seven codes in the SPACE project. Each code partner describes the ongoing optimization strategy, which has been designed after the work done in previous tasks and reported in deliverables D1.2, D2.1 and D2.2.

The optimizations are focused either on a specific kernel or on the code as a whole, and fall on the following categories: vectorization improvement, CPU optimization, GPU porting and optimization, and scalability improvements. The optimizations are described with technical terms. For CPUs, most of the work relates to the communications between MPI domains and to the implementation of hybrid MPI and OpenMP parallelization. For GPU porting and optimization, different strategies have been proposed. These cover a broad overview of approaches to port existing codes to GPUs, such as CUDA, Kokkos, and OpenACC.

Finally, the gains obtained regarding CPU, GPU and scalability performances are assessed for each code. The future actions envisioned for further code optimization are also briefly discussed.

This deliverable serves as a proof that the project reached Milestone 6 in M24 which states: **"At least 30% of codes optimized at preliminary level, or 100% of codes have optimized at least one fundamental kernel"**. The following table details the number of optimized kernels for each code, and an assessment of the optimization level.

Code	number of optimized kernels	application optimized at preliminary level
PLUTO	3	YES - (a) with the new version of the code gPLUTO (GPU) we obtain around 30X speed-up respect to the legacy CPU PLUTO; (b) the parallel scalability was improved (30%) thanks to the asynchronous boundary kernel
OpenGadget	1	YES - 15 to 20% overall expected improvement in application performance
iPIC3D	3	YES (further performance improvement can be expected only for the GPU version, CPU version fully optimized)
RAMSES	1	YES (CPU version of Godunov solver fully optimized)
BHAC	2	YES (further improvement expected for GPU version)
FIL	2	YES (further improvement expected for GPU version)
ChaNGa	3	YES (but more improvement can be expected for both GPU and CPU versions)

Table 1: Summary table of code optimization actions with respect to Milestone 6.

Contents

1	Introduction	8
2	PLUTO	9
2.1	Previous Optimization Activities	9
2.2	Asynchronous Boundary	9
2.3	Functions Inlining	9
2.4	Usage of C++ Templated Functions	12
2.5	Debug messages	12
2.6	Conclusions	13
3	OpenGadget	15
3.1	Building blocks of code refactoring	15
3.2	GPU offloading	20
3.3	CPU updates	29
3.4	Conclusions	29
4	iPIC3D	31
4.1	Asynchronous Boundary	31
4.2	OpenMP + MPI	33
4.3	Conclusions	33
5	RAMSES	35
5.1	Summary of the optimization work for the entire code	35
5.2	CPU optimization with OpenMP	35
5.3	CPU optimizations	39
5.4	Conclusions	42
6	BHAC	43
6.1	Short description of the code	43
6.2	Summary of optimization work for the entire code	43
6.3	Summary of optimizations effort performed on individual kernels, modules	44
6.4	Conservative to primitive routine (<code>con2prim</code>)	44
6.5	Riemann solver	45
6.6	Conclusions	46
7	FIL/GRACE	47
7.1	Short description of the code	47
7.2	Summary of optimization work for entire code	47
7.3	Kernel porting and optimization	49
7.4	Conclusions	50
8	ChaNGa	51
8.1	Summary of optimization work for entire code	51
8.2	Scalability optimization	51
8.3	GPU porting and optimizations	53
8.4	Conclusion	55
9	Conclusions	56

List of Figures

1	Representation of the synchronous way of exchanging inter-process ghost points.	10
2	Direct exchange of the edge point as done in the asynchronous implementation.	11
3	Comparison of weak scaling tests of PLUTO on Leonardo Booster with synchronous (a, left panel) and asynchronous (b, right panel) approaches. Each node is equipped with 1 CPU with 32 Cores and 4 GPUs.	11
4	Example of a local array depending on the direction of integration in the HLLD solver.	12
5	Newer version of the HLLD solver using templates in PLUTO.	13
6	Top: network control unit (NCU) profiles of the HLLD solvers with indexes known at runtime. Bottom: Same but with indexes known at compilation time.	14
7	Vectorization results obtained on Leonardo with two different compilers, gnu gcc and Intel icx compiler, for a case with 50M particles in total and $\sim 1\text{M}$ particles active, each with ~ 200 neighbors.	17
8	A sketch of topology awareness. Communications among two nodes ("0" and "1" in this example) are managed by two tasks, one per node, designated as "node masters". A "inter-node" communicator includes only node masters. Within each node, tasks may be either grouped in communicators specific to each NUMA node (with a designated master that exchanges data with other masters) or live in a unique Communicator and share a fraction of the node memory as MPI shared memory. Green tasks represent the "masters".	20
9	[Left Panel] Schematic view of the code's workflow. The operational blocks on the left are grouped by the particles' timesteps they refer to. Low timesteps levels are occupied by particles with small timesteps and higher levels by those with larger timesteps. Blocks with green tags represent routines that solve local particle-wise physical processes that do not involve inter-particle interactions. Blocks with orange tags represent points that rely on a tree walk. Magenta tags indicate sections that require MPI communications. [Right Panel]. Zoom-in on the tree walk pattern. The Tree walk phase is always split into two symmetric sub-phases, each of which involves a multi-threaded computation stage followed by a communication stage.	21
10	The speed-up of GPU over CPU implementation for an entire cosmological box (2048^3) with Gravity only. This plot is courtesy of Dr. Tiago Batalha de Castro (INAF) who ran the simulation on Leonardo using 32 nodes in the frame of another science-focused project. On the x -axis there is the expansion factor a of the Universe: smaller values correspond to pristine times, whereas values closer to 1 correspond to today. The conversion in time is increasingly non-linear towards small a : the reader should consider that $\sim 0.57\%$ and $\sim 0.76\%$ of the time elapses from $a = 0.5$ and $a = 0.3333$ to $a = 1$, respectively. Only $\sim 4\%$ of the time elapses from $a \gtrsim 0$ and $a = 0.1$. As for the computational time, the proportion is even more severe since from $a \gtrsim 0.1$, the formation of substructures leads to deeper and deeper potential well, which reflects in larger accelerations and, hence, the need for smaller timesteps. The effective speed-up of the simulation is then the one at $a = 1$, which is ~ 3	23
11	Details of the 2048^3 boxes, gravity only. The <i>parallel efficiency</i> (left) and the <i>timing breakdown</i> (right) of the gravity-only case.	23
12	A screenshot from the NVIDIA profiler with the roofline model of the gravity tree routine that implements the Barnes&Hut algorithm. The blue and red points refer to single- and double-precision floating-point operations. Maximum theoretical peaks for double- and single-precision are the lower and higher horizontal lines. The bottom panel shows the relative efficiency in memory transfer and calculations. Evidently, the current implementation of the Barnes&Hut is severely under-performing due to algorithmic limits.	24

13	An example of 2D particle distribution (upper row, left panel), and of the correspondent quad-tree representation (bottom panel). In the tree representation, the empty nodes are pale-azure dots, the particles (i.e. the leaves) are green circles, and the nodes are ivory circles. To help follow how the tree is built, the children nodes of node N have been named as {NA, NB, NC, ND} with A and D being the northwest and southwest nodes, respectively. To further exemplify, the region framed in magenta in upper left panel corresponds to the same region framed in the bottom panel. See also Sec. 3.2.2. The target particle used in the discussion is the child of node AA, with violet background. In the upper row, right panel , we show some examples of nodes too wide to be open (those with the red angle of view) and of nodes small enough to be open (those with the green angle of view). Particles highlighted with a yellow contour are some of those that will be added in direct summation.	25
14	In the upper panel we sketch the organization of the Tree structure in memory for the tree shown in Fig. 13. Colors refer to the same main branch, <i>A, B, C, D</i> , while they become fainter with the depth of the tree. Red arrows remind the reader that all the nodes are subsequent in memory; the different sections have been split into different lines to enhance the readability. In the Bottom panel we reveal the implementation of the indexed traversal; from a given node a complete tree traversal is possible without having to actually walk from the root node. Some of the same links are shown in the upper panel. See the text for more details.	27
15	NVIDIA's NCU screenshot. Top Panel: performance metrics, Bottom left Panel: memory characteristics, Bottom Right Panel: thread divergence and stall. Comparison of experimental kernel run in "OpenGadget-like" mode and with the new re-arrangement of active particles. Note (i) the much smaller kernel time (0.68s vs 5.5s), (ii) the much larger compute throughput (63%SoL vs 8%SoL), (iii) that the v1 kernel is compute-bound whereas v0 is memory-bound and that (iv) that the thread divergence is much smaller. This NVIDIA's NCU screenshot is from comparison mode, with the baseline set to v0.	28
16	Comparison of the required time per time step at different numbers of particles in each time bin for the old (black) version. The one with the improved loops (blue) and the first updates on the gradient calculation (red). It can be seen that the time required for the smallest time steps is reduced by a factor of ~ 2 and by the optimized loops and even further to a factor of ~ 5 with the updated gradient calculation.	29
17	Figure shows the runtimes of <code>MPI_Barrier</code> (red lines) bounded by <code>MPI_Allreduce</code> (pink lines, top) and <code>MPI_Sendrecv_replace</code> (green lines, middle and bottom). It is evident that a large portion of the code runtime is wasted, i.e., no computation is performed, whilst waiting for the data to be transferred using blocking communication. These plots were obtained during the POP analysis.	32
18	Figure illustrates the speed-up obtained with shared memory parallelization using OpenMP over using MPI-only parallelism on Leonardo DCGP. All data points are normalised to the runtime for 112 tasks using MPI-only parallelism. One can clearly see that we get $\sim 20\%$ improvement in performance whilst using OpenMP.	34
19	RAMSES - Comparison of the strong scaling of the MPI-only versus hybrid MPI+OpenMP implementation for the Sedov test case. Top: Test performed on the internal Eviden cluster with BullSequana X2410 blades equipped with AMD Milan architecture processors with 128 cores (256 threads) per node. The code was compiled with the AOCC compiler. Bottom: Benchmark performed on the MeluXina cpu partition using the GNU + OpenMP compiler.	36
20	RAMSES - Comparison of the time spent on the different parts of the code for the Sedov test case, using the MPI-only implementation versus the new hybrid parallelization. The routines (rev) <code>ghostzones</code> deal directly with MPI communication. The routines <code>courant</code> and <code>coarse levels</code> contain an <code>MPI_ALLREDUCE</code> . Top: Eviden cluster, bottom: MeluXina CPU partition. . . .	37
21	Memory usage per node for the strong scaling benchmark of the Sedov test on MeluXina for the RAMSES code.	39
22	Roofline analysis of the hydro module of RAMSES, Sedov blast benchmark with a 128^3 mesh. . .	41
23	RAMSES <code>godfine1</code> subroutine, original uloc filling.	41
24	RAMSES <code>godfine1</code> subroutine, uloc filling optimized.	41
25	Timeline histograms of initial (top) and optimized (bottom) version of BHAC. Wait and overhead time (see color legends) are enormously reduced while CPU utilization is increased at the bottom panel.	43

26	Nsight Systems timeline histograms of part of the <code>con2prim</code> module: semi-sequential version (top) and fully parallel version (bottom). The problematic kernel can be easily identified by abnormally long runtime. The performance increase is approximately 116-fold.	45
27	Nsight Systems timeline histograms of initial (top) and optimized (bottom) version of BHAC. Shown here is part of the Riemann solver module. Host to device and device to host data transfer is enormously reduced and GPU utilization is increased at the bottom panel.	46
28	Strong and weak scaling plot for FIL on Marenstrum, FIL shows excellent scaling up to 512 nodes. Super-optimal scaling is the result of each data point being generated from a single run due to limited computational resources.	47
29	Performance comparison between FIL and GRACE where the performance metric is cell updates per second. This metric ignores IO contributions to the runtime of both codes.	48
30	Breakdown of runtime of FIL and GRACE spent in kernel 1, add flux to RHS and kernel 2 the conservative to primitive solver. The data is a comparison between FIL ran on a single processor on the p03 partition of the IT4I complementary systems and a single M250 GPU on LUMI-G.	49
31	Breakdown of runtime of GRACE by code region on Goethe University Calea partition.	50
32	CPU scaling tests for the full physics merger simulation performed on the Karolina supercomputer (details provided in deliverable D1.1 and D2.1), comparing different numbers of communication threads (denoted as COMx) and configurations with or without tree replication (with tree replication denoted as TP). speed-up is normalized to 1 at 8 nodes, with relative speed-up shown on the left and parallel efficiency on the right. Results indicate significant performance improvements at higher thread counts with the tree replication approach. Increasing communication threads also enhances performance but is not the primary limiting factor at the highest thread counts.	53
33	Scaling of the elapsed time with respect to the number of CUDA streams for a single time step on a single GPU. The three curves represent different numbers of tree pieces, showing that, in this case, a smaller number of tree pieces is more beneficial for GPU performance and that more streams can improve result for more tree pieces.	54

List of Tables

1	Summary table of code optimization actions with respect to Milestone 6.	1
2	Summary of the optimization work performed on PLUTO.	14
3	Labels used in the vectorization study. <code>v1</code> and <code>v2</code> refers to the memory contiguity: in <code>v1</code> , the neighbor particles are not contiguous in memory, and hence looping over them results in memory jumps; in <code>v2</code> the neighbor particles are copied into local buffer before looping over them, and hence the loop is over contiguous memory locations. There are two groups of data structures: Array of Structures (AoS group) and Structure of Arrays (SoA group). AoS cases are treated as big structures ("AoS"), as 4-element vectors that coincide with "small structures" ("AoS _v "), and as small structures explicitly processed with vector intrinsic ("AoS _i "). Finally, the Structure of Arrays group is processed either as standard arrays, referencing one entry per iteration ("SoA"), or with explicit vector intrinsics ("SoA _i "). See text for more details.	16
4	The events set specified for the profiling with the PAPI library, and the compilation options (bottom framed box) used on the Leonardo platform.	18
5	Summary of the optimization work performed on OpenGadget. As mentioned at the beginning of this section, the activity mainly focused on polishing the main code and developing building blocks to be tested in the mini-app.	30
6	Performance comparison of the old and new versions of iPIC3D for 3D magnetic reconnection. The speed-up is defined as the ratio of runtime of the old version to the code to the new one.	31
7	Summary of the optimizations for iPIC3D.	33
8	Overall performance gain on the total runtime of RAMSES depending of the compiler used on Intel Sapphire Rapids.	42
9	Summary of the optimization work performed on RAMSES.	42
10	Summary of the optimization work performed on BHAC.	44
11	Summary of the performance gains of porting FIL kernels to GRACE.	49
12	Summary of the optimization work performed on ChaNGa.	51
13	Summary table of code optimization actions with respect to Milestone 6.	56

14 Summary of the optimization work performed on all codes - part 1. (Note: For iPIC3D there three points we need to highlight: (a) we have a new version of the code that can be 20 - 30 times faster than the older one, and (b) CPU code is production ready — 100% optimized, (c) we have a fully-working GPU code (needs improvements).) 57

15 Summary of the optimization work performed on all codes - part 2 58

List of Acronyms

API	Application Programming Interface
AMR	Adaptive Mesh Refinement
CD	Continuous Deployment
CI	Continuous Integration
GRMHD	General Relativistic Magnetohydrodynamics
HD	Hydrodynamics
HLLD	Harten-Lax-van Leer discontinuity
MHD	Magnetohydrodynamics
MPI	Message Passing Interface
NCCL	NVIDIA Collective Communications Library
OpenMP	Open Multi-Processing
PE	Processing Element
PIC	Particle-in-Cell
POP	Performance Optimization and Productivity
RHD	Relativistic Hydrodynamics
RMHD	Relativistic Magnetohydrodynamics
SFC	Space-Filling Curve
SMP	Symmetric Multi-Processing
SPACE	Scalable Parallel Astrophysical Codes for Exascale
SPMHD	Smoothed Particle Magnetohydrodynamics
SRMHD	Special Relativistic MHD
TOV	Tolman–Oppenheimer–Volkoff
BHAC	Black Hole Accretion Code

1 Introduction

This deliverable reports on the optimization activities performed before month M24 for the seven codes in the SPACE project. All code partners describes their ongoing optimization strategy, which has been designed after the work done in previous tasks and reported in deliverables D1.2, D2.1 and D2.2.

The objective is to provide an extensive review of all the activities that have led to performance gains with respect to CPU, GPU and scalability. Each code has a dedicated section which is organised as follows. After a brief recap of the general properties of the code, the optimizations performed over the first two years of the project are described in detail. These developments include both CPU and GPU implementations.

The optimizations are focused either on a specific kernel or on the code as a whole, and fall on the following categories: vectorization improvement, CPU optimization, GPU porting and optimization, and scalability improvements. The optimizations are described with technical terms. For CPUs, most of the work relates to the communications between MPI domains and to the implementation of hybrid MPI and OpenMP parallelization. For GPU porting and optimization, different strategies have been proposed. These cover a broad overview of approaches to port existing codes to GPUs, such as CUDA, Kokkos, and OpenACC.

Finally, the gains obtained regarding CPU, GPU and scalability performances are assessed for each code. The future actions envisioned for further code optimization are also briefly discussed.

2 PLUTO

PLUTO [1] is designed to integrate a general system of conservation laws that we write as

$$\frac{\partial U}{\partial t} = -\nabla \cdot T(U) + S(U). \quad (1)$$

Here U represents a state vector of conserved quantities (typically density, momentum and energy), $T(U)$ is the flux tensor (rank 2) and $S(U)$ is a source term. The explicit definition of T , S and U depends on the particular physics module that the user selects. In its current implementation, the GPU-ported version (gPLUTO) comes with 5 different modules, HD (Euler equations of gas dynamics), Magneto-Hydrodynamics (MHD) (ideal and resistive magnetohydrodynamics) and their corresponding special relativistic extensions (see, e.g., [17, 18, 19]) plus the non-ideal relativistic MHD equations. A comprehensive review of the numerical methods, algorithms and kernel identification has already been given in D1.2.

2.1 Previous Optimization Activities

Previous achievements in the optimization process have included four physical modules (i.e., [HD](#), [MHD](#) and their relativistic extensions [RHD](#) and [RMHD](#)) and different kernels which include the conservative to primitive mapping functions (4 kernels), reconstruction of primitive or characteristic variables (3 kernels) and Riemann solvers (9 kernels). In the following subsections we describe a second round of optimization, specifically targeting some of these functions, new Riemann solvers and the parallel communication.

2.2 Asynchronous Boundary

The Boundary routine is crucial in PLUTO when the code employs multiple processes. It permits data exchange between them to fill inter-process ghost zones, which is needed to reconstruct function values at zone interfaces. A major improvement has been undergoing during the last year, going from blocking (`MPI_Sendrecv`) to non-blocking (`MPI_Isend` and `MPI_Irecv`) MPI calls and from synchronous to asynchronous kernels. Having independent communication paths for every direction of data exchange required the introduction of a new path to fill all the points needed.

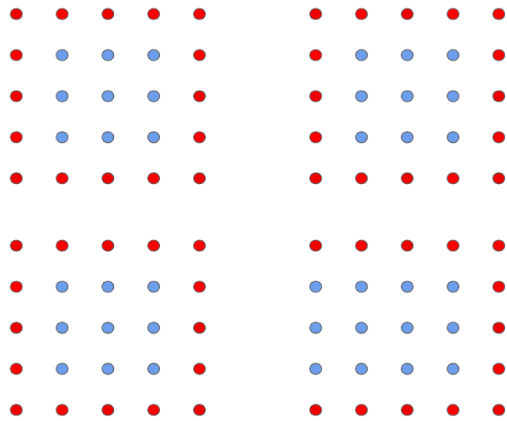
More specifically, the previous (old) approach consisted of 6 directions of exchange (two for x , y and z) taking place sequentially and allowing, eventually, to fill not only face adjoining ghost zones but also domain's corner ghost zones. This is represented - in the case of a 2-D domain split amongst 4 processes - in Fig. 1.

On the other hand, the asynchronous approach requires dedicated communication paths to exchange data in the corners (8 new MPI calls) and edges (12 new MPI calls), for a total of 26 MPI `Send/Recv` calls. Fig. 2 shows the direct communication pattern necessary to fill ghost cells in the edge.

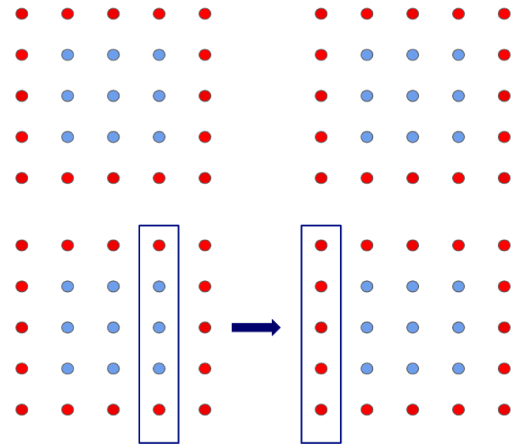
In the weak GPU scaling tests on **Leonardo** with the synchronous approach, the efficiency dropped to almost 60% with [MPI](#) and 80% with [NCCL](#) going to 256 nodes, as reported in Fig.3a. The test problem simulated is the 3D Orszag-Tang with RK3, weighted essential non oscillatory (WENO) reconstruction scheme, [HLLD](#) Riemann Solver and Constrained Transport for the magnetic divergence control. The resolution is 352^3 points per process (i.e. per GPU). On the most recent test employing asynchronous calls we stay around 90% efficiency even at 512 nodes simulating the same problem, as reported in Fig.3b.

2.3 Functions Inlining

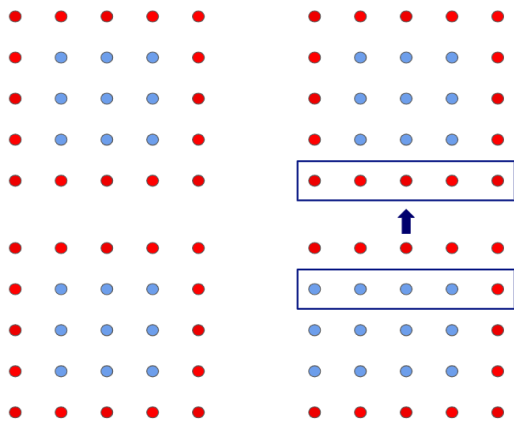
A general optimization strategy that affected especially complex kernels such as Riemann solvers and reconstructions is function inlining. GPUs handle thousands of threads in parallel, and traditional function calls introduce overheads that can bottleneck performance. Inlined functions are expanded directly into the calling function's body, avoiding the cost of saving/restoring registers and jumping to/from another memory location. When a function is inlined, the compiler gains a complete view of the code, allowing it to perform aggressive optimizations: i) manage registers more effectively; ii) reorder instructions to reduce latency; iii) detect opportunities for coalescing memory accesses or using shared memory.



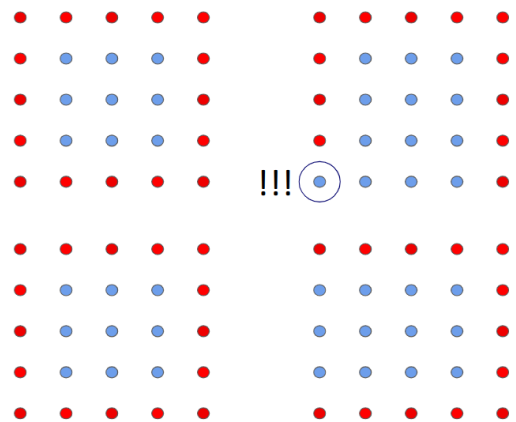
(a) Domain splitted between four processes.



(b) Copy to the right process.



(c) Copy to the upper process.



(d) The corner of the upper-right sub-domain is filled successfully.

Figure 1: Representation of the synchronous way of exchanging inter-process ghost points.

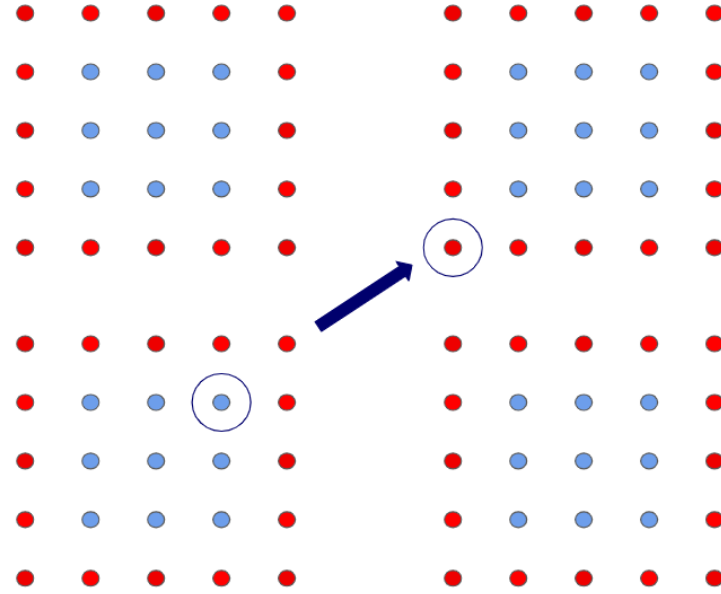
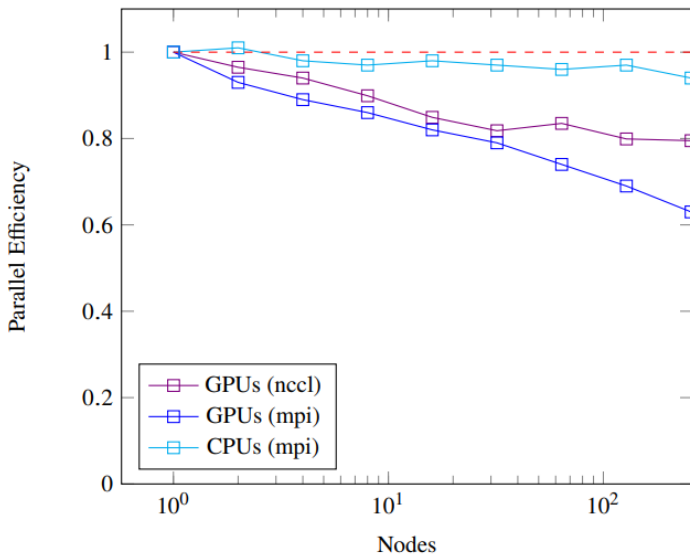
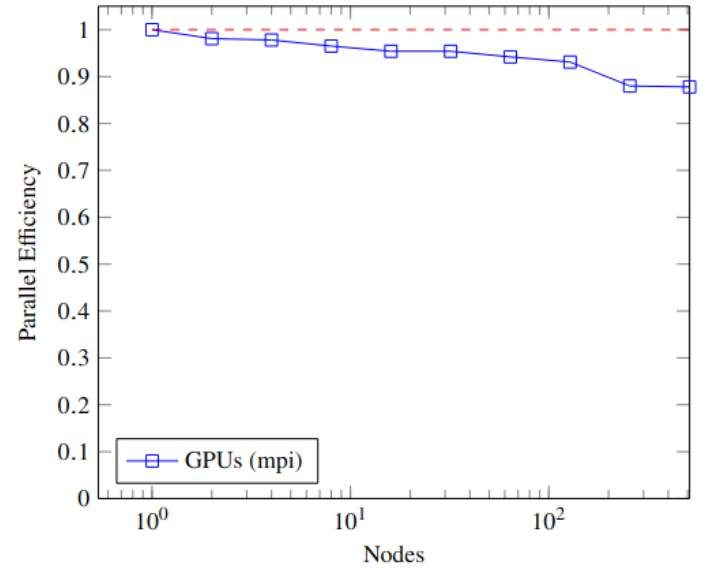


Figure 2: Direct exchange of the edge point as done in the asynchronous implementation.



(a)



(b)

Figure 3: Comparison of weak scaling tests of PLUTO on Leonardo Booster with synchronous (a, **left panel**) and asynchronous (b, **right panel**) approaches. Each node is equipped with 1 CPU with 32 Cores and 4 GPUs.

2.4 Usage of C++ Templated Functions

For reasons similar to inlining, specifically the registers management, C++ templated functions have been introduced in many kernels (such as Riemann solvers and reconstructions) and subroutines.

In the original implementation, we employed local 1-D arrays, whose index depended on the direction of integration as shown in Fig. 4 for the HLLD Riemann solver. As a consequence, the indexes of some variables were runtime-dependent and the compiler could not determine which specific registers would be needed ahead of time. As a result, these variables were often stored in slower memory spaces like local memory (which resides in global memory) or shared memory, rather than registers. Registers, in fact, are allocated by the compiler at compile time, based on the static analysis of the code. If the index of a variable is known at compile time, the compiler can allocate a register for that variable, enabling faster access.

```

#define INDEX_CYCLE(dir, e1, e2, e3) \
    e3 = e1 + (dir+2)%3; \
    e2 = e1 + (dir+1)%3; \
    e1 = e1 + (dir+0)%3;

void HLLD_Solver (Data *d, timeStep *Dts, Grid *grid, RBox *box)
{
    int VXn = VX1, VXt = VX2, VXb = VX3;

    INDEX_CYCLE (dir, VXn, VXt, VXb);

    #pragma acc parallel loop collapse(2)
    for (k = kbeg; k <= kend; k++){
        for (j = jbeg; j <= jend; j++){

            #pragma acc loop
            for (i = ibeg; i <= iend; i++){

                int v[NVAR];

                v[VXn] = ...;

            }
        }
    }
}

```

Figure 4: Example of a local array depending on the direction of integration in the HLLD solver.

To fix this situation and improve optimization, we recurred to C++ templated functions. Templated functions allow for code specialization based on template parameters, in our case `dir`. The compiler can inline the specialized functions and propagate constant values, eliminating the need of branching and thus, improve the performance. In particular, the decision based on `dir` is made at compile time rather than at runtime, which removes the need for runtime conditionals. In Fig.5 we show the templated version of the HLLD solver (note the use of `constexpr` to evaluate indexes at compile time).

The top and bottom panels of Fig. 6 show the impact of the discussed modification on the performance of the HLLD solver. On the left side of both panels is the profile of the kernel, having the indexes such as `BXt` not known at compile time. The execution time is about 390 ms. On the right side is the profile of the new version, with `dir` introduced as a templated parameter. In this case, the execution time is about 270 ms. In the instruction section of the profiles of the slower version we see twice more operations being done, in particular `IMAD.MOV.U32` and `IMAD` that indicate addresses or indexes calculations performed at runtime and `STL` stands for Store within Shared memory, while in the fastest case the same variable resides on the registers. This is just one line of the kernel but the same differences are present everywhere an array uses an indexes such as `BXt`.

2.5 Debug messages

The numerical algorithms implemented in the code are based on the discretization of models of real phenomena. However, the discretization process may introduce truncation or machine error that may lead to violation of

```

#define GEN_INDEX_CYCLE(dir, base, e1, e2, e3) \
    constexpr int e3 = base + (dir+2)%3; \
    constexpr int e2 = base + (dir+1)%3; \
    constexpr int e1 = base + (dir+0)%3;

template<int dir>
void HLLD_Solver (Data *d, timeStep *Dts, Grid *grid, RBox *box);

void HLLD_Solver_T (Data *d, timeStep *Dts, Grid *grid, RBox *box){
    if(g_dir==IDIR) HLLD_Solver<IDIR>(d, Dts, grid, box);
    else if(g_dir==JDIR) HLLD_Solver<JDIR>(d, Dts, grid, box);
    else if(g_dir==KDIR) HLLD_Solver<KDIR>(d, Dts, grid, box);
}

template<int dir>
void HLLD_Solver (Data *d, timeStep *Dts, Grid *grid, RBox *box)
{
    GEN_INDEX_CYCLE (dir, VX1, VXn, VXt, VXb);

    #pragma acc parallel loop collapse(2)
    for (k = kbeg; k <= kend; k++){
        for (j = jbeg; j <= jend; j++){
            #pragma acc loop
            for (i = ibeg; i <= iend; i++){

                int v[NVAR];
                v[VXn] = ...;
            }
        }
    }
}

```

Figure 5: Newer version of the HLLD solver using templates in PLUTO.

physical constraints such as positivity of density or pressure, or superluminal velocities. While there are a number of backup strategies or fail-safe solutions present in the code, error diagnostic is an essential tool for a proper control of the simulation status. In the old legacy code, printing output warning messages during the execution on a CPU was trivial, as the error could be printed as a simple output message inside the kernel at the time the error was generated.

In the GPU programming paradigm, however, print statements inside kernels are highly discouraged since these operations are typically not thread-safe. This results in multiple threads trying to print simultaneously, inducing conflicts and eventually resulting in garbled or interleaved output and unpredictable behavior of the application. For these reasons, it is good practice to move all the outputs outside the kernel parallelization region to reduce the memory transfer from the host to the device. To this purpose, we store debug information in an array and analyze it after the parallel region.

The code now features multiple diagnostic levels ($-1, 1, 2, 3$) that can be specified by the user through the macro `WARNING_MESSAGES`. In all cases, if an error is generated inside a kernel, a corresponding flag is activated. After the kernel execution, outside the parallel region, a dedicated function `Where()` checks whether there are active flags and print an error message whose verbosity depends on the aforementioned macro.

2.6 Conclusions

At present, we can say that approximately $\sim 50\%$ of the original legacy code (PLUTO) has been, not only ported, but also optimized in the upcoming GPU version (gPLUTO). A summary of the activities related to the optimization phase described so far are listed in 2. The next modules that will be imported in the code in the forthcoming months (geometry and mapped grids) will take advantages of the framework and optimization strategies developed so far. Finally, our CI/CD testing process allows for daily check of performance results and consistency. Benchmarks are updated with the new code features and new problems are added as new kernels become available.

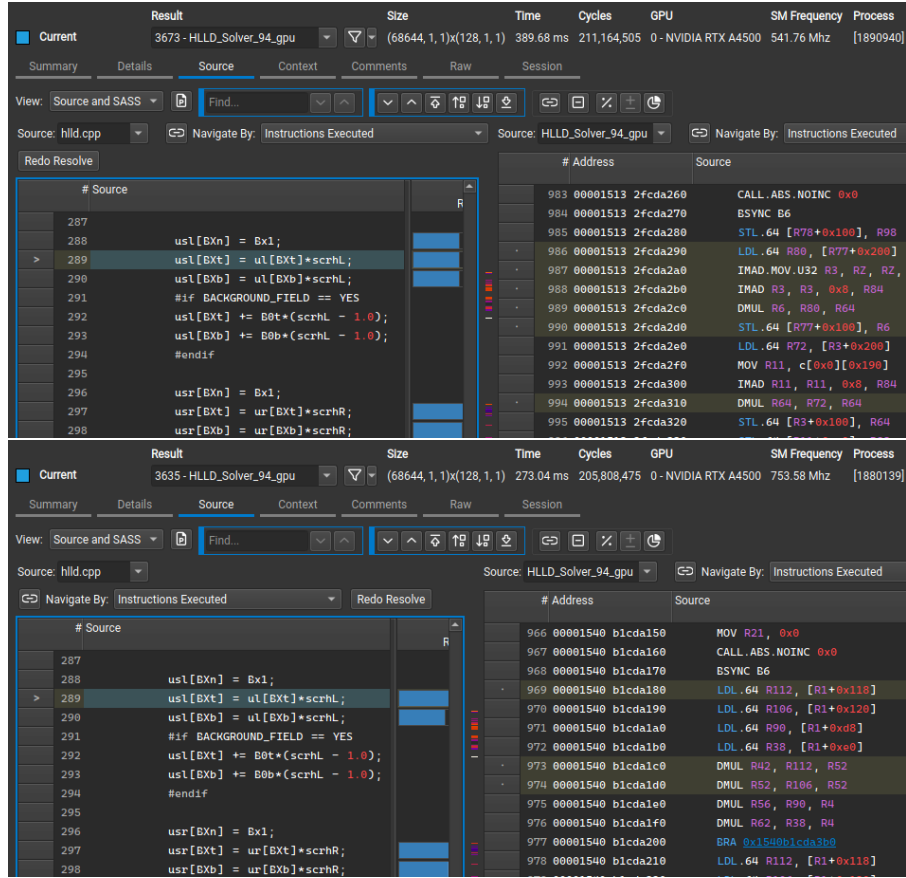


Figure 6: **Top:** network control unit (NCU) profiles of the HLLD solvers with indexes known at runtime. **Bottom:** Same but with indexes known at compilation time.

	CPU optimization		GPU optimization		Scalability optimization		
Kernel name	focus	performance gain [%]	focus	performance gain [%]	focus	number of nodes/G-PU/CPU cores	Parallel efficiency improvement [%]
Riemann Solver	Template and inlining	5%	Template and inlining	10%	—	—	—
Reconstruct	Template and inlining	5%	Template and inlining	10%	—	—	—
Boundary	Asynch. implementation	30%	Asynch. implementation	30%	Asynch. implementation	512/2048/-	30%

Table 2: Summary of the optimization work performed on PLUTO.

3 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 [2]. OpenGadget has been improved significantly compared to its base version by, e.g., adding a new state-of-the-art SPH implementation [3], an MFM solver [4] as well as adding OpenACC support to run on GPUs [5].

3.1 Building blocks of code refactoring

3.1.1 Data structures and vectorization

The code profiling in the first-year activity (see D2.1 and D1.2) has highlighted that the vectorization ratio achieved on average (i.e., on the whole code) is dramatically low, where we define that ratio to be the fraction of vector floating point (FP) instructions issued to the total number of FP instructions.

That is mostly due to the interaction of data structures we use in the code and the compiler's capability of expressing the potential data parallelism of the operations. As such, we started a specific activity to explore how different data structures and computation layouts could impact vectorization in specific kernels.

Adopting radically different particle data structures would significantly impact the code since, in its current status, all the references to the variables that represent the physical quantities involved in every computation are explicitly dependent on the specific data structures. As such, while we acknowledge that it would be advisable that those references be agnostic of the data structures, for the current code development, adopting different structures is a major step that requires careful evaluation.

For that purpose, we studied the achievable vectorization in the direct (i.e. pairwise) Gravity force calculation, which is a good proxy for the typical memory-access and float-per-byte patterns and for the possible speed-up of that operation. In the following, we report the result of this work on vectorization.

We built a kernel code that reproduces the pattern of direct Gravity force calculation as follows:

1. an array of N_p particles is randomly generated with masses m_i distributed in $[0 : 1] \times [0 : 1] \times [0 : 1]$;
2. a subset of N_A particles, randomly chosen, are considered "active", as it happens in OpenGadget where, due to hierarchical time-steps, only a fraction of particles is updated at every time-step;
3. every active particle i interacts with a number $N_{\text{ngb},i}$ of other particles, that in the actual code would be its neighboring particles. We refer to those particles as "the neighbors" in the following;
4. every active particle i calculates the gravitational force resulting from the contribution of its neighbors:

$$\vec{F}_i = m_i \times \sum_{j=0}^{N_{\text{ngb},i}} m_j \cdot \vec{r}_{ij} / \|\vec{r}_{ij}\|^3$$

There are two key points to be highlighted in studying the characteristics of the kernel described above: **(i)** neighbors are not contiguous in memory, and the looping over them amounts to jumps in memory. In fact, in the actual code *(i)* the loop over the neighbors is a loop over the indexes of the neighbors; *(ii)* the particles have no ordering in memory, and, as such, their spatial locality does not correspond to memory contiguity. **(ii)** We use different data structures and techniques to achieve the best possible vectorization and test how that reflects on run-time.

Specifically, we tested five different approaches:

1. **Using Arrays of Structures (AoS)**

AoS (Array of Structures) The particle data structures is:

	Array of Structures			Structure of Arrays	
Memory	large struct.	small struct. as vector	small struct. with intr.	arrays	arrays with intr.
No contiguity	v1.AoS	v1.AoS _{Sv}	v1.AoS _{Si}	v1.SoA	v1.SoA _i
contiguous	v2.AoS	v2.AoS _{Sv}	v2.AoS _{Si}	v2.SoA	v2.SoA _i

Table 3: Labels used in the vectorization study. **v1** and **v2** refers to the memory contiguity: in **v1**, the neighbor particles are not contiguous in memory, and hence looping over them results in memory jumps; in **v2** the neighbor particles are copied into local buffer before looping over them, and hence the loop is over contiguous memory locations. There are two groups of data structures: Array of Structures (AoS group) and Structure of Arrays (SoA group). AoS cases are treated as big structures ("AoS"), as 4-element vectors that coincide with "small structures" ("AoS_{Sv}"), and as small structures explicitly processed with vector intrinsic ("AoS_{Si}"). Finally, the Structure of Arrays group is processed either as standard arrays, referencing one entry per iteration ("SoA"), or with explicit vector intrinsics ("SoA_i"). See text for more details.

```

1      struct { double pos[3];
2              double mass;
3              double filling_elements[FILL]; };
4

```

where the `filling_elements` arrays mimic the "large structure" pattern when `FILL` $\gg$ 1. This case, labelled as **AoS**, reproduces the OpenGadget pattern as it is at the moment of writing.

AoS_S (Array of small Structure) The particle data structure is

```

1      struct { double pos[3];
2              double mass; };
3

```

hence it is an array of "small structure".

We treat this case in two different ways, considering a single element as a **vector**: either using the vector extension that the C compilers offer (we use the label **AoS_{Sv}**), or by explicitly using the vector intrinsics instructions exposed by the compilers (labelled as **AoS_{Si}**). For this study we only used the `x86_64` family of intrinsics.

2. Using Structure of Arrays (SoA)

In this second group, we build separate arrays for every physical quantity

```

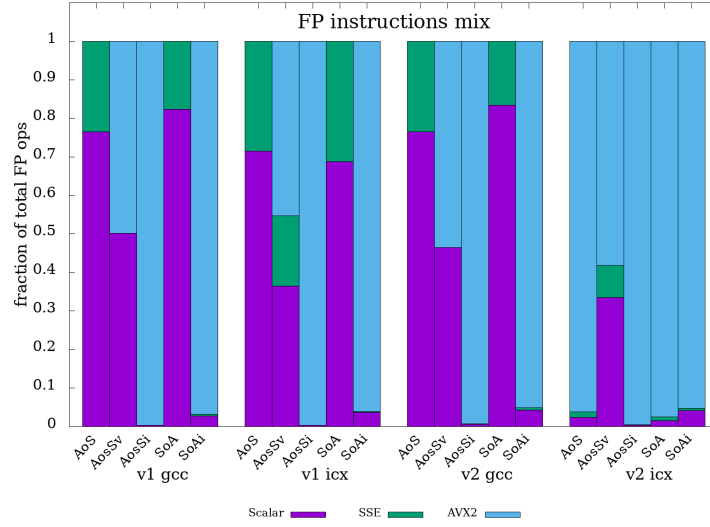
1      double *pos_x;
2      double *pos_y;
3      double *pos_z;
4      double *mass;
5

```

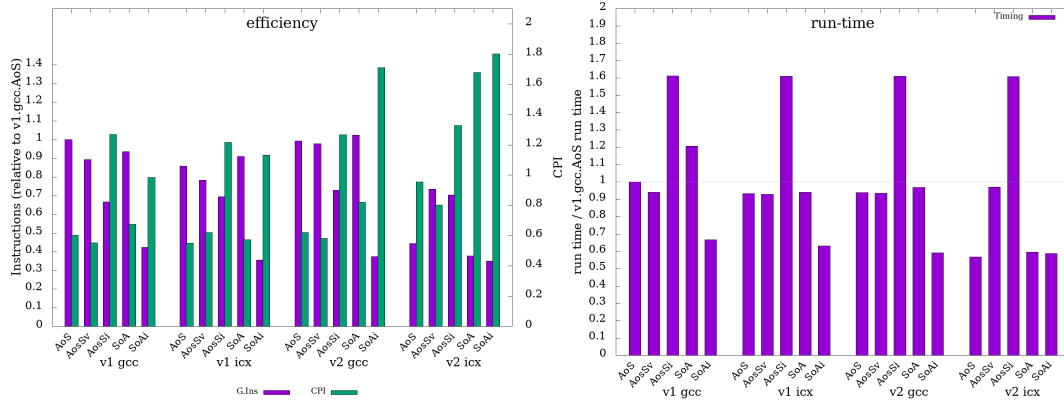
We process them either without any particular formalism, leaving to the compiler to manage the vectorization (label `textbfSoA`), or implementing explicit vectorization by using the vector intrinsics (label **SoA_i**).

In addition, recognizing that jumping in memory from one neighbor to the next can severely hamper the vectorization, we tested the effect of introducing a memory buffer in which we pre-emptively copy all the neighbors; this way, the loop over the neighbors runs through contiguous memory locations. In Tab. 3 we summarize all the options described above:

We have run our tests on both Leonardo and LUMI machines for three different sizes of the problem: 50M particles, 10M particles and 1M particles, with $\sim 10\%$ of particles "active" and with ~ 200 neighbors each. On each system we have tested the results with two different compilers: GNU's `gcc` and Intel's `icx` on Leonardo; the AMD's `AOCC` and the generic `cc` loaded by `module load LUMI/24.03` on LUMI. The results, in terms of relative run time, issued instructions and Cycles-per-Instruction (CPI) are very similar, and hence we discuss only those obtained on Leonardo.



(a) The Floating-Point instruction mix in the different cases, for v1 and v2 (see tab. 3), obtained with gnu gcc and Intel icx compilers on Leonardo. All ops are in double-precision, no single-precision was issued.



(b) [**Left Panel**] The Instructions and the Cycles-per-Instruction (CPI), with violet and green color, respectively, for the different cases. *The instructions are normalized to the reference case, v1.gcc.AoS.* [**Right Panel**] The run-time normalized to the reference case v1.gcc.AoS.

Figure 7: Vectorization results obtained on Leonardo with two different compilers, gnu gcc and Intel icx compiler, for a case with 50M particles in total and $\sim 1\text{M}$ particles active, each with ~ 200 neighbors.

We have also instrumented the code using the PAPI library, in order to profile the precise mix of instructions. However, we note that the performance counters exposed by the AMD architecture are not as rich as the Intel architecture. As such, also in this respect, we discuss only the Leonardo's results, leaving to the future to seek more help to exploit the AMD platform.

The set of events profiled for the CPU (Intel Xeon Platinum 8480p) on Leonardo-DCGP partition is reported in Tab.4 along with the precise command-line used for compilation.

ref	Event Name	Meaning
[a]	PAPI_TOT_INS	Total number of instructions issued
[b]	PAPI_TOT_CYC	Total number of cycles
[c]	PAPI_L2_DCA	Level-2 data cache accesses
[d]	PAPI_L2_LDM	Level-2 data caches misses
[e]	FP_ARITH_INST_RETIRED:SCALAR_DOUBLE	FP scalar ins. double-precision
[f]	FP_ARITH_INST_RETIRED:SCALAR_SINGLE	FP scalar ins. single-precision
[g]	FP_ARITH_INST_RETIRED:128B_PACKED_DOUBLE	FP 128b vector ins. (SSE) double-precision
[h]	FP_ARITH_INST_RETIRED:256B_PACKED_DOUBLE	FP 256b vector ins. (AVX/AVX2) double-precision
[i]	FP_ARITH_INST_RETIRED:512B_PACKED_DOUBLE	FP 512b vector ins. (AVX512) double-precision
gcc command line: -O3 -march=native -mtune=native -ftree-vectorize -funroll-loops		
icx command line: -O3 -xHost -vec -axCORE-AVX2,CORE-AVX512		

Table 4: The events set specified for the profiling with the PAPI library, and the compilation options (bottom framed box) used on the Leonardo platform.

Events [a] and [b] are tracked to check the overall efficiency of the code issued; in fact, in general, vector ops (especially AVX512) may be quite expensive in terms of cycles, and the CPU tends to lower the clock when intensively issuing vector ops. This, in turn, may result in a run-time larger than expected.

Events [c] and [d] are included to track the memory effects, if present.

The contrast of events [e] to [h] and [i] gives the vectorization fraction that is defined as the ratio of vector FP instructions over the total number of issued FP instructions.

The results obtained on Leonardo are shown in Fig.7.

The main findings can be summarised as follows:

1. The original pattern (AoS) leads to a very poor vectorization (Fig. 7a AoS entry of the leftmost histogram cluster), which matches with the findings of the preliminary analysis discussed in D2.1. However, Intel's compiler can achieve a very high vectorization, nearly 95%, in the case of contiguous memory (v2.icx.AoS). That is because in the v2 case, the loop runs explicitly over contiguous memory locations, which makes it easier for the compiler's analyzer to spot vectorization opportunities (however, GCC fails in that case, and its FP mix remains the same).
2. Using "small structures", i.e. keeping the approach of clustering the physical quantities in structures but reducing their size, and hence enlarging the number of single different structures, may help significantly in enhancing the vectorization.
3. The adoption of SoA approach does not lend per se a better vectorization nor a smaller run-time (only marginally so for all v2 cases and in v1.icx case). However, when the loop is over contiguous memory locations, and the compiler individuates the opportunity, i.e. case v2.icx.SoA, the gain in both optimization (which reaches $\sim 100\%$) and run time ($\sim 2\times$) is substantial.
4. Using the explicit intrinsic ops leads to a very high, virtually complete, vectorization (cases v*.AoSSi, v*.SoAi). At the same time, the run-time is larger by $\sim 60\%$ in all v1 cases for reasons that we are still inquiring. At odds, the usage of intrinsic allows us to run $2\times$ faster in all v*.*.SoAi cases. However, optimally using the intrinsic is generally difficult and has a very low portability since it would require specialized, carefully crafted versions for different platforms. We tested them for comparison, but we are not seriously considering adopting such approach in our code, also because it would require to convert all the data structures to SoA.

In conclusion, no single recipe seems to win the race for a better run time. That is tightly linked to our peculiar pattern that is very different from the all-to-all direct summation of the "true" Nbody codes. The strongest hints that we envision are

1. Having the neighbors stacked in a contiguous buffer seems to give a general possible advantage. While that is difficult to achieve for *all* the neighbors, we may consider the opportunity of developing a framework in which a thread fetches the neighbors' relevant data in half of a limited buffer while other threads process data on the other half of the same buffer.
2. Breaking the "large structures" into smaller ones may offer some advantage, which we can envision not only in calculation kernel but also in other points (for instance, tree building and walking). Moreover, in case we opt for a topology-aware framework with memory shared among MPI processes at the node level, that may be an advantage in terms of memory occupation.

Final note. For the completeness of the report, we highlight that while developing these kernels we have also developed some useful header files which offer a self-contained, simple and *compiler-agnostic* syntax for (i) defining the vectorization and unrolling pragmas and hints, that differ among the compilers; (ii) discovering the extension of the vector capabilities (i.e. SSE, AVX/2, AVX/512) and defining a vector size to be used throughout the code that does not explicitly contains that size; (iii) defining the vector extension for vector types, which also differ among the compiler.

The code is publicly available at this [Git URL](#) and will be further developed in collaboration with FORTH and Eviden partners.

3.1.2 "Aligning" the data structures at the domain decomposition

Eventually, breaking down the large data structures into more structures of smaller sizes would require some adaptation in the code.

In fact, as for now, the data particle layout is the following:

1. OpenGadget has six different types of particles that represent different matter types or matter phases: baryonic matter, either gas or stars, dark matter (three types, two of which are used in zoom-in simulations to provide large-scale average fields), and black holes. N , N_{gas} , N_{stars} and N_{bh} are integer variables that store the number of all, gas, stars and black-hole particles.
2. The structure **P** collects physical quantities that are common to all particle types (for instance, positions, mass, ID, ...); three structures collect the physical quantities peculiar to specific particle types: **SphP**, **MetP** and **BHP** refer to gas, star and black-holes particles.
3. The gas structure **SphP** is "aligned" with the structure **P**: the N_{gas} particles are always the first N_{gas} of the **P** array, so that $\text{P}[i]$ refers to **SphP** $[i]$ for $i < N_{\text{gas}}$.
4. Stars and Black Holes are not sorted in any way; the index j of the corresponding **P** $[j]$ structure of any star (or BH) i live in the range $N_{\text{gas}} \leq j < N$. The structure **P** and the structures **MetP** and **BHP** are cross-indexed.

At any moment when particle exchange is needed (for instance, at the domain decomposition), the four structures must be kept "aligned"; when a particle **P** $[i]$ is to be sent/received, also the corresponding structure **SphP** $[i]$ or **MetP**/**BHP** $[\text{cross-index}]$ must be received/sent.

3.1.3 A thread-safe library for Stellar Evolution

Currently, this alignment is directly enforced for the three additional types of structures. However, if a different approach is to be adopted, in which the number of structures grows consistently and, above all, new structures appear for new physical modules, that would quickly become a bad technical debt.

On this purpose, we have developed a framework that automatizes this alignment once the "links" between **P** and other structures are specified for every particle type.

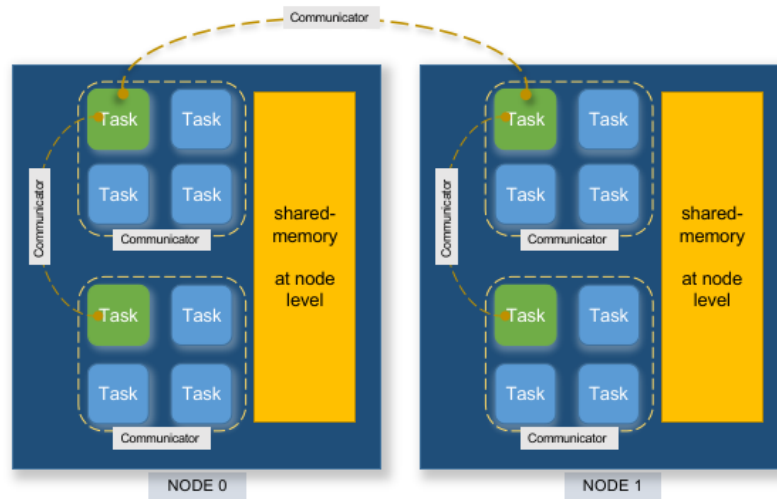


Figure 8: A sketch of topology awareness. Communications among two nodes ("0" and "1" in this example) are managed by two tasks, one per node, designated as "node masters". A "inter-node" communicator includes only node masters. Within each node, tasks may be either grouped in communicators specific to each NUMA node (with a designated master that exchanges data with other masters) or live in a unique Communicator and share a fraction of the node memory as MPI shared memory. Green tasks represent the "masters".

3.1.4 A framework for topology-awareness

While OpenGadget has very good adaptability in terms of Nodes/Cores occupancy because of its hybrid MPI+OpenMP nature, more efficiency could be gained by implementing explicit topology awareness and, in turn, the capability of using (i) shared-memory among MPI tasks in the same nodes, and (ii) RDMA (Remote Direct Memory Accesses) among different nodes. Furthermore, all the RDMA among two nodes could be managed by only two tasks, one per node, acting as "node master". This may greatly reduce the communication surface and foster the adoption of synchronization-free algorithms that may enhance parallelism.

In the framework of the SPACE project, we have developed and refined an easily adoptable framework to explore the topology and build a hierarchy of MPI communicators, as depicted in Fig.8.

Our code can either exploit the `MPI_COMM_SPLIT()` capabilities, which at the time of writing are becoming increasingly supported by major MPI implementation (which was not when we started the development), or explore the topology autonomously.

To test it, we have developed an implementation of Reduce, which exploits these capabilities (MPI shared memory within nodes and RDMA among nodes) without calling a single MPI communication routines and is 4× to 5× faster than both `MPI_Reduce` and `MPI_Ireduce` on Leonardo.

The code is available at [this link](#).

3.2 GPU offloading

As mentioned in D1.3, for the GPU porting of OpenGadget, the modules with the largest contribution to the total runtime have been chosen: the tree walk and the Barnes&Hut oct-tree.

In the following Sec.3.2.1, we detail the current status and performance of GPU offloading, whereas in Sec.3.2.2, we discuss our current work to develop a new approach.

3.2.1 Current status and performance

In the larger framework of the code, the tree walk is called at multiple stages as the tree conveys information about the spatial locality of the particles. This means that the computation of each physical process in which a particle impacts additional particles relies on the tree walk to determine the neighborhood of the active particle. Or, conversely, to determine the neighborhood that is affecting a target particle. Additionally, a specific algorithm, Barnes&Hut, is adopted for solving the gravitational interaction, which we detail in the following.

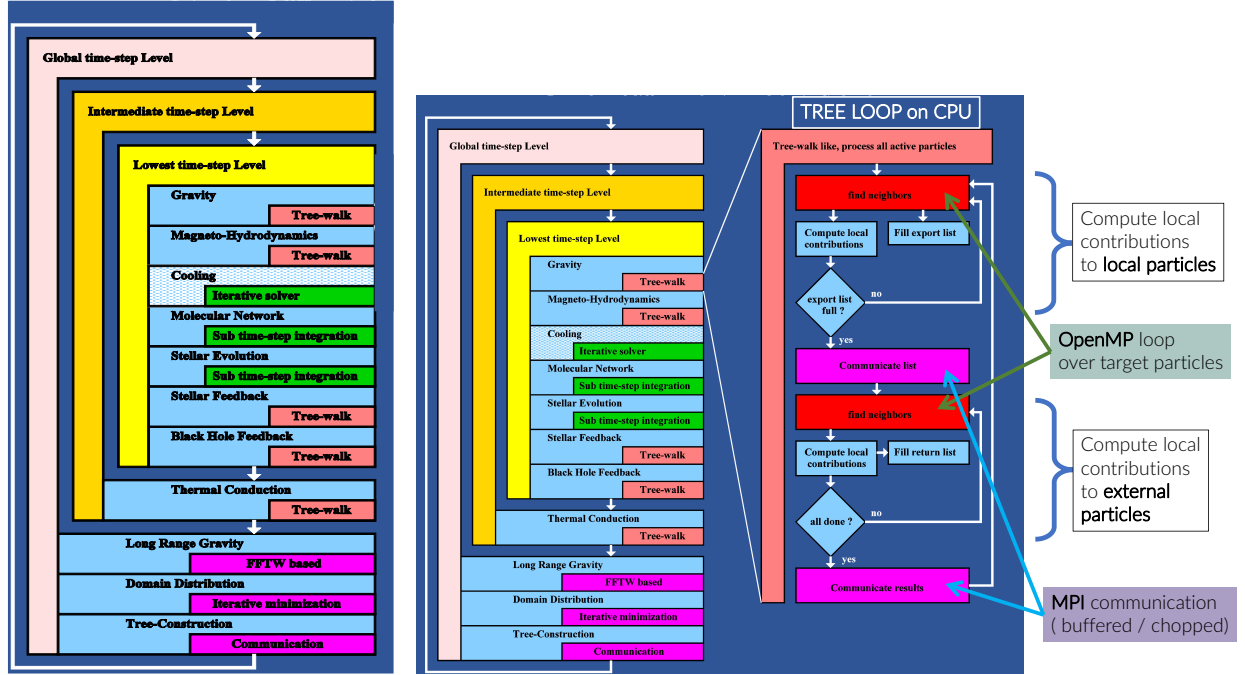


Figure 9: [**Left Panel**] Schematic view of the code’s workflow. The operational blocks on the left are grouped by the particles’ timesteps they refer to. Low timesteps levels are occupied by particles with small timesteps and higher levels by those with larger timesteps. Blocks with green tags represent routines that solve local particle-wise physical processes that do not involve inter-particle interactions. Blocks with orange tags represent points that rely on a tree walk. Magenta tags indicate sections that require MPI communications. [**Right Panel**]. Zoom-in on the tree walk pattern. The Tree walk phase is always split into two symmetric sub-phases, each of which involves a multi-threaded computation stage followed by a communication stage.

In the left panel of Fig. 9, a schematic view of the workflow for one timestep is shown. On the left, we report the major operational blocks, grouped by the particles’ timesteps they refer to. Low timesteps levels are occupied by particles with small timesteps and higher levels by those with larger timesteps. Blocks with green tags represent routines that solve local particle-wise physical processes that do not involve inter-particle interactions. Blocks with orange tags represent points that rely on a tree walk. Magenta tags indicate sections that require MPI communications. The grey boxes highlight the physical processes involved in the mentioned operation blocks. In the right panel of Fig. 9, we break down the tree-walk task in its sub-tasks. The tree walk itself is split into two phases. In the first phase, the algorithm searches for all neighboring particles of a specific target particle, which is the one for which the gravitational force has to be calculated. Actually, in this case, the search is performed with the Barnes&Hut algorithm; more details will be discussed when we expose recent developments.

While walking the tree, the list of tree nodes that are “in the neighborhood” but reside on other nodes’ MPI task domain is created. At the end of this first phase, every MPI task communicates a list of “requests” to the other MPI tasks whose domain intersects the neighborhood of some of its active particles. Every MPI task receives a list of target positions from some other MPI task for which it has to communicate the force contribution of its particles. Note that when the FFT-based Particle-Mesh algorithm is not used for long-range forces, this exchange is all-to-all since Gravity is a long-range force, and every MPI task will contribute to the force on every other MPI task’s domain. After the communication, a second iteration of the loop starts in the second phase, perfectly symmetrical with the previous one, in which every task is calculating the contribution of its own particles to the particles of other tasks. The resulting contribution to force is then communicated back to the MPI tasks that requested it. During this second communication round, every MPI task sums up the contributions received by its particles to their force.

The current GPU implementation is based on porting the computationally heavy part of the tree-walk to the GPU while keeping the global structure. Consequently, only the local contributions are computed on the device, while the communication and all other aspects remain on the host. For this, a new data structure is created, which only contains the information about the available particles, which is then communicated to the device and returned after the calculation is finished.

This approach allows us to implement the GPU tree walk by simply annotating CPU functions. The benefit of this approach is that the same code is used on the host and on the device, so updates and bug fixes on the tree walk itself are immediately available in both code versions.

Summarising (see D1.3 for more extensive discussion), the following modules have been ported onto GPU in the current version:

- **Gravity calculation with tree**
- **SPH Hydrodynamics**
- **Thermal conduction**

NOTE: the GPU implementation has started before the project SPACE, and is due to the work of Dr. Antonio Ragagnin and LMU's staff. An important part of the work has been conducted during the first 18 months, by LMU within the SPACE project in collaboration with Dr. Ragagnin. Namely, several optimizations on the Gravity part (mostly the overlapping of computation and communication) and the porting of the hydrodynamics loop.

Scaling of the current GPU offloading

The first aspect to consider is the speed-up of the GPU implementation compared to the CPU runs. We are able to report this for an entire big run (2048^3 particles) from a recent simulation run by an INAF collaborator, Dr. Tiago Batalha de Castro, working at INAF, in the frame of another science-focused project. The plot in Fig. 10 shows this ratio for a gravity-only simulation. As explained in the figure caption, most of the computational time is spent in the epoch $a \gtrsim 0.2$ (where a , reported in the x -axis, is the cosmic expansion factor, a common parameter to measure the time in Cosmology). In practical terms, what changes from very pristine epochs (small a values, on the left) to evolved situations (larger values of a , on the right) is the formation of cosmic structures (galaxies and clusters of galaxies) that collapse under the action of the Gravitational force, coalesce in bigger ones, and create deeper and deeper potential wells.

In other words, in the beginning, the density contrasts are very tiny, and the distribution of the particles is very homogeneous. Later, the evolution of the Universe leads to extreme density contrasts (up to 7 to 9 orders of magnitude in state-of-the-art simulations) and largely non-uniform distribution of particles that are, instead, very clustered. In turn, this causes a much deeper tree structure.

For reasons that will be clarified afterwards (see Sec. 3.2.2), a more clustered configuration and a deeper tree reflect in a much more limited efficiency of the Barnes&Hut algorithm, which ends in the loss of general efficiency evident in Fig. 10.

As we have detailed in D1.3, 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).

While in D1.3, we focused on the overall scalability, here we report a closer inspection of the behaviour of the 2048^3 box as shown in Fig. 11 where the analysis of the different code's components is reported for the gravity only case. We show the parallel efficiency and timing breakdown on the left and right panels, respectively. In the gravity-only case, the driver of the efficiency loss is the "treegrav" component, i.e. the calculation of short-range Gravity with the Barnes&Hut algorithm. In the Hydrodynamics case (not shown for brevity), while the gravity tree still loses efficiency steadily, the main driver of the efficiency loss is the hydrodynamical section (the "sph" component) that requires most of the computational effort.

In the next section, we discuss our current development activity which aims to solve the issues highlighted in the previous discussion.

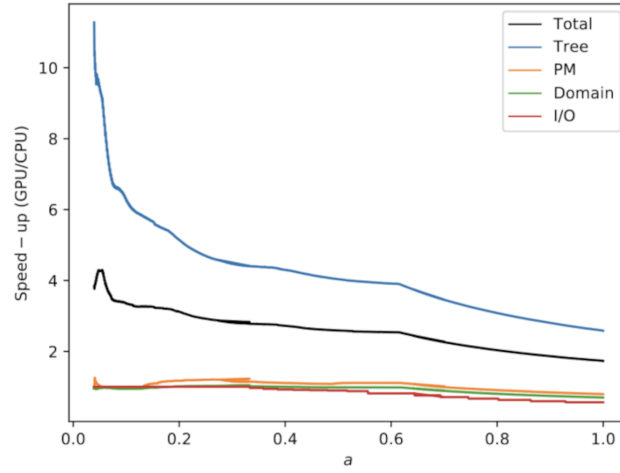


Figure 10: The speed-up of GPU over CPU implementation for an entire cosmological box (2048^3) with Gravity only. This plot is courtesy of **Dr. Tiago Batalha de Castro (INAF)** who ran the simulation on Leonardo using 32 nodes in the frame of another science-focused project.

On the x -axis there is the expansion factor a of the Universe: smaller values correspond to pristine times, whereas values closer to 1 correspond to today. The conversion in time is increasingly non-linear towards small a : the reader should consider that $\sim 0.57\%$ and $\sim 0.76\%$ of the time elapses from $a = 0.5$ and $a = 0.3333$ to $a = 1$, respectively. Only $\sim 4\%$ of the time elapses from $a \gtrsim 0$ and $a = 0.1$. As for the computational time, the proportion is even more severe since from $a \gtrsim 0.1$, the formation of substructures leads to deeper and deeper potential well, which reflects in larger accelerations and, hence, the need for smaller timesteps. The effective speed-up of the simulation is then the one at $a = 1$, which is ~ 3 .

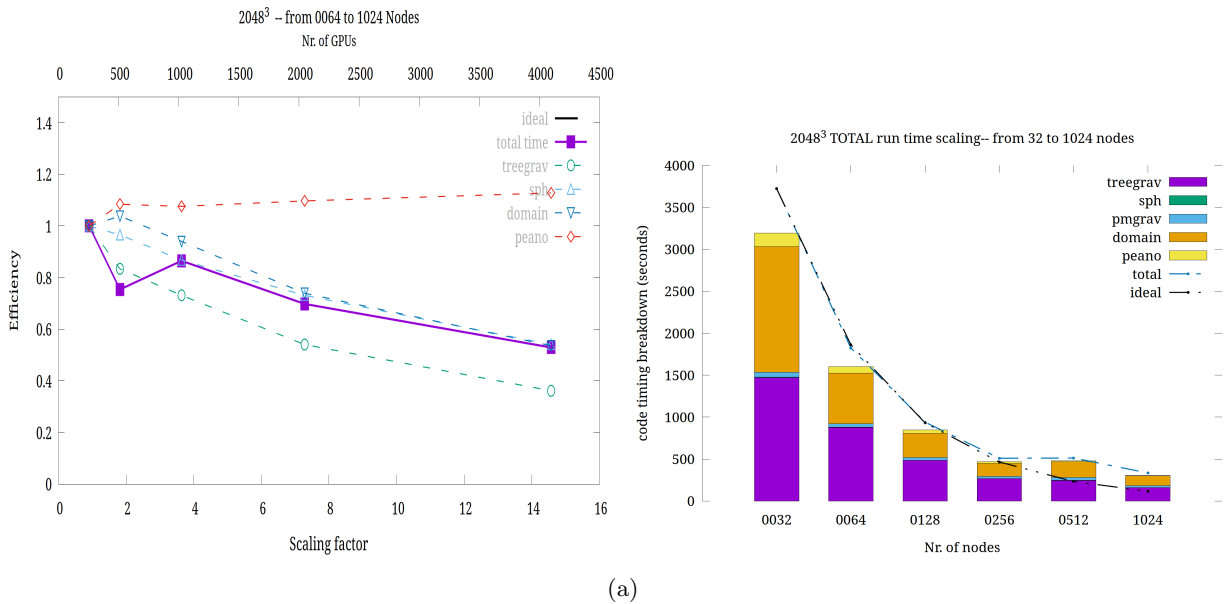


Figure 11: Details of the 2048^3 boxes, gravity only. The *parallel efficiency* (**left**) and the *timing breakdown* (**right**) of the gravity-only case.

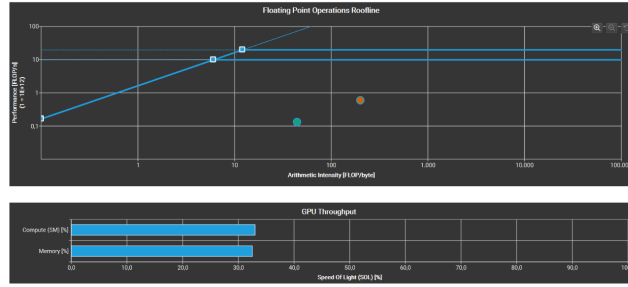


Figure 12: A screenshot from the NVIDIA profiler with the roofline model of the gravity tree routine that implements the Barnes&Hut algorithm. The blue and red points refer to single- and double-precision floating-point operations. Maximum theoretical peaks for double- and single-precision are the lower and higher horizontal lines. The bottom panel shows the relative efficiency in memory transfer and calculations. Evidently, the current implementation of the Barnes&Hut is severely under-performing due to algorithmic limits.

3.2.2 A new strategy for the GPU-offloading of the Gravity tree calculation

The direct solution to the problem of estimating the Gravitational force on a given particle i due to a distribution of other particles is provided by the Newtonian force $F_i \propto \sum_{j=0}^{j < N} m_j / r_{ij}$. Many-body codes, such as OpenGadget, solve the problem of the gravitational interaction among particles that are a sample of a density distribution and not *true* bodies and aim to simulate systems of tens or hundreds of billions of particles (cosmological simulations, clusters of galaxies or galaxies, to name some). As such, approximate solutions, with a second- or third- order error with respect to the direct summation, are adopted to achieve a better scaling than the $O(N^2)$. In a many-body code, such as OpenGadget, we use the Barnes & Hut algorithm to estimate the force F_i , reducing the algorithmic complexity to $O(N \log N)$ (compared to $O(N^2)$ via direct summation): relying on the tree structure to distinguish between "close particles" that are summed directly as in the former equation, and "clusters of distant particles" that are considered as an effective unique particle whose contribution is summed to F_i .

In short, while the tree is walked, the angle of view θ under which it is seen from the position of particle i is calculated for every node. All the nodes for which $\theta > \theta_{BH}$ are "open", i.e. all their children nodes are considered for the same criterion. Conversely, the nodes for which the opposite case $\theta \leq \theta_{BH}$ holds are considered unique particles (we are omitting the unnecessary details about how the effective mass and position are estimated).

The reader may have a clearer idea of the algorithm from Fig. 13, where we use a 2D representation for the sake of simplicity. From a set of particles in space (top left panel), a tree representation is built (bottom panel). Nodes with children are represented as ivory circles, leaves (i.e. single particles) as green smaller circles and empty nodes as pale blue dots. To help follow the order of nodes in the tree traversal and building, we name the children of a node N as $[NA, NB, NC, ND]$, A and D being the north- and southwest children, respectively. Hence, the root's children are $[A, B, C, D]$, the A's children are $[AA, AB, AC, AD]$, and so forth.

If the particle P at the northeast of node A is our target, the top-right panel of Fig. 13, in which shaded cones represent the "cones-of-view" from P, depicts approximately some nodes that will be treated as unique particles (those under the green cones). Nodes that are too wide, those under the red cones, will be instead "open", and their children will be considered individually. Particles highlighted with yellow circles are among those whose contributions will be added in direct summation.

From Fig. 13, it can be appreciated how the result of the tree walk under the Barnes&Hut is unique for every particle. If two threads were processing the particle P and its sibling in the southeast child of node A, respectively, which nodes would be open or not and which particles would be directly summed up would be different for the two threads. If they were two threads in the same warp, that would lead to excessive divergence and locking.

The tree walk is, by its nature, unique for every target particle, and the Barnes&Hut requires many conditional evaluations to decide which nodes must be open or considered as a whole. The Barnes & Hut algorithm, in its "classical" formulation, although reducing the algorithmic complexity, proves to be inefficient on modern architectures and particularly on GPUs since it leads to a large amount of thread divergence and locking. Its under-performance is visible in Fig. 12, which reports a screenshot from NVIDIA's nsys profile of the routine that implements the tree-based Gravity calculation.

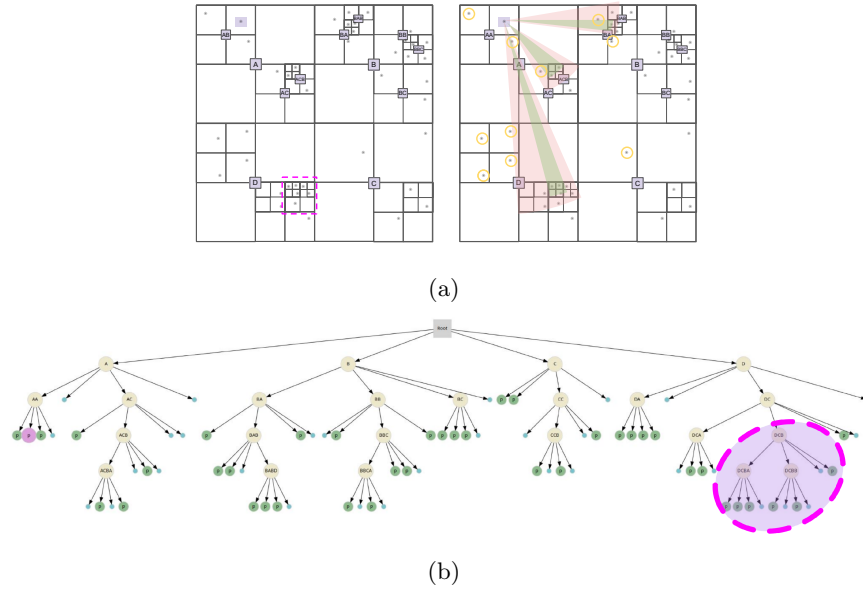


Figure 13: An example of 2D particle distribution (**upper row, left panel**), and of the correspondent quad-tree representation (**bottom panel**).

In the tree representation, the empty nodes are pale-azure dots, the particles (i.e. the leaves) are green circles, and the nodes are ivory circles. To help follow how the tree is built, the children nodes of node N have been named as $\{NA, NB, NC, ND\}$ with A and D being the northwest and southwest nodes, respectively.

To further exemplify, the region framed in magenta in upper left panel corresponds to the same region framed in the bottom panel. See also Sec. 3.2.2. The target particle used in the discussion is the child of node AA , with violet background.

In the **upper row, right panel**, we show some examples of nodes too wide to be open (those with the red angle of view) and of nodes small enough to be open (those with the green angle of view). Particles highlighted with a yellow contour are some of those that will be added in direct summation.

Our **new strategy** is based upon the following strategies:

1. Grouping particles; performing the tree walk & evaluation from the same position for a group of spatially close particles while assigning same-group particles to same-warp threads allows (i) to shrink the number of tree walks (as much as the "grouping factor") and (ii) to synchronize the ops for threads on the same warp (when assigning a target particle per thread). Equal or larger accuracy can be maintained with a more constrained opening criterion. As an example of using the already existing tree representation to individuate clusters of close particles, we address the reader to the node framed in magenta in the upper-left panel of Fig. 13.

In order to unify the tree walk of those particles, we could consider performing a unique tree walk for each of them, *having its origin as their centre of mass*. In this way, the threads to which they were assigned would be perfectly synchronized without, or with very little, divergence or locking. A node will be eligible to be a target node if (i) it contains a number of target particles $N_{\text{active}} > N_{\text{active}}^{\min}$ and (ii) its side length is smaller than a threshold value $L_{\text{node}} < L_{\text{node}}^{\max}$. $N_{\text{active}}^{\min}$ and $L_{\text{node}}^{\max}$ will be two additional code parameters.

2. Avoiding the tree walk in an "inner radius"; inside a given inner radius, we avoid evaluating nodes and immediately sum up, with direct summation, all the particles of nodes that are encompassed within that radius. That is feasible thanks to the memory arrangement of the tree nodes and the indexed traversal already implemented in OpenGadget. The upper panel of Fig. 14 shows that the tree nodes are saved in memory following the depth-first order. In other words, the branches of the tree are always as close as possible in memory.

Furthermore, once the tree is built, it is walked bottom-up for the first time, setting the indexes for a complete traversal without actual walking (defining "talking" as following the path to a node from the root node). This way, every particle points to the next particle, be it a sibling (i.e. a child of the same father node) or a particle on some upper level of the same branch or of a different branch. That is shown in the bottom panel of Fig. 14. Still referring to the tree reported in Fig. 13, from the root node we can directly descend to the first particle, AA.a (note: here we identify a node by its complete ancestry, separating the levels with a dot and naming the children of a node as A, B, C, D , as explained before. Small case letters refer to particles). From AA.a then we jump to AA.b, then to AA.c. From there, we should skip AB, which is empty. In fact, AA.c's pointer goes to AC.a, and from there to ACBA.b, ACBA.d. At this point, the branch A has been exhausted, and there is a "big" jump to the branch B , specifically to BA.a. And so on, without the need of travelling up and down the chain of father nodes.

Hence, let's imagine that we perform a collective search for the particles living under the node DCB (the magenta-framed in the upper-left panel of Fig. 13). Let's also imagine that the node ACB is included in our search radius so that we do not want to further verify any of its children nodes. Following the indexed traversal, we may jump right to ACB.b and then traverse directly to ACB.d and ACB.d. After that, we would jump to BA.a, which, however, is outside the boundaries of ACB, and then we could stop. The same could be unrolled for the node DA , which contains four particles and so on.

It is worth noting that keeping the particles partitioned in memory (no precise ordering is needed) by their branch membership will keep the particles close in memory; the smaller the branch segment considered, the closer they will be. Referring to Fig. 13, particles living under the volume of the node A will be closer among them than to those living under the node D volume. In the same way, particles under node ACB will be even closer and so on.

As a first trial, we opted not to keep the particle structures close in memory. The actual loop that updates particles' force value gets through the list of active particles and not the list of particles (note: we remind the reader that the particles are assigned with individual time-steps within a hierarchy of power-of-two decomposition of the timeline. Particles that are subject to more intense accelerations are integrated more often. Hence, at every code step, only a subset of particles is to be updated). Hence, we opted for

- traversing the list of active particles, propagating the information through the hierarchy of tree nodes so that every node knows how many active particles it contains;
- individuating the target nodes, i.e. the group of particles that will be treated collectively, following the criteria explained above;

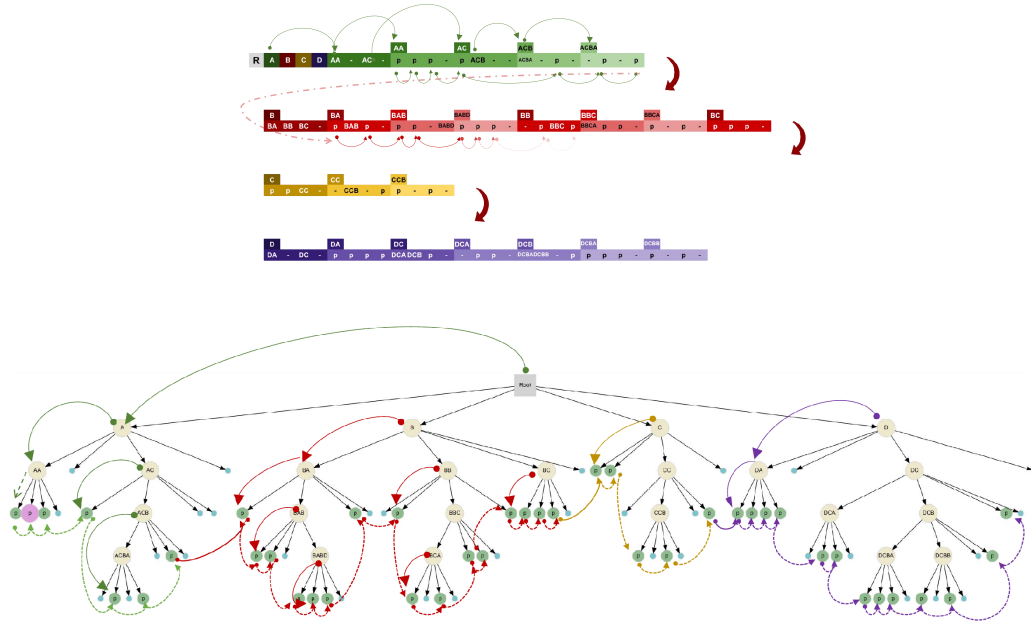


Figure 14: In the **upper panel** we sketch the organization of the Tree structure in memory for the tree shown in Fig. 13. Colors refer to the same main branch, A, B, C, D , while they become fainter with the depth of the tree. Red arrows remind the reader that all the nodes are subsequent in memory; the different sections have been split into different lines to enhance the readability.

In the **Bottom panel** we reveal the implementation of the indexed traversal; from a given node a complete tree traversal is possible without having to actually walk from the root node. Some of the same links are shown in the upper panel. See the text for more details.

- partitioning the list of active particles so that the active particles that belong to the same target node are subsequent in the list, which is a crucial step. After that, the loop which offloads the new routine for the modified Barnes & Hut assigns subsequent active particles to threads in the same warp.

To assess the potential of the approach drafted above, we have developed a kernel that reproduces on GPU a pattern similar to Barnes&Hut's.

In this kernel an array of N_p particles is generated with random positions and masses, of which N_A will calculate the force due to the contribution of their neighbors. The number $N_{\text{NGB},i}$ of neighbors for every particle is randomly determined as well in the range $[1000 : 5000]$.

Each of the N_p particles is then labelled as belonging to a "target node", defined by a unique integer id. We randomly generate this labelling, ensuring that every "target node" is linked to a maximum fraction f_p of the total particles, which is a parameter.

A list of the indexes of the active particles is then built. The "force calculation" amounts to call, for every active particle, a function `walk_particles()` which executes a set of calculations, among which the accumulation of $\propto 1/r^2$ contributions, that depends on an initial integer "seed" (we omit the fine details for the sake of brevity).

The GPU offloading consists of the offloading of the relevant data to the GPU and in executing the `walk_particles()` routine on gpu.

We can run in three different "modes", which represent different offloading strategies:

- v0 - Offloading the loop over the list of active particles, seeding the operations in `walk_particles()` with the particle's index.
- v1 - Sorting the list of active particles by their target node number, and seeding `walk_particles()` with that same number.
- v2 - As in the previous case, but the offload is split in two levels of parallelism; an outer loop over the unique target nodes is offloaded with `gang worker` parallelism, and hence the loop over all the particles is parallelized at `vector` level.

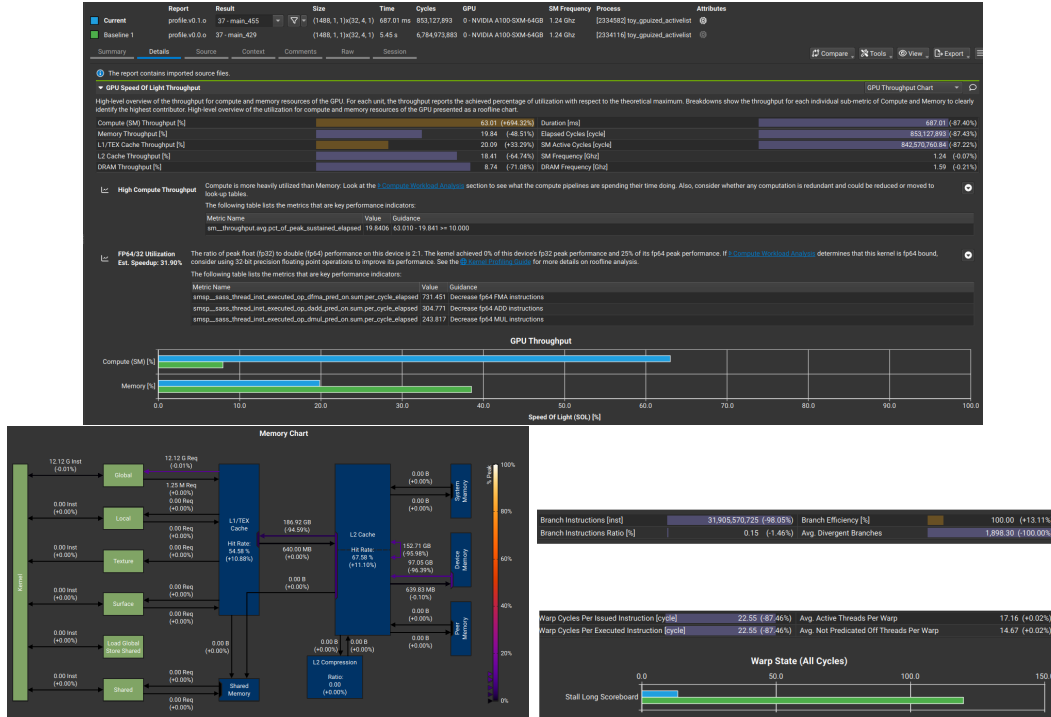


Figure 15: NVIDIA's NCU screenshot. **Top Panel:** performance metrics, **Bottom left Panel:** memory characteristics, **Bottom Right Panel:** thread divergence and stall. Comparison of experimental kernel run in "OpenGadget-like" mode and with the new re-arrangement of active particles. Note (i) the much smaller kernel time (0.68s vs 5.5s), (ii) the much larger compute throughput (63%SoL vs 8%SoL), (iii) that the v1 kernel is compute-bound whereas v0 is memory-bound and that (iv) that the thread divergence is much smaller. This NVIDIA's NCU screenshot is from comparison mode, with the baseline set to v0.

We have run this kernel (available at this [GitHub link](#)) on a Leonardo's node collecting the following insights, which confirm that our strategy has an interesting potential. First, versions v1, v2 are 5× faster than version v0, and that descends purely from the different data organization in memory and from the fact that the operations issued per particles in the same warp are synchronized when those particles belong to the same target node.

Fig.15 summarizes some key metrics for a case with 100M particles, of which 20M are active. We report snapshots from the NVIDIA's ncu profiler in comparison mode; the v0 is the baseline, and the v1 is shown. In the top panel, the performance metrics show that the v1 kernel is much faster (~ 0.69 s vs ~ 5.8 s) and delivers a much larger throughput ($\sim 63\%$ vs $\sim 8\%$ of maximum throughput). The reasons for it are, at least partially, explained in the bottom panels. The left panel shows that the v1's memory movements are $\sim 5\%$ of v0's: 190GB from L2 to L1/TEX, 100GB from host to device instead of 3.5TB and 2.7TB, respectively. Since the real amount of data processed is the same, the difference is entirely due to unnecessary and non-coalesced memory movements (more than 70%) in v0. Moreover, the thread divergence is much smaller and its impact is dramatically lower in v1. In fact, (i) the branch efficiency results to be 100%, as expected, instead of 87% and the average divergent branched just 2K instead of 230M; (ii) the branch instructions are 31G instead of 1630G; (iii) the warp cycles per issued instruction 22 vs 140.

At the moment of writing, we are acting on the mini-app, where we have already modified the tree-building to natively incorporate the concept of "target node", to partition the particles in memory by their belonging to a top-tree node (note: top-tree nodes coincide with the very first levels of the tree, used to decide the domain decomposition) and then by their belonging to target nodes (then enhancing the correspondence between spatial- and memory contiguity). We also have implemented the new tree walking and currently we are refining the coupling with the gravity force calculation to ensure its correctness and error estimate.

3.3 CPU updates

While the GPU offloading and restructuring of the building blocks of the code are highly important for the development of the code towards exascale, we also work on improving the CPU implementation in parallel.

We have identified two cases for further improvement. One is the optimization of various loops over all particles. Here we were able to restructure the loop over all particles with a loop over gas or star particles only. In this process, we have been able to reduce the time required for the small time steps by a factor of ~ 2 , which can be seen in Figure 16, which reduces the time per time step overall by $\sim 25\%$.

The second improvement is related to a bottleneck in the computation of the gradients. In this case, a more precise memory allocation combined with a reduced zeroing of the related matrices provided an additional improvement. These additional improvements can also be seen in Figure 16. In total, these improvements speed up the calculation of the smallest time bins by up to a factor of ~ 5 .

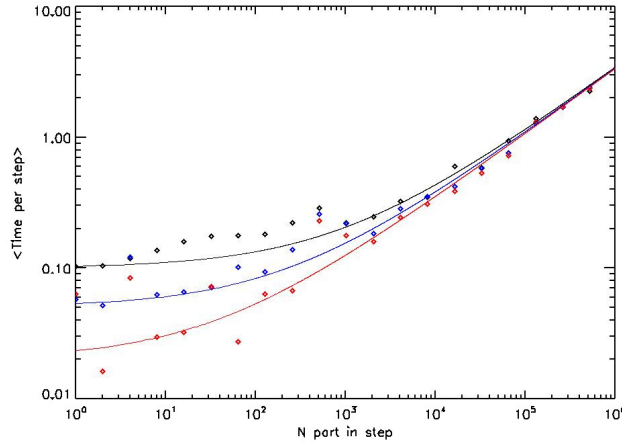


Figure 16: Comparison of the required time per time step at different numbers of particles in each time bin for the old (black) version. The one with the improved loops (blue) and the first updates on the gradient calculation (red). It can be seen that the time required for the smallest time steps is reduced by a factor of ~ 2 and by the optimized loops and even further to a factor of ~ 5 with the updated gradient calculation.

Since the computation of the small time steps in most cases takes about half of the total simulation time, these updates improve the capabilities of the code significantly.

3.4 Conclusions

At present a global polishing of the code has been conducted to enhance modularity and readability, in the view of the public release.

Algorithmic and coding optimization have brought up to $2\times$ speed-up in the "small time-steps", i.e., in the steps where a very small number of particles are updated. Due to the hierarchical time-stepping strategy of OpenGadget, about half of the time is spent in small steps. As such, we expect an overall beneficial impact that we will quantify in forthcoming tests.

In this first period we have also collected many building blocks necessary to the refactoring of core components of the code, which will be beneficial in the next phase of the activity.

As detailed in D1.4, the CI/CD framework is very complete and allows for comprehensive testing of correctness and validation of every merge request of feature branches.

	CPU optimization		GPU optimization		Scalability optimization		
Kernel name	focus	performance gain [%]	focus	performance gain [%]	focus	number of nodes/ GPUs/ CPU cores	Parallel efficiency gain [%]
Density & SPH	Loop Restructuring	~ 15%	-	tested on CPU only	same than CPU	-	to be established
Gravity, Density & SPH	Change of Data Str. & loop restructuring	potential: ~ 10% to 2×	same than CPU	tested on CPU only	<i>same than CPU</i>	-	to be established
Tree Walk	-	-	Reduction of thread divergence	<i>to be established</i> potential: 3× to 5×	-	-	-
Tree Build	Partitioning of particles by spatial contiguity & fat leaves	to be established	same than CPU	to be established	-	-	-
NUMA awareness	Exploiting MPI memory windows & Comm-free algorithms	to be established	same than CPU	<i>to be established</i>	same than cpu	-	to be established

Table 5: Summary of the optimization work performed on OpenGadget. As mentioned at the beginning of this section, the activity mainly focused on polishing the main code and developing building blocks to be tested in the mini-app.

4 iPIC3D

iPIC3D [6] is a particle-in-cell (PIC) code developed to study plasma dynamics at the kinetic scale, where the individual (macro)particles of a plasma are evolved in a Lagrangian framework, whereas the moments (electric current, density, etc.) and the electric and magnetic fields are tracked on an Eulerian grid. It solves the Maxwell-Vlasov equation self-consistently using the implicit moment method. The three main kernels of iPIC3D are (a) Particle Mover, (b) Moment Gatherer, and (c) Field Solver.

Owing to the “implicitness” of the method involved, unresolved scales do not result in any numerical instabilities, as opposed to explicit PIC methods, where one is restricted to the Courant-Friedrich-Lewy constraint. This allows us to choose time-step sizes and grid sizes that can be 10 to 100 times larger than those used in traditional PIC codes.

We have switched to a new, publicly available version of the code, iPIC3D-CPU-SPACE-CoE (<https://github.com/Pranab-JD/iPIC3D-CPU-SPACE-CoE>), that is primarily developed by Prof Markidis and his team at KTH Sweden. Significant improvements have been made in the newer version of the code, which includes non-blocking asynchronous communication across the different MPI subdomains, reducing data dependence for writing data to files, and in situ visualization with **Catalyst**. We duly note that this CPU version has been highly optimized in terms of computational performance and future developments would only include inclusion of new mathematical algorithms. In Table 6, we illustrate the performance enhancement of the new version of iPIC3D in terms of speed-up. A one-to-one comparison of MPI-only parallelization yields a speed-up of a factor of ~ 5 . This speed-up is even further enhanced should one consider hybrid OpenMP+MPI parallelization.

For the first time, we have a fully-functioning, GPU version of the code - iPIC3D-GPU-SPACE-CoE (<https://github.com/Pranab-JD/iPIC3D-GPU-SPACE-CoE>) that has also been primarily developed by Prof Markidis and his team. The particle mover and moment gatherer modules run on the GPU whereas the field solver runs on the CPU. This necessitates data communication from the GPU to the CPU and back at every time step. Preliminary tests show that porting the field solver to the GPU does not offer any significant benefits. This GPU code is written using CUDA and is supported on NVIDIA GPUs.

Kernel	iPic3D (old)	iPIC3D (new) MPI only	speed-up MPI only	iPIC3D (new) MPI + OpenMP	speed-up MPI + OpenMP
Particle Mover	1166 s	301 s	3.8	31 s	37
Moment Gatherer	479 s	24 s	20	11 s	44
Field Solver	351 s	65 s	5.7	18 s	20
Total	2067 s	393 s	5.3	61 s	33

Table 6: Performance comparison of the old and new versions of iPIC3D for 3D magnetic reconnection. The speed-up is defined as the ratio of runtime of the old version to the code to the new one.

4.1 Asynchronous Boundary

The previous version of iPIC3D-CPU-SPACE-CoE used blocking synchronous communication, which necessitated multiple `MPI_Barrier` calls to ensure that the positions and velocities of the particles that move across the compute nodes, end up the correct MPI subdomain. This was one of the main findings of the POP analysis of iPIC3D (https://code.it4i.cz/space_coe/iPic3D and <https://github.com/CmPA/iPic3D>), which was carried out by Radim Vavrik at IT4I. The main findings of the POP analysis were the following,

- Excessive use of `MPI_Barrier` - might cause unnecessary serialization (Fig. 17),
- Sequences of many small processor to processor messages.

The subsequent recommendations were

- Identify data dependencies in communication phases,
- Fuse small MPI messages – lower latency, higher transfer rate,
- Reduce number of `MPI_Barrier` – less synchronization.

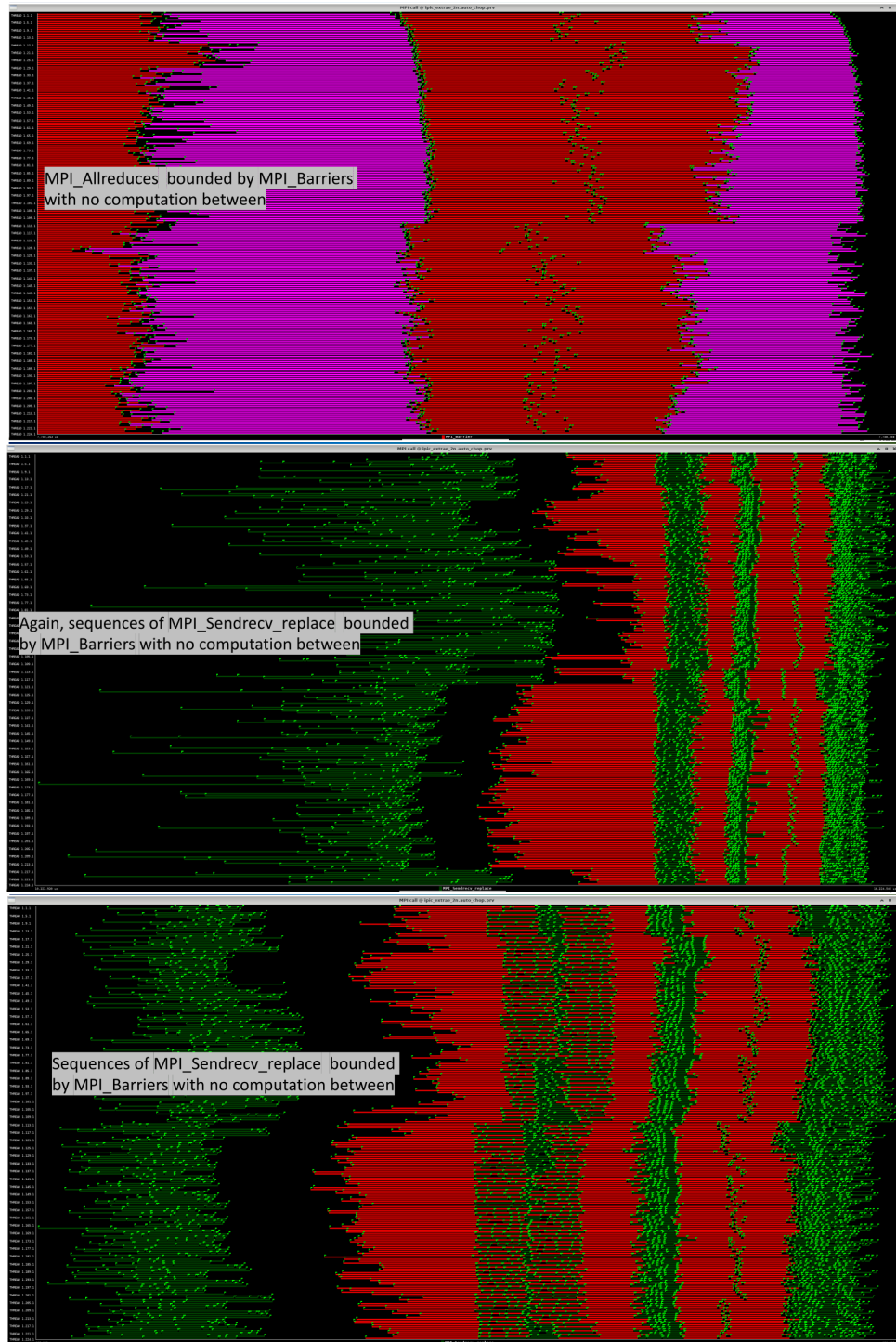


Figure 17: Figure shows the runtimes of MPI_Barrier (red lines) bounded by MPI_Allreduce (pink lines, top) and MPI_Sendrecv_replace (green lines, middle and bottom). It is evident that a large portion of the code runtime is wasted, i.e., no computation is performed, whilst waiting for the data to be transferred using blocking communication. These plots were obtained during the POP analysis.

The new version of the code iPIC3D-CPU-SPACE-CoE uses asynchronous nonblocking communication to communicate data between computing nodes in distributed memory systems, the functions of which are defined in `Com3DNonblk.cpp`. Similar performance improvements have been made in PLUTO, the details of which have been described in great detail in Sec. 2. Implementing nonblocking communication essentially involves switching from `MPI_Sendrecv` to `MPI_Isend` and `MPI_Irecv`, followed by `MPI_Waitall` to ensure that data transfer has been completed. The primary advantage of nonblocking communication is that the send and receive commands are initiated asynchronously, which implies that the program can continue with computation and communication, simultaneously. If the amount of data transfer is large, this can effectively hide the latency of the code. This can be particularly effective in 3D cases where the data have to be transferred across to 26 neighboring cells. A schematic illustration of synchronous and asynchronous communication is presented in Figs. 1 and 2, respectively.

4.2 OpenMP + MPI

One of the main improvements in the new version of the code is the usage of hybrid OpenMP and MPI implementation for parallelization. This has several benefits:

- This allows for both intra-node and inter-node parallelism.
- This allows for fewer MPI processes per node, thereby reducing communication between the different processes and overall memory usage. This reduced communication can also improve the scalability of the code.
- OpenMP threads within a node can share large data structures, avoiding the memory replication required by MPI processes.

In iPIC3D-CPU-SPACE-CoE, the directives `#pragma omp parallel` and `#pragma omp for` have been used in the Particle Mover and Moment Gatherer modules of the code. To compute the moments, these directives create parallel regions for moment computations and distribute the iterations of loops among the different threads to accumulate the moments (density, current, and pressure). As one could potentially simulate several tens of millions of particles, and each particle contributes to the moments, OpenMP can effectively reduce the runtime by parallelising using shared-memory parallelization. `#pragma omp critical` has been used to ensure that data updates to arrays or reduction operations are performed in a safe way, i.e., avoiding race conditions.

Similarly, `#pragma omp parallel` and `#pragma omp for` have been used to iterate over the particles to update their positions and velocities. This enables multiple particles to be processed concurrently. `#pragma omp simd` has been used to hint the compiler to vectorise and the directive `#pragma omp master` directive ensures that the initialization is executed by only one thread.

The application of `#pragma omp` is crucial for obtaining high efficiencies in scaling when the number of particles is large, which is typical of most realistic PIC simulations. In Table 6, we compare the performance of MPI-only parallelization (40 nodes, 100 tasks per node, 1 CPU per task) with OpenMP + MPI parallelization (80 nodes, 50 tasks per node, 2 CPUs per task). This hybrid approach leverages the fact the data stored in a node can be shared across the different CPUs in a node, thereby reducing the need for data transfer across the nodes. Table 7 presents a summary of the overall improvements in iPIC3D-CPU-SPACE-CoE over the older version of the code.

Kernel	CPU optimization	Scalability optimization	Overall performance gain
Particle Mover	OpenMP + MPI	Asynchronous Boundary	> factor of 30
Moment Gatherer	OpenMP + MPI	Asynchronous Boundary	> factor of 30
Field Solver	OpenMP + MPI	Asynchronous Boundary	~ factor of 20

Table 7: Summary of the optimizations for iPIC3D.

4.3 Conclusions

iPIC3D-CPU-SPACE-CoE has been highly optimized for computational performance and is ready for production. We obtained over 90% efficiency on up to 16000 cores for strong and weak scaling on multiple EuroHPC

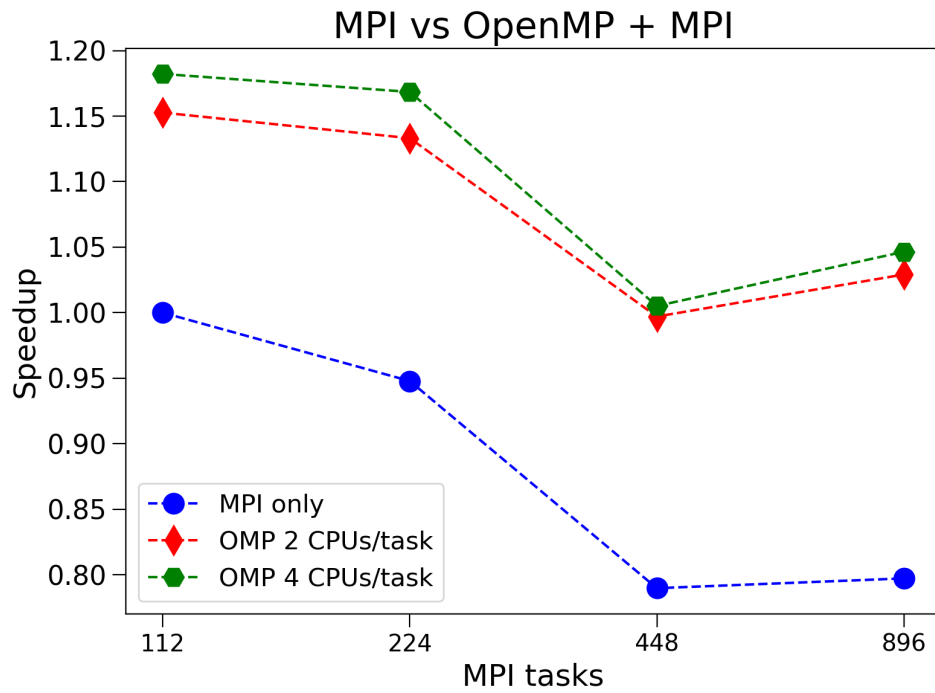


Figure 18: Figure illustrates the speed-up obtained with shared memory parallelization using OpenMP over using MPI-only parallelism on Leonardo DCGP. All data points are normalised to the runtime for 112 tasks using MPI-only parallelism. One can clearly see that we get $\sim 20\%$ improvement in performance whilst using OpenMP.

JU clusters (see Deliverable 1.4). Future developments will include new mathematical algorithms such as the energy-conserving semi-implicit method (ECSIM, [7]) and relativistic semi-implicit method (RelSIM, [8]).

We have an ongoing POP analysis of iPIC3D-GPU-SPACE-CoE, which is currently under development, to better understand its current status and potential scope for improvement. We will further develop the GPU version of the code based on the results of this POP analysis and make it ready for production.

5 RAMSES

RAMSES [9, 10] is an Adaptive-Mesh-Refinement (AMR) code written in Fortran 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.

5.1 Summary of the optimization work for the entire code

The work to optimize RAMSES for the pre exascale EuroHPC systems proceeds along two main paths. As a first objective, we are increasing the performance and scaling on *CPU* partitions by

- adding OpenMP in addition to the existing MPI paradigm,
- optimizing vectorization (work done in collaboration with Eviden).

We completed the work on the first kernel identified as an optimization target: the hydrodynamical solver (see deliverable 1.2). The performance is measured using the Sedov test case on a uniform grid (as identified in deliverable 1.1). The implementation details and results are presented in the next sections.

Secondly, we are porting the solvers of RAMSES into the new Shamrock framework, which is designed for GPUs. We finished porting the hydro solver (kernel 1), and obtained a weak scaling efficiency of 84% on 512 GPUs when using 128^3 cells per GPU. We will not describe the implementation in the current deliverable, as the paper presenting the framework is under review and has not yet been accepted for publication.

5.2 CPU optimization with OpenMP

Today, modern computing CPU architectures embed more and more cores. A common way to exploit such architectures efficiently is to combine the MPI standard to enable the use of multiple computational nodes, with a multithreading paradigm such as OpenMP to better exploit intra-node parallelism.

The MPI parallelism can be qualified as distributed parallelism. The program is launched with separate processes, each with its own memory space. This enforces the developers of MPI applications to implement explicitly the required communications between the processes, depending on the algorithm's needs. In contrast, multi-threading enables the sharing of the same memory space between different threads, also known as lightweight processes. It is then the responsibility of the programmer to ensure the data is well declared as shared or private to threads and to avoid race conditions in order to conform with the algorithms.

Multithreading can be of great help in improving the scalability of simulation codes. In RAMSES, the whole computational domain is decomposed into several subdomains and one subdomain is assigned to one MPI process. Then, for every time step, each subdomain has to retrieve the information that has been previously updated at the boundaries of its neighbors, and also to send (pieces of) its boundaries to its neighbors so that they can update theirs. This becomes problematic in the case of high numbers of MPI processes, and will impact the strong scalability of the code. Indeed, when increasing the number of MPI domains while keeping the computational volume constant, the number of boundary cells increases, whereas the subdomains' size decreases. This implies an increase in the proportion of time spent on communications, while the computational cost per core keeps decreasing.

One way to improve the strong scaling of a code like RAMSES is thus to introduce the multithreading parallelism level, turning the code into a hybrid MPI+OpenMP code. This enables the use of fewer MPI processes, and hence less subdomains, to exploit a larger number of cores to solve the same problem. To introduce multithreading into the RAMSES code we use the well-known standard OpenMP. Each MPI process will run a certain amount of OpenMP threads, so that `num_processes x num_threads` equals the number of cores (or hyperthreads) on the system. We completed the work for the hydro kernel module and associated subroutines, of which the implementation and results are discussed in the remainder of this section.

5.2.1 OpenMP implementation

The main hydro kernel is `godunov.fine`. It consists of a loop over blocks of `nvector` cells (see also 5.3.1) which are first gathered before sending them off to the calculation routine `godfine1`. This loop structure is straightforward to parallelize using OpenMP:

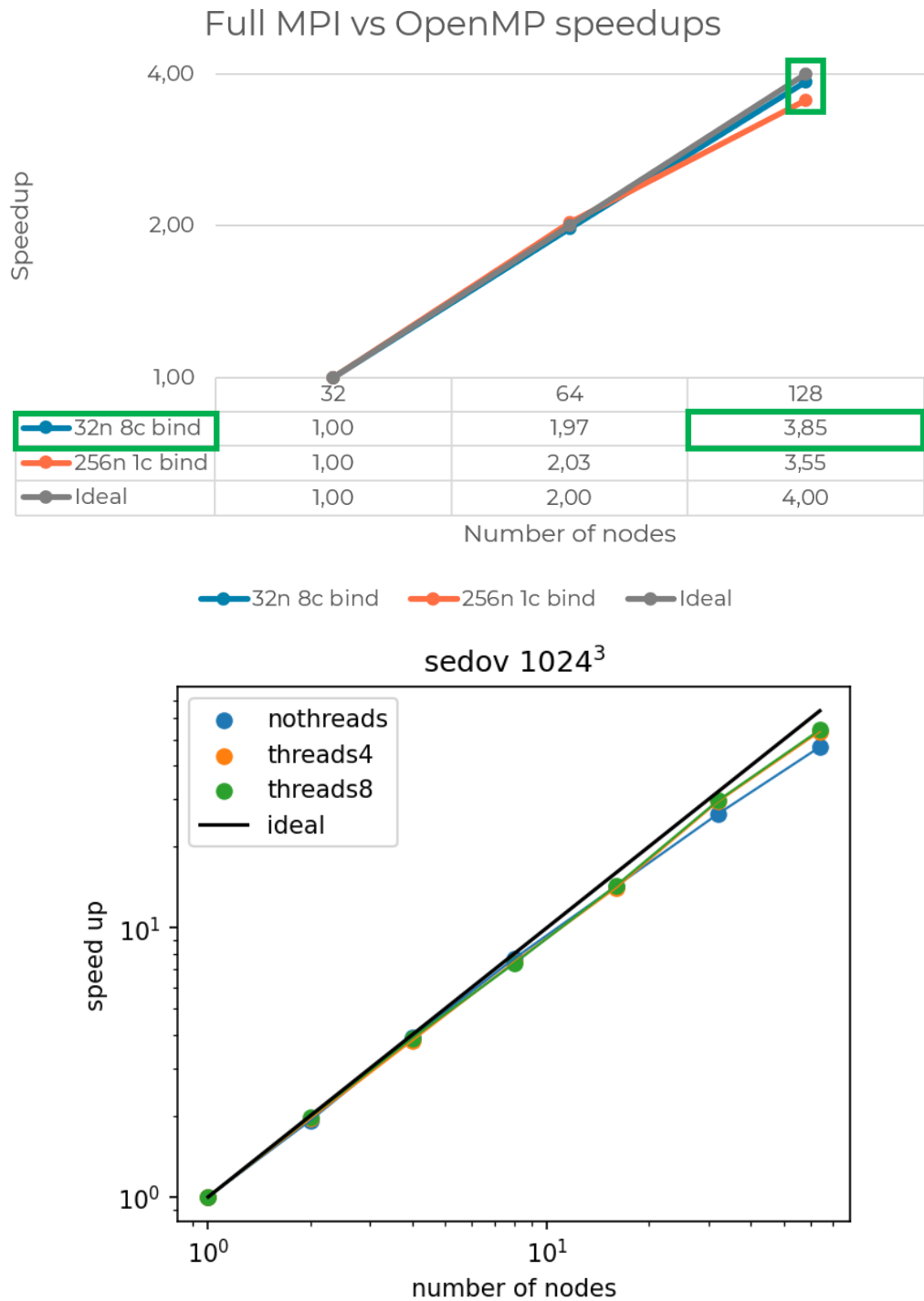


Figure 19: RAMSES - Comparison of the strong scaling of the MPI-only versus hybrid MPI+OpenMP implementation for the Sedov test case. Top: Test performed on the internal Eviden cluster with BullSequana X2410 blades equipped with AMD Milan architecture processors with 128 cores (256 threads) per node. The code was compiled with the AOCC compiler. Bottom: Benchmark performed on the MeluXina cpu partition using the GNU + OpenMP compiler.

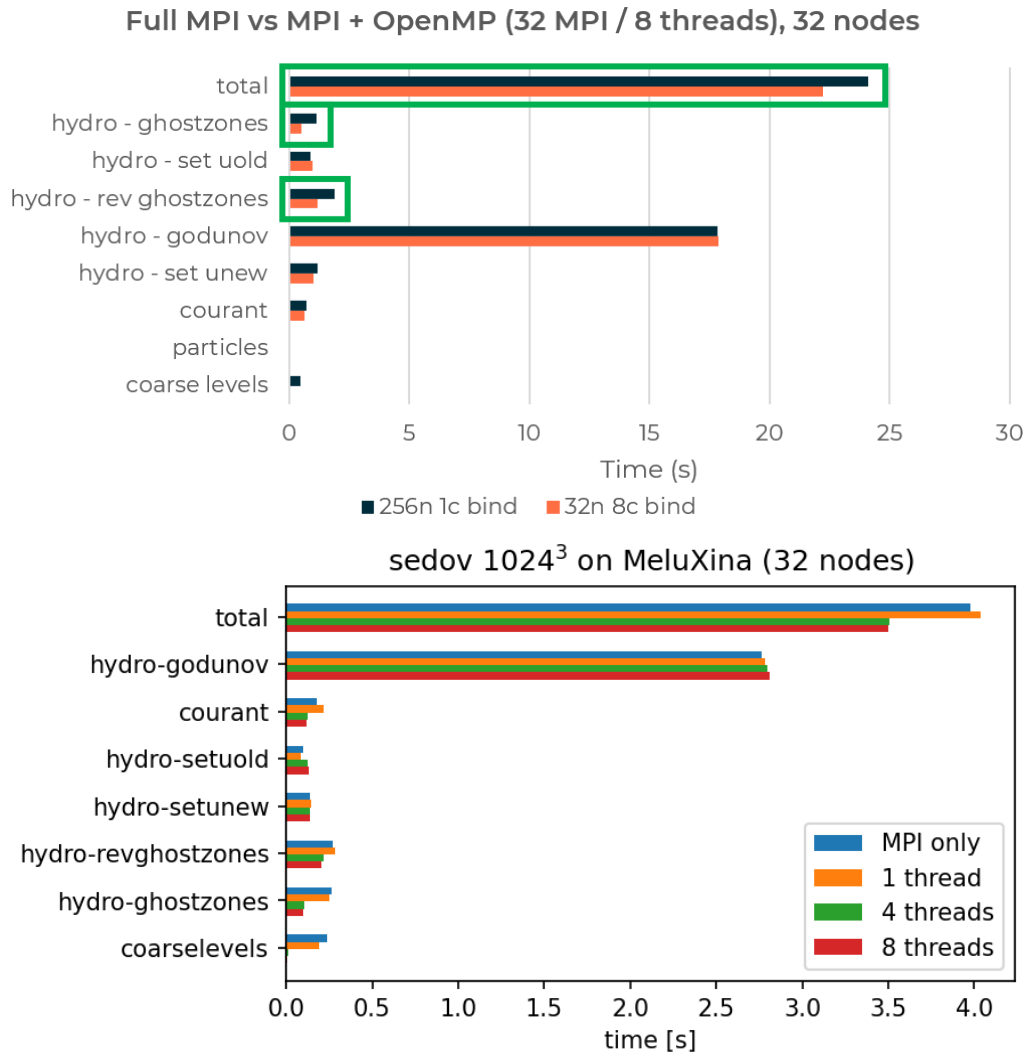


Figure 20: RAMSES - Comparison of the time spent on the different parts of the code for the Sedov test case, using the MPI-only implementation versus the new hybrid parallelization. The routines (rev) `ghostzones` deal directly with MPI communication. The routines `courant` and `coarse levels` contain an `MPI_ALLREDUCE`. Top: Eviden cluster, bottom: MeluXina CPU partition.

```

1 subroutine godunov_fine(ilevel)
2   ncache=active(ilevel)%ngrid
3   !$omp parallel do default(none) shared(ncache,ilevel,active) private(ngrid,ind_grid)
4   do igrd=1,ncache,nvector
5     ngrid=MIN(nvector,ncache-igrd+1)
6     do i=1,ngrid
7       ind_grid(i)=active(ilevel)%igrd(igrd+i-1)
8     end do
9     call godfine1(ind_grid,ngrid,ilevel)
10  end do
11 end subroutine godunov_fine

```

With this solution, each thread assembles its own vector, consisting of a portion of the mesh and its neighborhood. Then, each thread calls the same Godunov solving routine in order to solve numerically the hydro equations at the current time iteration. In the end, each thread writes back its results into the main arrays holding the physical variables of importance regarding the hydro module.

In order to maximize the coverage of OpenMP for the Sedov test, we further apply OpenMP to the loops in the subroutines `set_uold`, `set_unew` and `courant_fine`. The routine `courant_fine` calculates the time step using the Courant-Friedrich-Levy stability condition, and then takes the minimum among domains. Its loop structure is vectorized in the same manner as `godunov_fine`, and thus straightforwardly parallelized with OpenMP. In addition, using the clause `reduction(MIN:dt_loc)` gathers the minimum among OpenMP threads, before proceeding to `MPI_ALLREDUCE` to find the minimum between all MPI domains.

The loop structures in `set_uold` and `set_unew` have not been vectorized. They do not benefit from vectorization since no significant calculations are done and the routines are memory bound. Indeed, rewriting the loops to apply the same vectorization strategy as in `godunov_fine` led to a loss in performance when tested on the MareNostrum system. We opted to apply OpenMP to the loops over `twotondim`, which is equal to 8 for 3D simulations (or 4 for 2D simulations):

```

1 !$omp parallel do private(ind,iskip)
2   do ind=1,twotondim
3     iskip=ncoarse+(ind-1)*ngridmax
4     do ivar=1,nvar
5       do i=1,active(ilevel)%ngrid
6         unew(active(ilevel)%igrd(i)+iskip,ivar) = uold(active(ilevel)%igrd(i)+iskip,ivar)
7       end do
8     end do
9     ...
10  end do

```

This means that these routines would not benefit from choosing more than 8 OpenMP threads per MPI process. This is not an issue because of two reasons. First, the time spent in these routines is usually low. Second, threads compete for the cache, meaning that having many threads can lead to a loss in performance when the problem is memory bound. Experience teaches us that often the optimal number of OpenMP threads is 4 or 8.

As a low-level optimization, RAMSES declares certain variables using the `save` keyword. This will lead to issues when these variables are shared among OpenMP threads. In order to keep the advantages of the `save` option, we make sure these variables are declared to be thread private.

5.2.2 OpenMP results

With these modifications, we have a full OpenMP implementation for hydrodynamics simulations on uniform grids such as the Sedov test case. The results of strong scaling benchmarks on the MeluXina cluster and an Eviden in-house cluster are shown in Figure 19. Importantly, the reduced amount of MPI communication leads to an improved scaling. On MeluXina, the efficiency on 64 nodes increased from 73 to 85 %.

We obtain a decrease in the total execution time by 13% on 32 nodes when using 4 or 8 OpenMP threads on MeluXina. In Figure 20, we show a breakdown of the time spent in the different parts of the code. The `godunov` routine is purely computational. We see that, apart from a very minimal overhead, the computation time remains the same regardless of the number of threads, as expected. There is a significant reduction in the time spent in the ghost-zone routines, which handle direct MPI communication of the boundaries. Further, we obtain a gain in the routines `coarse-levels` and `courant`, which contain an `MPI_ALLREDUCE`. Finally, as in `godunov`, the time spent in `set_uold` and `set_unew` remains roughly constant.

For the Sedov test, most of the computational cost comes from calculations in `godunov`. In other cases, notably simulations with AMR where the grids are unevenly distributed among the MPI domains, communi-

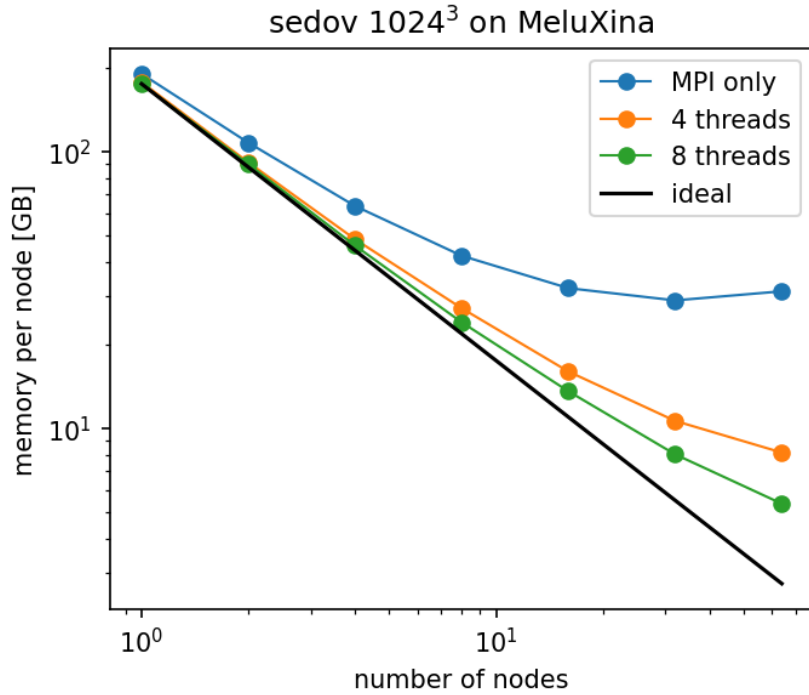


Figure 21: Memory usage per node for the strong scaling benchmark of the Sedov test on MeluXina for the RAMSES code.

cation takes up a bigger part of the total execution time. We are currently adding OpenMP to the refinement routines and preliminary results show that we can expect even larger increase in performance for these types of simulations.

Adding OpenMP does not only benefit the scalability, but also reduces the memory imprint of simulations. As mentioned before, in RAMSES the boundary cells between MPI domains are stored in the neighboring domains as so-called ghost-zones. This creates a memory overhead that increases with the number of MPI domains for a constant problem size. In Figure 21, we compare the memory consumption per node for a Sedov test case with 1024^3 cells using the MPI-only implementation versus the hybrid-implementation using either 4 or 8 OpenMP threads. For the MPI-only case, we see that around 32 nodes, the ghost-zones take up about as much memory as the actual grid and increasing the number of MPI processes even further increases the memory consumption per node. By using OpenMP, we can dramatically reduce the combined memory imprint of the ghost-zones. On 32 nodes, using 4 OpenMP threads (and thus decreasing the number of MPI domains by a factor 4) decreases the amount of memory needed by a factor about 3: we need only 37% of the memory per node compared to the MPI-only implementation. Using 8 threads decreases the required memory even further, to 26%; only a fourth of what was needed for the original MPI implementation. The addition of OpenMP will thus be especially beneficial for memory-heavy simulations, for example those using radiative transfer or chemistry where extra variables need to be stored.

5.3 CPU optimizations

In this section, we present pure CPU optimizations made to the Godunov solver of the RAMSES code. After an algorithmic preamble in a first subsection, a roofline analysis of the code highlights the main hotspots of the code. Then, the two remaining subsections present the optimizations performed regarding the memory accesses and the vectorization for x86 based architectures.

5.3.1 Algorithmic preamble: software cache of size nvector for the AMR grid

In order to save computations where high precision is not required, the RAMSES code uses an AMR mesh. An important side effect of this approach is that the mesh might change during the simulations. This leads to a

complex data description. In particular, it is complicated to retrieve the neighborhood of every cell in the mesh, and the underlying indirect memory access patterns might be harmful regarding code performances.

To avoid this problem, the hydro module of RAMSES begins by filling a kind of software cache, often referred to as *vectors*, into the code. This enables to load a contiguous portion of the mesh and all its neighborhood once for all before calling the Godunov solver, hence favouring data locality and compiler based vectorization.

The size of these vectors, denoted by `nvector`, is defined at compile time. This parameter has a strong impact on the Godunov solver's performances. Indeed, large vectors lead to a better vectorization rate, but small vectors are better for hardware cache performance. Hence, the best compromise has to be found between hardware cache performances and vectorization rate for each architecture.

5.3.2 Roofline analysis

This part aims at identifying the intra-MPI process optimization opportunities for the hydro module of the RAMSES application. To this end, a roofline analysis has been performed in order to find the most time consuming kernels of the Godunov solver.

This study has been performed on Intel Sapphire Rapids processor (codename 9480) of the Eviden in-house cluster. The corresponding nodes are equipped with bi-socket processors, with 56 cores per socket and a total of 512 GB of RAM. The Intel legacy compiler was used (`ifort`) in version 2024.1. The case considered for this study is the well known Sedov blast test case, with 3 dimensions in space. Several mesh sizes are available. Only two of them were considered in order to exploit a significant fraction of the memory of one node: the first with a cubic mesh of 128 cells in each direction (128^3 in total), and a second with a total of 256^3 cells. For this particular test case and architecture, the parameter `nvector` (see 5.3.1) has been set to 16, leading to the best achievable performances.

As the study focuses on intra-node performance, the code was compiled without the MPI support. However, the runs were replicated over all the cores of the node in order to mimic the memory behaviour of a classic MPI run, where all the cores of a node are in use.

The annotated Intel Advisor roofline chart given by Figure 22 shows several hotspots that might be interesting to optimize in order to reduce time to solution into the Godunov solver.

The first and most important one, the loop in `godfine1` at line 582 (bottom left on the figure), corresponds to the filling of *vector* arrays that store a contiguous part of the mesh and its neighborhood before performing the computations of the Godunov solver. This hotspot is limited by the memory bandwidth and also suffers from low vectorization rate. Investigating the code to minimize these memory accesses and to improve the vectorization could help to reduce the time spent into this part of the code.

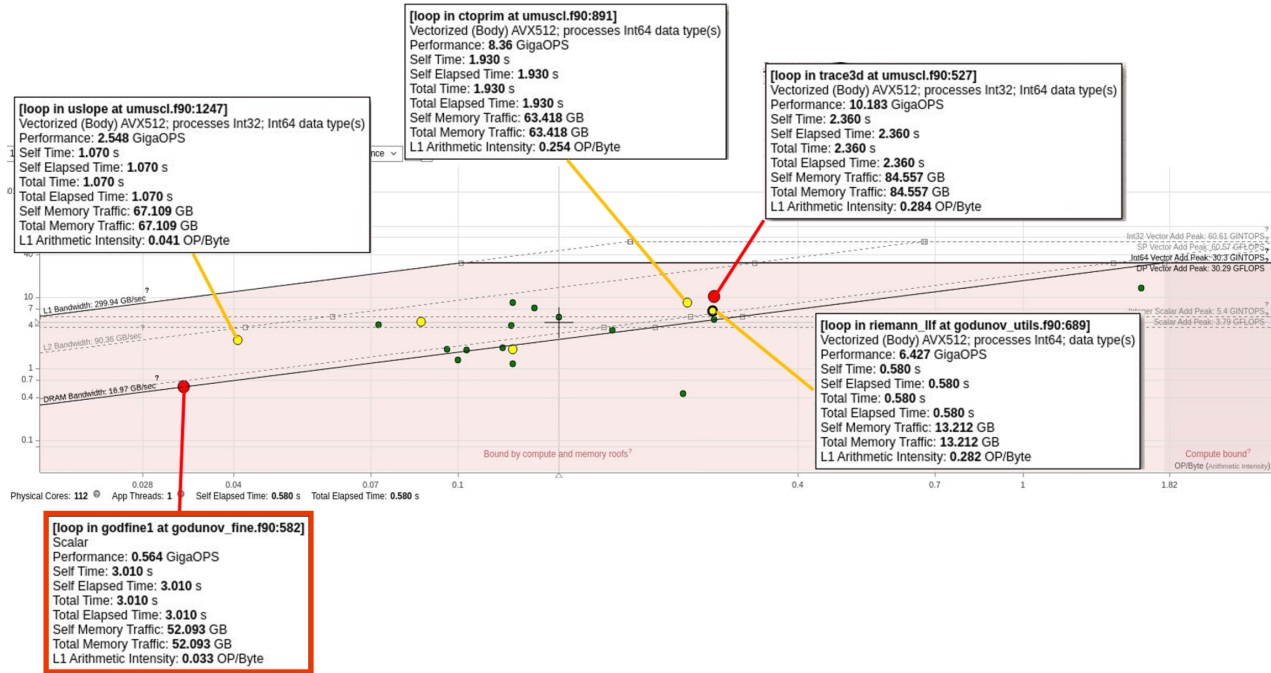
The 4 next hotspots correspond to computational parts (onto *vectors* of size `nvector`). These functions are deeper into the callstack of the Godunov solver. The `trace3d` function suffers from a low vectorization rate (34%). Some code modification and or manual vectorization can be tried to reduce computational time. For other functions, despite the vectorization rate is higher (more than 50%), these parts can be investigated the same way than `trace3d` to obtain better performances.

5.3.3 Memory optimizations

The main hotspot highlighted by the roofline analysis (see Figure 22) corresponds to the filling of *vectors* prior to the computations of the Godunov solver. By further inspecting the code performances with the MAQAO tool, we first notice that the high execution time is partly due to a poor vectorization rate jointly with a high number of indirect non-unit stride memory accesses. To optimize this part of the code, we managed to factorize as many indirect memory accesses as possible and to remove the non-unit stride stores when possible.

As a main example, one can inspect the differences between the original (Figure 23) and optimized (Figure 24) versions of the gathering of the hydro variables into the `godfine1` subroutine. We can see that, in the optimized version, the store operations into `uloc` are now unit-strided (one loop over index `i` instead of two loops) and that an array `shiftnexist`, also used in other loops, has been precomputed to perform an index shift when reading into the array `u2`.

As a result, the optimized `godfine1` subroutine shows a 100% vectorization rate (with respect to 0% in the original version) and more unit-strided store operations, resulting in an overall performance gain of 9.2% on the Sedov blast test case with a mesh of size 128^3 .

Figure 22: Roofline analysis of the hydro module of RAMSES, Sedov blast benchmark with a 128^3 mesh.

```
! Gather hydro variables
do ivar=1,nvar
  do i=1,nexist
    uloc(ind_exist(i),i3,j3,k3,ivar)=uold(ind_cell(i),ivar)
  end do
  do i=1,nbuffer
    uloc(ind_nexist(i),i3,j3,k3,ivar)=u2(i,ind_son,ivar)
  end do
end do
```

Figure 23: RAMSES godfine1 subroutine, original uloc filling.

```
! Gather hydro variables
do ivar=1,nvar
  nbuffer=0
  do i=1,ncache
    if(igrid_nbor(i)>0) then
      uloc(i,i3,j3,k3,ivar)=uold(ind_cell(i),ivar)
    else
      uloc(i,i3,j3,k3,ivar)=u2(shift_nexist(i),ind_son,ivar)
    end if
  end do
end do
```

Figure 24: RAMSES godfine1 subroutine, uloc filling optimized.

Compiler	GNU 13.2.0	Intel IFORT 2023.2.0	Intel IFX 2023.2.0
Gain	3.7%	1%	9%

Table 8: Overall performance gain on the total runtime of RAMSES depending of the compiler used on Intel Sapphire Rapids.

	CPU optimization		Scalability optimization		
Kernel name	focus	performance gain [%]	focus	number of nodes/CPU cores	Parallel efficiency improvement [%]
Godunov solver	vectorization	up to 9%	–	–	–
Godunov solver	indirect memory access	up to 9.2 %			
Godunov solver			OpenMP implementation	64/8192	up to 11%

Table 9: Summary of the optimization work performed on RAMSES.

5.3.4 Vectorization optimizations

Regarding the computational kernels, obtaining better performances was difficult, especially when using the legacy Intel `ifort` compiler (version 2023.2.0). The software cache, controlled by the compile-time parameter `nvectorf` (see section 5.3.1), is already very efficient to allow a good level of compiler-based vectorization in most of the computational kernels involved in the hydro module of RAMSES.

However, various optimizations have been tried in order to get better performances. Among them, factorizing division instructions and turning conditional branching in Fortran `MERGE` operations into the `trace3d` and `uslope` hotspots lead to better performances depending on the compiler used to build the code. The original implementation of function `ctoprim` is already well optimized, showing good vectorization rates. The `riemann_11f` function was not further investigated due to its low contribution to the overall timing of the considered Sedov blast test case with a 256^3 mesh size (less than 4%).

Table 8 summarizes the overall performance gains thanks to the vectorization optimizations depending on the compiler in use. As stated in the previous paragraph, the performances regarding the vectorization optimizations are very sensitive to the chosen compiler.

5.4 Conclusions

The hydrodynamical Godunov solver of RAMSES has been ported to a hybrid MPI and openMP parallelization on CPU. We obtained an increase in the scalability of RAMSES of more than 10% on 8192 cores. The solver has been further optimized focusing on automatic vectorization and the identification of indirect memory access. The next step will be to port openMP implementation to the AMR routines and in the Particle-Mesh solver.

6 BHAC

6.1 Short description of the code

The Black Hole Accretion Code (BHAC) [11, 12, 13, 14] is a multidimensional General Relativistic Magneto-hydrodynamics (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 [15, 16] 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.

6.2 Summary of optimization work for the entire code

An extensive OpenMP porting and optimization campaign has already been carried out [17] for BHAC just before the start of the SPACE CoE project in cooperation with the Astro and Plasma physics Application Lab (AstroLab) of the Leibniz Supercomputing Center. The performance of the optimized version of BHAC improved by approximately 30% compared to its original version, making it viable for scaling on hundreds of thousands of supercomputer cores. An example of the optimization achievements accomplished during this campaign is summarized in Figure 25. The considered run features 2 MPI task per node and 24 OpenMP threads per task. The Intel VTune Profiler timeline histograms refer to a single task, with the master thread on top followed by 6 working threads that depend on it. Comparing the initial (top) to the optimized (bottom) panel one observes that the idle and wait times are very much reduced, resulting in much more compact parallel regions and thus highly increased core utilization. The bottom panel demonstrates higher thread utilization, reduced waiting and transitions, and more consistent execution across threads. This points to better exploitation of available hardware resources and an overall more efficient parallel implementation. Having these achievements in mind, we have not devoted any efforts to further optimize BHAC on a CPU level as it is already highly CPU-optimized.

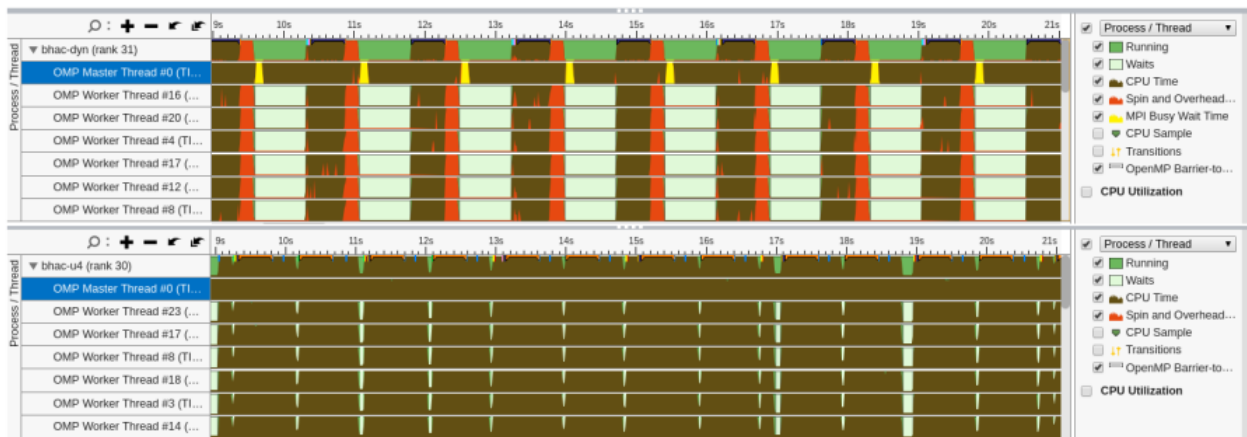


Figure 25: Timeline histograms of initial (top) and optimized (bottom) version of BHAC. Wait and overhead time (see color legends) are enormously reduced while CPU utilization is increased at the bottom panel.

BHAC entered the SPACE CoE project as a pure CPU code that is able to run and scale efficiently on thousands of CPU-cores. No preliminary work has been done on any aspect of the GPU-porting of BHAC before the start of the SPACE CoE project. Since March, an intensive GPU porting campaign has been launched in collaboration with IT4I. Substantial work has already been done on porting individual modules at the beginning and the whole code later to GPU. As a result, BHAC has been successfully ported to GPU using OpenACC and the Nvidia Nsight tools to track the performance of the porting. On a single GPU we can parallelize approximately 95% of the workload. Although it may seem like a high portion, it leaves 5% of the workload to be computed sequentially on a CPU, which means that the theoretical maximum speed-up

	CPU optimization		GPU optimization	
Kernel name	focus	performance gain [%]	focus	performance gain [%]
con2prim	vectorization + OpenMP optimization	30%	OpenACC code optimization	90%
Riemann	vectorization + OpenMP optimization	30%	OpenACC code optimization	90%

Table 10: Summary of the optimization work performed on BHAC.

is 20-fold. To go beyond this we need to parallelize the remaining 5%. We have taken some actions in this direction like simplifying the kernel responsible for varying the timestep Δt between consecutive timeslices and offloading the boundary conditions kernel to GPU. BHAC shows at the moment a 10-fold speed-up on a single GPU. Recently, BHAC was able to run on multiple GPUs for the first time allowing us to investigate the scalability of the GPU code. Although the results are at a very early stage, currently BHAC scales reasonably well on a single GPU node on Karolina. Here the most critical thing is the data movement between host (CPU) and device (GPU). The block based grid structure of BHAC where the numerical domain is divided into a specific number of blocks with overlaps (halo/ghost zones) for communication, should allow us to implement some kind of Comm/Comp overlap where we can hide the latency of host to device data transfers. At a later stage, we could also use CUDA-aware MPI to transition from CPU-mediated communication to direct GPU communication. By overlapping computation and communication and leveraging CUDA-aware MPI, we can achieve significant performance gains by minimizing idle times and reducing the latency associated with data transfers.

6.3 Summary of optimizations effort performed on individual kernels, modules

Currently, two modules have been fully ported to a single and multi-GPU environment: the primitive reconstruction and Riemann solver. Of course they need further testing and optimization in order to be able to function as efficiently as possible on a single as well as on multiple GPUs. Notice that we use a uniform grid while porting to GPU for the reason that the MPI-AMRVAC framework that provides the AMR functionality to BHAC has not been ported to GPU—a task that is outside of the scope of SPACE CoE.

6.4 Conservative to primitive routine (con2prim)

6.4.1 CPU optimizations

See discussion in Sec. 6.2.

6.4.2 GPU porting and optimizations

The module responsible for the conservative to primitive inversion has already been ported to GPU using OpenACC. Only one of the three inversion strategies available in BHAC has currently been ported to GPU: the so-called *entropy inversion method*. This method is sufficient for most of BHAC's applications as it has been designed to deal with highly magnetised regions. Although it can also be used in regions of less extreme magnetization, it is more efficient to use there the other two less resource demanding methods 1D [18] and 2DW [19]. Thus, the porting of the entropy inversion method will be used as a blueprint for the GPU-porting of the other two inversion methods. The **con2prim** routine is based on a Newton-Raphson method, we have measured the average thread divergence due to the iterative nature of the Newton-Raphson method and have found that divergence is not hindering performance, so no further actions were needed here. Sources of inefficiencies are discovered by examining GPU traces obtained using Nvidia Nsight Systems. Figure 26 depicts an example of

such an inefficiency related to a kernel that is not fully parallelised because it is running sequentially or in a single block fashion. Resolving this issue involves rewriting part of the code and using appropriate OpenACC data directives. In this particular example the speed-up is approximately 116-fold.

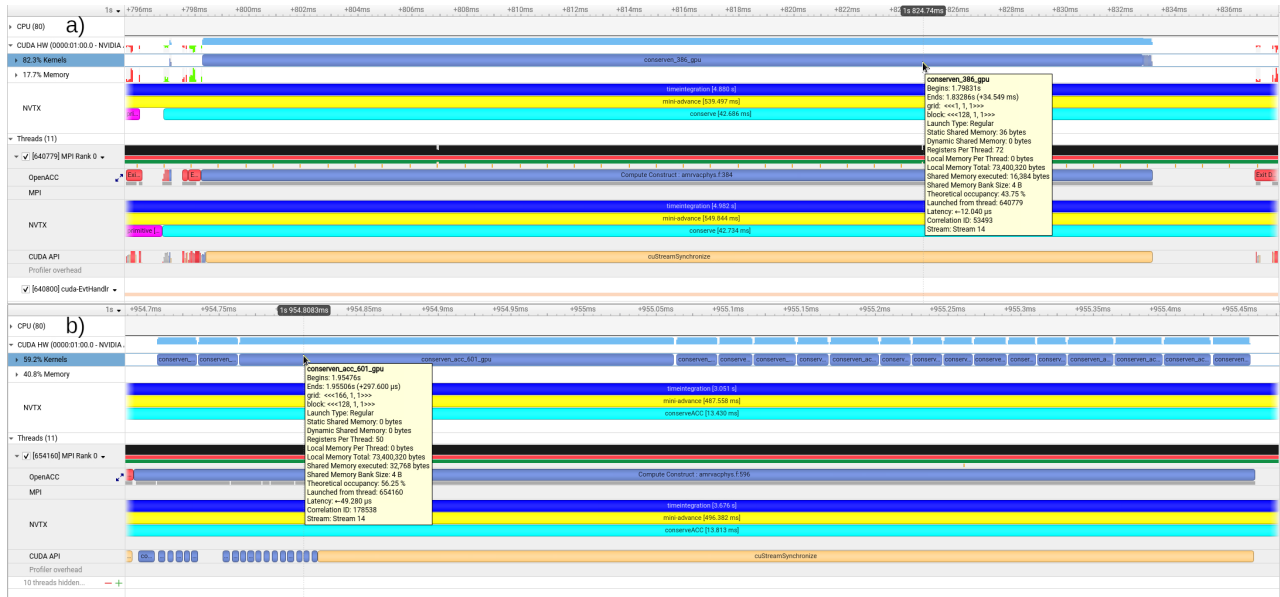


Figure 26: Nsight Systems timeline histograms of part of the `con2prim` module: semi-sequential version (top) and fully parallel version (bottom). The problematic kernel can be easily identified by abnormally long runtime. The performance increase is approximately 116-fold.

6.4.3 Scalability optimizations

We have not taken any actions here yet, as the code was only recently enabled to run on multiple GPUs. We are planning to implement some kind of computation and communication overlap between the inner grid regions and the ones close to the boundaries at the exterior of the grid. In order to allow for direct communication between GPUs across different processes without involving the CPU we plan to use CUDA-aware MPI, which will eliminate the overhead of intermediate CPU copies, significantly reduce latency and improve performance.

6.5 Riemann solver

6.5.1 CPU optimizations

See discussion in Sec. 6.2.

6.5.2 GPU porting and optimizations

The Rusanov flux, also known as the Total Variation Diminishing Lax-Friedrichs scheme (TVDLF), has been already ported to GPU using OpenACC. Currently, the ported TVDLF solver is being optimized in order to increase the parallelization efficiency and reduce—if possible eliminate—the host (CPU) to device (GPU) data transfer. Figure 27 depicts an example of the work that is currently carried out to reduce data transfers between host and device and vice versa. The reduction in the number of data transfers visible in Figure 27 is approximately 80%. With the TVDLF solver already ported we continue porting additional solvers available in BHAC’s infrastructure like the Harten-Lax-van Leer (HLL), the Harten-Lax-van Leer-Contact (HLLC) etc.



Figure 27: Nsight Systems timeline histograms of initial (top) and optimized (bottom) version of BHAC. Shown here is part of the Riemann solver module. Host to device and device to host data transfer is enormously reduced and GPU utilization is increased at the bottom panel.

6.5.3 Scalability optimizations

See discussion in Sec. 6.4.3.

6.6 Conclusions

On the CPU side BHAC is highly optimized and can scale on thousands of CPU-cores. BHAC has been already ported to a single GPU using OpenACC and it is currently further optimized. Recently, it has been ported to multiple GPUs and a scalability optimization plan has been already set up.

7 FIL/GRACE

7.1 Short description of the code

The Frankfurt Illinois GRMHD code (FIL) has been the flagship code of the Relativistic astrophysics group at Frankfurt since its development in 2018 [20]. FIL is written in C++ and uses OpenMP and MPI. The code was designed to simulate Binary neutron star (BNS) collisions and uniquely provides a fourth-order accurate conservative finite-difference scheme, making it a state of the art within the numerical relativity community. FIL leverages the computational infrastructure provided by the Einstein Toolkit (ET)[21], an open source framework designed to facilitate numerical relativity simulations. The ET provides the numerical grid, the AMR and parallelization that FIL uses. CPU scaling tests as seen in figure 28 shows FIL has excellent scaling results up to 512 nodes. The scaling tests were produced using a head on collision of two neutron stars. A head on collision contains all the physics and computational complexity of a full BNS inspiral. Whilst the CPU performance of FIL is good for GPU porting of the code things are a little more complicated.

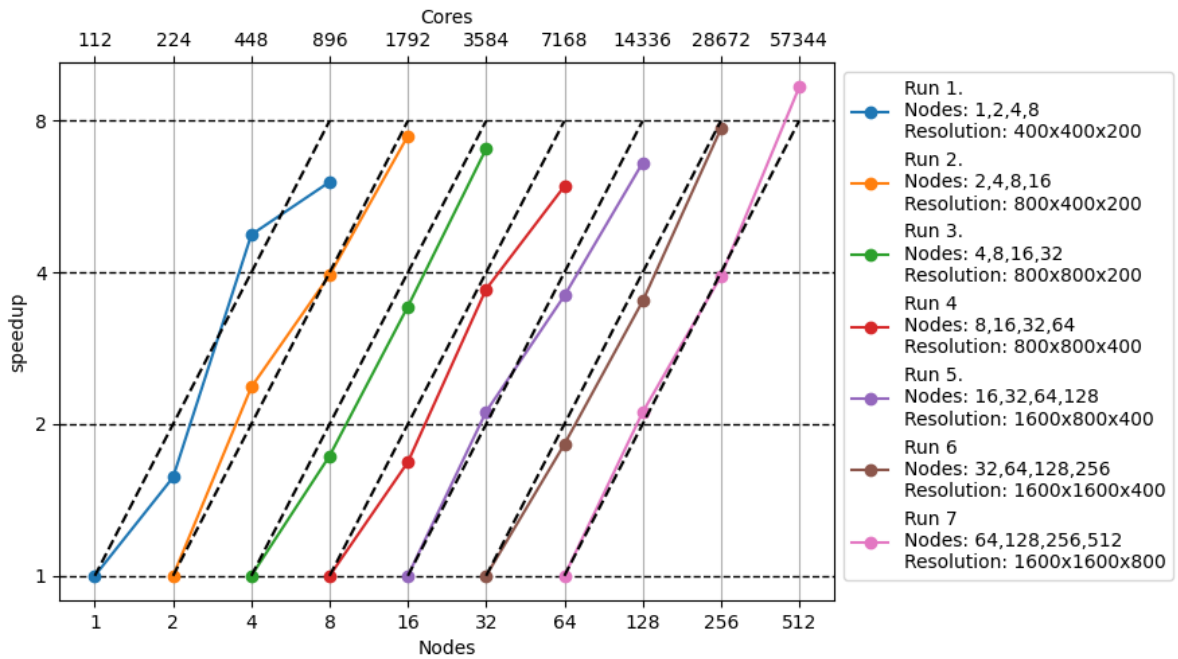


Figure 28: Strong and weak scaling plot for FIL on Marenostrum, FIL shows excellent scaling up to 512 nodes. Super-optimal scaling is the result of each data point being generated from a single run due to limited computational resources.

Currently the ET team in North America are developing a newer version of the Einstein toolkit that is GPU compatible, the current version is not. Unfortunately the development of the new GPU ET has experienced delays and the current timeline is incompatible with the timeline of the SPACE project. It was therefore decided to create GRACE, an internal code that is designed from the ground up to do GRMHD simulations on GPUs. The development goal of GRACE is to perform BNS simulations, exactly the same as FIL, with all the FIL kernels targeted for optimization in D2.1 being ported to GRACE. GRACE leverages Kokkos for its parallelization strategy on heterogeneous hardware and uses the p4est library which provides an Octree based AMR scheme for computational efficiency. With Kokkos we are able to run GRACE on CPUs and GPUs with support for AMD and CUDA. In this report we will be showing the current performance of GRACE and comparing it to FIL.

7.2 Summary of optimization work for entire code

Development of a numerical relativity code designed for modelling BNS mergers follows a tried and tested development path. First the code is devolved to perform simple relativistic hydrodynamic (HD) simulations,

it is here that kernels that will be central to all stages of development such as the conservative to primitive solver and the rhs flux evaluator are first implemented. These are the two kernels identified in report D2.1 as the kernels to target for optimization. Once the HD simulations are working and are tested by simulating a shock tube (a test with a semi-analytical solution) the codes are adapted to perform static GRHD simulations, by modelling isolated neutron stars in a non-evolving space time. This is the current stage of development of GRACE. Once a static GRHD simulation is conducted efforts can be made to implement a space time evolution framework that allows for BSN simulations to be performed. Currently work is being done to implement the most tested space time evolution frame work, the Baumgarte-Shapiro-Shibata-Nakamura (BSSN) formalism, into GRACE. This is expected to be complete and merged with the main branch in the first half of next year. Once GRHD simulations on a evolving background are complete then work can be done to implement more accurate microphysics such as magnetic fields and neutrino transport.

As GRACE utilises Kokkos it designed to be run on heterogenous hardware meaning it can be run on both AMD and CUDA GPUs. GRACE was originally developed on AMD GPUs and so has already been ported to LUMI and the 03 partition of the IT4I complementary systems. GRACE was recently successfully compiled and run on NVIDIA GPUs on the GPU partition of HAWK, the high performance computing centre Stuttgart's supercomputer, making it one of the first codes in the SPACE project to be run on both NVIDIA and AMD hardware. Currently work is underway to start porting GRACE to the HPC JU systems which have NVIDIA GPUs.

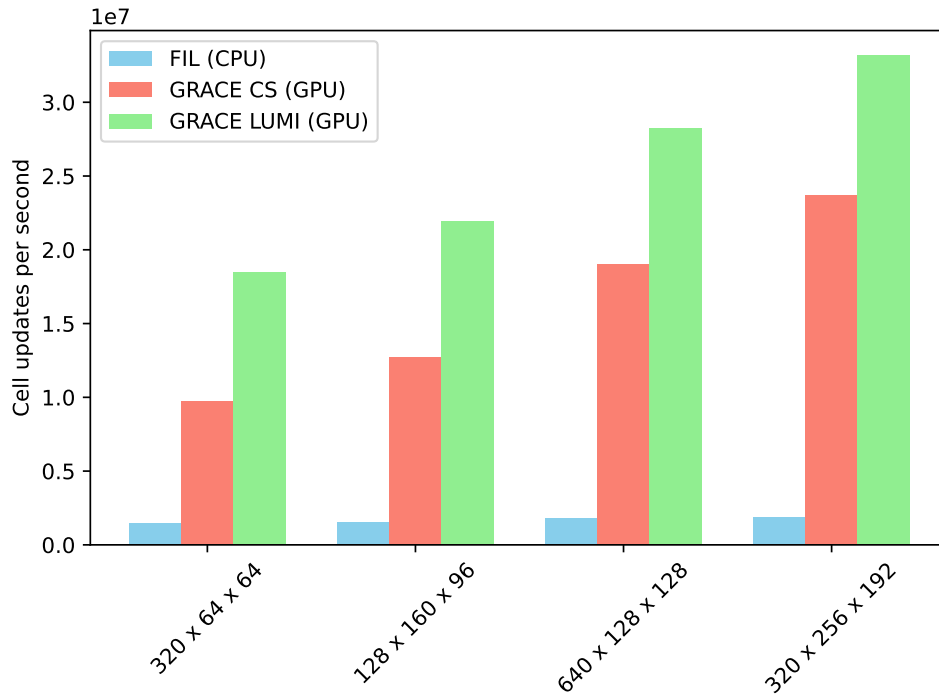


Figure 29: Performance comparison between FIL and GRACE where the performance metric is cell updates per second. This metric ignores IO contributions to the runtime of both codes.

Figure 29 shows a direct comparison between FIL and GRACE when running a simple shock tube test on a uni grid configuration. The comparison is made between a single CPU processor on the IT4I complementary systems (CS) p03 partition, a single M100 AMD GPU on the same node for the CS GPU case and a single M250 GPU on LUMI-G partition. As the memory on a single GPU is filled the GPU performance increases, at a resolution of 320x256x192 the M100's memory capacity was reached. On the M250 though GRACE was able to perform a simulation with resolution 720x128x128 which resulted in a cell updates per second of 4.47×10^7 which is about 25 times speed up when compared to the equivalent FIL performance.

	CPU optimization		GPU optimization	
Kernel name	focus	performance gain [%]	focus	performance gain [%]
Evaluate RHS	–	–	Porting Kernel using Kokkos framework 100% complete	99%
con2prim	–	–	Porting Kernel using Kokkos framework 100% complete	91%

Table 11: Summary of the performance gains of porting FIL kernels to GRACE.

7.3 Kernel porting and optimization

For a direct comparison between the performance of the FIL kernels identified for optimization in report D2.1 see table 11. By porting the **Evaluate RHS** to GRACE we achieved a 99% performance gain, the **con2prim** kernel saw a 91% performance improvement. Interestingly, the percentage of time spent in the **con2prim** kernel as a fraction of the total run time increases for GRACE compared to FIL, as can be seen in figure 30, and becomes a larger fraction of the overall run time compared to the kernel **Evaluate RHS**. The **con2prim** kernel depends on conditional statements and has a convergence criteria which will be reached at different times at different points on the grid. Both of these contribute to the **con2prim** kernel not reaching the same level of performance gain as the **Evaluate RHS** kernel however overall a 91% performance improvement is still very good. The remaining run time for FIL and GRACE is taken up predominately by boundary exchange and IO.

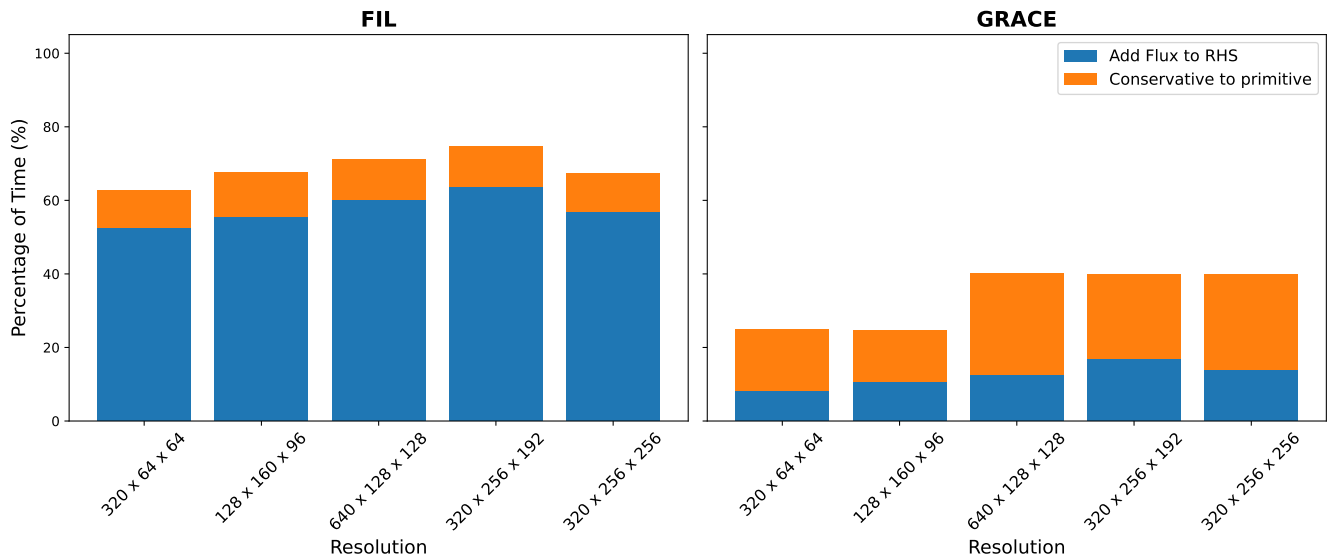


Figure 30: Breakdown of runtime of FIL and GRACE spent in kernel 1, add flux to RHS and kernel 2 the conservative to primitive solver. The data is a comparison between FIL ran on a single processor on the p03 partition of the IT4I complementary systems and a single M250 GPU on LUMI-G.

7.3.1 Scalability optimizations

Figure 31 shows the scaling performance of GRACE broken down by region as the number of GPU are increased. The set up for the scaling test was a unigrid shock tube as in figures 29 and 30 however the output was modified to be less frequent due to memory limitations. Figure 31 shows that overall GRACE scales well as the number of GPU cores are increased with the evolution region which is predominately comprised of kernel 1 and 2 also scaling well. So far the number of GPUs available to the GRACE development team have been limited but now with GRACE being included in the SPACE project after the last interim review work can be done to test the performance of GRACE on larger number of GPUs.

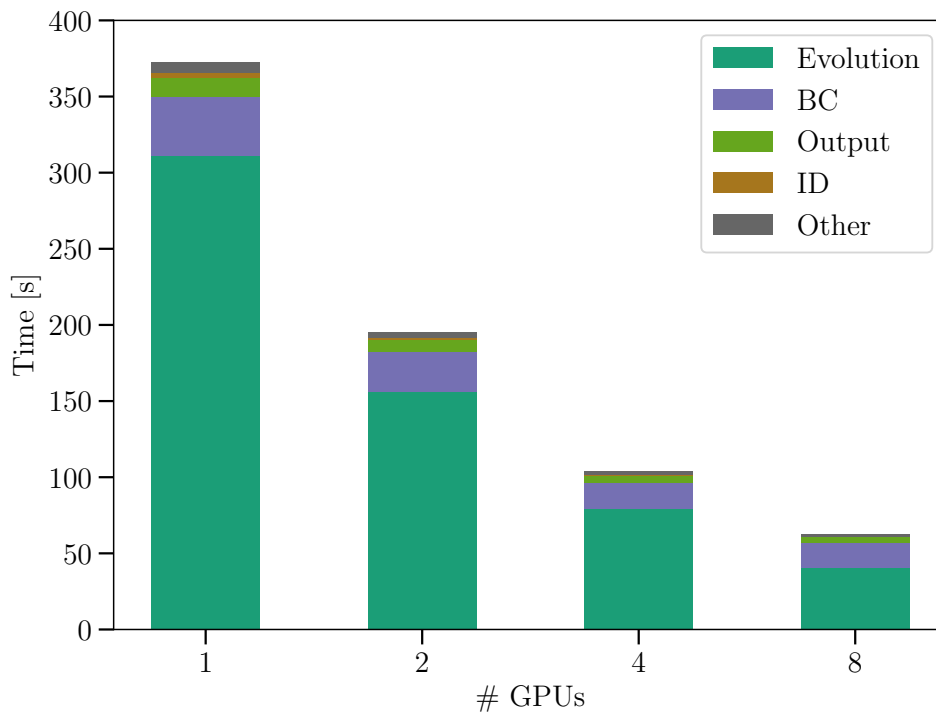


Figure 31: Breakdown of runtime of GRACE by code region on Goethe University Calea partition.

7.4 Conclusions

By developing GRACE from the ground up to perform GRMHD simulations on GPUs using the Kokkos library we have been able to side step the development bottle necks imposed on us by the Einstein toolkit non GPU compatibility. By using Kokkos we are able to run GRACE on heterogenous hardware CPUs, NVIDIA GPUs and AMD GPUs, by setting flags at compile time. By porting kernel 1 and kernel 2 to GPUs we have seen a 99 and 91 percent performance improvement respectively for a simple shock tube test. We expect this to be the high benchmark for the performance gain as when more complicated test/physics are performed the performance improvement will naturally decrease however as the kernels ported play a major role in all GRMHD simulations we are confident that significant performance gains will still be seen. Preliminary results of GRACE's scaling are good showing over 90% scaling efficiency up to 8 GPUs. With GRACE's inclusion into the SPACE project GRACE's scaling can be more thoroughly tested on more GPU cores.

8 ChaNGa

ChaNGa [22][23][24] 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 Smoothed Particle Magnetohydrodynamics (SPMHD) algorithms are based on the gasoline [25] and pkdgrav [26] codes, the unique feature of ChaNGa is its 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 D1.2 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 Symmetric Multi-Processing (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.

8.1 Summary of optimization work for entire code

	Scalability optimization			GPU optimization	
Kernel name	focus	number of nodes/G-PU/CPU cores	Parallel efficiency improvement [%]	focus	performance gain [%]
Gravity+SPH	Tree piece replication	see figure 32	see figure 32	–	–
Gravity+SPH	SFC-based load balancer	N32	20% in multi-time step	–	
GPU module				Removal of workrequest API	
Radiative cooling				Porting of radiative cooling	400% of module

Table 12: Summary of the optimization work performed on ChaNGa.

During the first phase, we focused on optimizing four major sections of the code. The first two involve scalability improvements. The first optimization is tree piece replication, which addresses the communication imbalance, identified in our initial performance analysis of D2.1. The second is the implementation of a new Space-Filling Curve (SFC)-based load balancer, which enhances parallel efficiency during multi-stepping runs and mitigates some of the shortcomings of the previous load balancer. We have also performed GPU optimizations. One of which is the removal of the deprecated WorkRequest Application Programming Interface (API) of Charm++, previously used for asynchronous kernel launches to the GPU. This have been replaced by handling the kernel launches inside the application itself. Another significant GPU optimization was offloading the radiative cooling module to the GPU. A summary of these optimization efforts is provided in Table 12. More details on these optimizations follows below.

8.2 Scalability optimization

8.2.1 Tree piece replication

During our performance analysis (D2.1) it became clear that ChaNGa suffers from communication imbalance for highly clustered datasets involving all the main physics modules. We have tackled this issue by introducing

a tree piece replication method within the code. In this method we replicate the information about the tree nodes on multiple processors, which spreads out the communication load and ensures that no single processor becomes overloaded with messages. A rough version of tree replication was implemented in an older version of ChaNGa, which we have now merged and updated to the newest version of ChaNGa. Below we give an example on how the tree piece replication code is implemented within the code. We begin by generating replicates and assigning a Processing Element (PE) to them:

```

1 void TreePiece::replicateTreePieces(const CkCallback& cb) {
2   ...
3   int hash_pe = thisIndex % CkNumPes();
4   int k;
5   // Distribute tree piece to 4 PE
6   for (int i = 0; i < 4; i++) {
7     k = hash_pe;
8     k += i*23;
9     k %= CkNumPes();
10
11     TreeReplicaMsg *reply = new (depth_count * ALIGN_DEFAULT(sizeof(Tree::
    BinaryTreeNode)))
12       TreeReplicaMsg(root->getKey(), thisIndex);
13
14     ((Tree::BinaryTreeNode*)root)->packNodes((Tree::BinaryTreeNode*)reply->data, depth_count,
    0);
15     tpReplicaProxy[k].recvTreePiece(reply);
16     ...
17   }
18   ...
19 }

```

Requests are then sent randomly to one of the four PEs hosting the tree piece information.

```

1   ...
2   //Previous code
3   //treeProxy[*idx.data()].fillRequestNode(msg);
4   //return NULL;
5
6   int hash_pe = (*idx.data()) % CkNumPes();
7   //CkPrintf(" requesting gravityNode %d key %d to pe %d\n", *idx.data(), key, hash_pe);
8   hash_pe += (rand() % 4) * 23;
9   hash_pe %= CkNumPes();
10  tpReplicaProxy[hash_pe].fillRequestNodeFromReplica(msg);
11  return NULL;
12  ...

```

There is still some cleanup to be done, right now the number of replicas is hardwired to be 4 and the hash function for replica placement needs to be refactored into an inline-function. We also want to make the hash function to be a little more general; in particular, it will break if you run on a multiple of 23 processors. The preliminary implementation of the tree replication approach have been seen to significantly improve the scaling for the full physics merger case, as can be seen in figure 32. We also observed that increasing the number of communication threads/SMP processes helps the scaling.

8.2.2 Improved load balancer and cleanup

Multi-time stepping poses several challenges for achieving accurate load balancing due to the varying load dynamics across substeps within a larger step. In multi-step simulations, only a subset of particles may be active during each substep. Predicting the load of a tree piece based solely on the preceding substep can lead to discrepancies between the expected and actual loads. For this reason, ChaNGa uses a hierarchical load balancing framework to improve the load balance for all the substeps. Processors are organized into independent groups following a hierarchical structure, with each group consisting of 512 processors. At each hierarchy level, the root node manages load balancing for the processors within its subtree. At higher hierarchy levels, a refinement-based load balancing strategy is applied. This approach minimizes task migration by considering the current task assignments. At the lowest hierarchy level, the Orthogonal Recursive Bisection (ORB) method have been used to distribute tree pieces among the processors within a subgroup.

The ORB strategy, unfortunately, occasionally resulted in scenarios where some processors were left without any assigned tree pieces. We have improved this by replacing the ORB strategy with an space filling curve

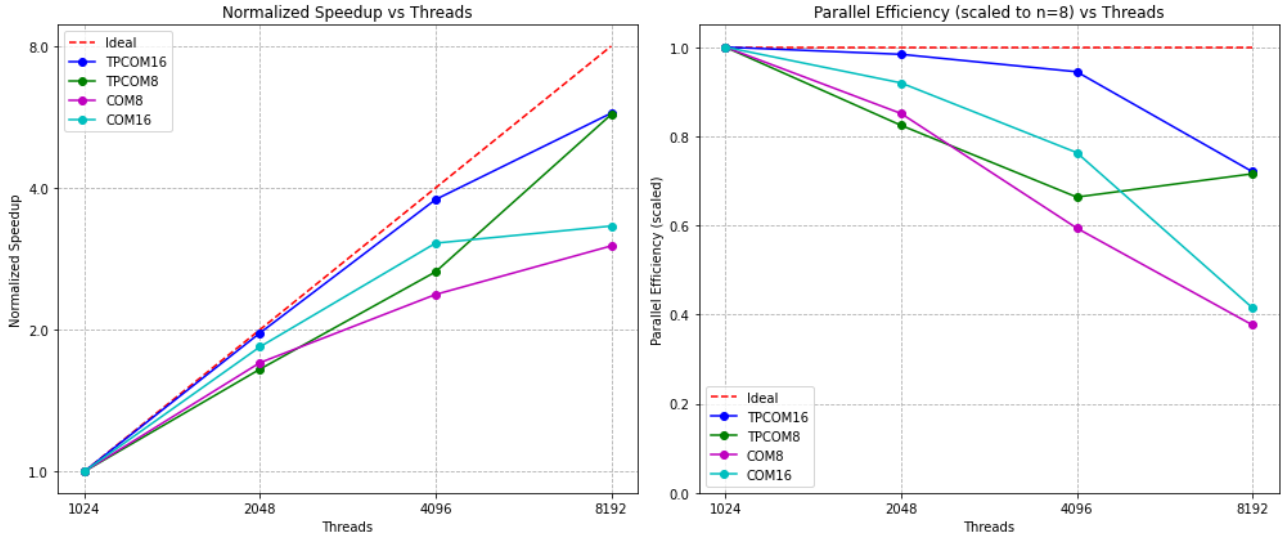


Figure 32: CPU scaling tests for the full physics merger simulation performed on the Karolina supercomputer (details provided in deliverable D1.1 and D2.1), comparing different numbers of communication threads (denoted as COMx) and configurations with or without tree replication (with tree replication denoted as TP). speed-up is normalized to 1 at 8 nodes, with relative speed-up shown on the left and parallel efficiency on the right. Results indicate significant performance improvements at higher thread counts with the tree replication approach. Increasing communication threads also enhances performance but is not the primary limiting factor at the highest thread counts.

(SFC) strategy, where we instead sort the tree pieces according to an SFC. The sorted list is then divided evenly to ensure a balanced distribution of tree pieces among processors. This approach eliminates the problem of idle processors and improves overall load balancing efficiency. This achieves roughly 20% speed-up for a multi-time step run of the galaxy merger simulation, where timestep (with 9 substep levels) gave the following improvement: $t_{\text{SFC}} = 47.569386$ sec and $t_{\text{ORB}} = 56.461131$ sec.

We also cleaned up the load balancing code a bit, which included the removal of some unnecessary counting of active particles.

8.3 GPU porting and optimizations

During the first period we have done two major optimizations to the GPU version of ChaNGa, this includes the removal of the deprecated workrequest [API](#) of Charm++ and the GPU offloading of the radiative cooling.

8.3.1 Removal of workrequest API and other GPU version improvements

The WorkRequest [API](#) in Charm++ was designed to facilitate GPU offloading asynchronously by allowing users to specify kernels, data buffers, and callbacks for execution on GPUs. However, this API has been deprecated in favor of the Hybrid API (HAPI), which offers an interface more akin to CUDA’s [API](#). As such ChaNGa directly handles kernel launches to the GPU. Currently GPU launches are tied to tree pieces, but this has recently been shown to degrade performance, as such kernel launches and device memory allocation will be handled by the DataManager in the future. The WorkRequest [API](#) was previously limited to a single CUDA stream per PE. However, with stream allocation now managed by the application, it supports the use of multiple CUDA streams, enabling finer-grained work to be performed with minimal overhead (see Figure 33).

8.3.2 GPU offloading of radiative cooling

Ionized gas undergoes cooling processes that dissipate thermal energy through mechanisms such as collisional excitation and ionization, inverse Compton scattering, recombination, and free-free emission. In numerical simulations, these cooling processes are incorporated as source terms in the internal energy equation. This is

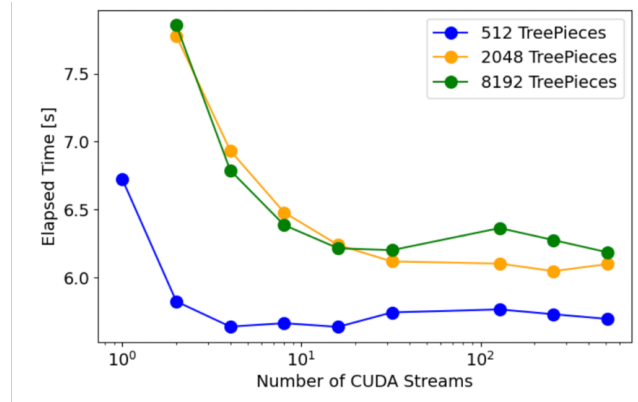


Figure 33: Scaling of the elapsed time with respect to the number of CUDA streams for a single time step on a single GPU. The three curves represent different numbers of tree pieces, showing that, in this case, a smaller number of tree pieces is more beneficial for GPU performance and that more streams can improve result for more tree pieces.

governed by a cooling function, which depends on the gas's ionization fractions, density, and composition. Due to the numerous local properties influencing the ionization rate, simplifying the calculations is essential to reduce computational overhead. Consequently, ionization rates are typically calculated based on a limited number of chemical species, and the cooling function relies on pre-computed tables for efficiency. Radiative cooling can become highly computationally expensive under certain conditions. This arises because radiative cooling operates as a semi-implicit module and requires large integration times when the cooling timestep is significantly shorter than the dynamical timestep. For this reason we have begun work to offload the radiative cooling to the GPU. This required making a few significant structural changes to the code. To reduce code duplication, the `__device__` `__host__` specifiers have been added to many of the cooling subroutines. Each cooling module now includes an empty `.cu` file, enabling the reuse of the existing C code on the GPU. When the CUDA flag is enabled, these `.cu` files are compiled separately for both the host and device using the `-dc` flag, with the final linking performed in a separate step at the end of the `cuda.mk` file. Additionally, the `clDerivs` function for cosmological cooling utilizes the `RootFind` routine, which necessitated making certain routines from `stiff.c` accessible on the device as well.

The parallel design of `CudaStiffStep` requires a separate `clDerivsData` and `Stiff` structure, along with space for their associated deep pointers, for each gas particle on both the host and device. Initially, the allocation of this data was managed by the `TreePieces` within the cooling subroutines. This responsibility has now been shifted to the `DataManager` through the `allocCoolParticleBlock` function, which pre-allocates host and device memory blocks. These blocks are then assigned to the `TreePieces` using the `setCoolPtrs` function. If the number of gas particles increases significantly, a larger memory block is re-allocated to accommodate the change. Below we show a timing improvement to the cooling module (uDot) done on the GPU when running a cosmological box.

1	With Gravity GPU + CPU performing cooling module (uDot timing):											
2	# Timings for step 9											
3	#	Rung	Count	Grav	uDot	DD	LoadB	TBuild	Adjust	EAdjust	Kick	Drift
4		0	1	7.790659	6.445027	1.201459	0.792857	0.447130	0.161316	0.000000	0.220131	1.599155
5		1	1	2.362555	0.199401	0.904625	1.163832	0.171340	0.041652	0.034467	0.041835	0.784393
6		2	2	6.239121	0.291784	1.485864	0.055339	0.631811	0.113557	0.074556	0.111437	2.402774
7	Totals:			16.392335	6.936212	3.591948	2.012029	1.250281	0.316525	0.109023	0.373403	4.786322
8												
9	With Gravity GPU + GPU performing cooling module (uDot timing):											
10	# Timings for step 9											
11	#	Rung	Count	Grav	uDot	DD	LoadB	TBuild	Adjust	EAdjust	Kick	Drift
12		0	1	6.861230	1.636123	1.370787	0.371337	0.227041	0.132859	0.000000	0.196331	1.056627
13		1	1	3.397212	0.196621	0.718322	0.025561	0.278802	0.052360	0.029349	0.051516	1.054147
14		2	2	4.919331	0.292127	1.642208	0.052545	0.561530	0.104879	0.056709	0.104596	2.124941
15	Totals:			15.177774	2.124872	3.731317	0.449443	1.067373	0.290099	0.086058	0.352443	4.235715

8.4 Conclusion

We have started to tackle the communication imbalance in ChaNGa by developing tree piece replication, which show significant improved scaling at higher node count. Continued testing of the tree replication code will be done and effort to further lower the communication volume (node-wide cache, improved cache manager). We have also improved scaling in multi-time stepping runs by adding an [SFC](#) load balancer. On the GPU side, we have removed the deprecated workrequest [API](#), giving us more fine-grain control over CUDA streams and started to port modules to the GPU, where a preliminary version of radiative cooling have already been ported to the GPU.

Code	number of optimized kernels	application optimized at preliminary level
PLUTO	3	YES - (a) with the new version of the code gPLUTO (GPU) we obtain around 30 of speed-up respect to the legacy CPU PLUTO; (b) the parallel scalability was improved thanks to the asynchronous boundary (30%)
OpenGadget	1	YES - 15 to 20% performance improvement expected
iPIC3D	3	YES (further performance improvement can be expected only for the GPU version, CPU version fully optimized)
RAMSES	1	YES (CPU version of Godunov solver fully optimized)
BHAC	2	YES (further improvement expected for GPU version)
FIL	2	YES (further improvement expected for GPU version)
ChaNGa	3	YES (but more improvement can be expected for both GPU and CPU versions)

Table 13: Summary table of code optimization actions with respect to Milestone 6.

9 Conclusions

This document reports the work performed on the optimization of the SPACE codes in the first 24 months of the project. It is divided in seven sections focusing on each of the seven codes of the SPACE project. Each code section details technicalities about the optimizations done for CPU, GPU and scalability. It is important to stress that this document does not fully reflect all the work that has been carried out to successfully optimize the codes, since failures and changes of strategy often take a lot of resources.

Tables 14 and 15 gather all the optimization work performed on all codes for this deliverable.

Overall, all codes have achieved either full optimization on CPU for a given kernel, and/or porting and optimization on GPUs. This deliverable serves as a proof that the project has reached Milestone 6 in M24 as stated in the Grant Agreement. Milestone 6 requires: **"At least 30% of codes optimized at preliminary level, or 100% of codes have optimized at least one fundamental kernel"**. A summary of the code optimization actions is given in Table 13.

	CPU optimization		GPU optimization		Scalability optimization		
Kernel name	focus	performance gain [%]	focus	performance gain [%]	focus	number of nodes/G-PUs/CPU cores	Parallel efficiency improvement [%]
PLUTO							
Riemann Solver	Template and inlining	5%	Template and inlining	10%	–	–	–
Reconstruct	Template and inlining	5%	Template and inlining	10%	–	–	–
Boundary	Asynch. implememntation	30%	Asynch. implememntation	30%	Asynch. implememntation	512/2048/-	30%
OpenGADGET							
Density & SPH	Loop restructuring	2× in loops, expected up to ~ 15% overall	–	–	–	–	
Tree Walk <i>in progress</i>	–	-	Reduction of thread divergence	8× on external kernel	–	–	–
Tree Build <i>in progress</i>	Change in algorithm	<i>to be established</i>	<i>same</i>	<i>to be established</i>	<i>same</i>	–	<i>to be established</i>
NUMA awareness <i>in progress</i>	Exploiting MPI memory windows & Comm-free algorithms	<i>to be established</i>	<i>same</i>	<i>to be established</i>	<i>same</i>	–	<i>to be established</i>
iPIC3D (new version of the code is 20 - 30 times faster than the older one)							
Particle Mover	OpenMP + MPI	9.7x	initial working version	–	Asynchronous Boundary	-/112-4000	1.2x - 3.8x
Moment Gatherer	OpenMP + MPI	2.8x	initial working version	–	Asynchronous Boundary	-/112-4000	1.2x - 20x
Field Solver	OpenMP + MPI	6.4x	initial working version	–	Asynchronous Boundary	-/112-4000	1.2x - 5.7x
RAMSES							
hydrodynamical solver	vectorization	9%	–	–	–	–	–
hydrodynamical solver	indirect memory access	9.2%	–	–	–	–	–
hydrodynamical solver	–	–	–	–	OpenMP implementation	64/8192	11%

Table 14: Summary of the optimization work performed on all codes - part 1. (Note: For iPIC3D there three points we need to highlight: (a) we have a new version of the code that can be 20 - 30 times faster than the older one, and (b) CPU code is production ready — 100% optimized, (c) we have a fully-working GPU code (needs improvements).)

	CPU optimization		GPU optimization		Scalability optimization		
Kernel name	focus	performance gain [%]	focus	performance gain [%]	focus	number of nodes/G- PUs/CPU cores	Parallel efficiency improvement [%]
BHAC							
con2prim	vectorization + OpenMP optimiza- tion	30%	OpenACC code opti- mization	90%	–	–	–
Riemann	vectorization + OpenMP optimiza- tion	30%	OpenACC code opti- mization	90%	–	–	–
FIL							
Evaluate RHS	–	–	Porting Kernel using Kokkos framework	99%	–	–	–
con2Prim	–	–	Porting Kernel using Kokkos framework	91%	–	–	–
ChaNGa							
Gravity + SPH	–	–	–	–	Tree piece replication	64/-/8912	30%
Gravity + SPH	–	–	–	–	SFC-based load balancer	32/-/4096	20% in multi-time step
GPU module	–	–	Removal of workre- quest API	to be established	–	–	–
Radiative cooling	–	–	Porting of radiative cooling	400%	–	–	–

Table 15: Summary of the optimization work performed on all codes - part 2

References

- [1] “The pluto code for astrophysical gasdynamics,” <http://plutocode.ph.unito.it/>, accessed: 28.10.2023.
- [2] V. Springel, “The cosmological simulation code GADGET-2,” *MNRAS*, vol. 364, no. 4, pp. 1105–1134, Dec. 2005.
- [3] 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,” *MNRAS*, vol. 455, no. 2, pp. 2110–2130, Jan. 2016.
- [4] F. Groth, U. P. Steinwandel, M. Valentini, and K. Dolag, “The cosmological simulation code OPENGADGET3 - implementation of meshless finite mass,” *MNRAS*, vol. 526, no. 1, pp. 616–644, Nov. 2023.
- [5] 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.
- [6] S. Markidis, G. Lapenta, and Rizwan-uddin, “Multi-scale simulations of plasma with iPIC3D,” *Math. Comput. Simul.*, vol. 80, no. 7, pp. 1509–1519, 2010.
- [7] G. Lapenta, “Exactly energy conserving semi-implicit particle in cell formulation,” *J. Comput. Phys.*, vol. 334, pp. 349–366, Apr. 2017.
- [8] F. Bacchini, “RelSIM: A Relativistic Semi-implicit Method for Particle-in-cell Simulations,” *ApJS*, vol. 268, no. 2, p. 60, Oct. 2023.
- [9] R. Teyssier, “Cosmological hydrodynamics with adaptive mesh refinement. A new high resolution code called RAMSES,” *AAP*, vol. 385, pp. 337–364, Apr. 2002.
- [10] “Repository of the ramses code on bitbucket,” <https://bitbucket.org/rteyssie/ramses/src/master/>, accessed: 29.10.2023.
- [11] 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.
- [12] 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.
- [13] “The Black Hole Accretion Code – Documentation,” <https://bhac.science>, accessed: 2023-11-01.
- [14] “Repository of BHAC on GitLab,” <https://gitlab.itp.uni-frankfurt.de/BHAC-release/bhac>, accessed: 2023-11-01.
- [15] “The MPI - Adaptive Mesh Refinement - Versatile Advection Code – Documentation,” <https://amrvac.org/>, accessed: 2023-11-01.
- [16] “Repository of MPI-AMRVAC on GitHub,” <https://github.com/amrvac/amrvac>, accessed: 2023-11-01.
- [17] S. Cielo, O. Porth, L. Iapichino, A. Karmakar, H. Olivares, and C. Xia, “Optimizing the hybrid parallelization of bhac,” *Astronomy and Computing*, vol. 38, p. 100509, 2022. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2213133721000639>
- [18] B. van der Holst, R. Keppens, and Z. Meliani, “A multidimensional grid-adaptive relativistic magnetofluid code,” *Comput. Phys. Commun.*, vol. 179, pp. 617–627, 2008.
- [19] S. C. Noble, C. F. Gammie, J. C. McKinney, and L. Del Zanna, “Primitive variable solvers for conservative general relativistic magnetohydrodynamics,” *Astrophys. J.*, vol. 641, pp. 626–637, 2006.
- [20] 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.
- [21] M. Zilhão and F. Löffler, “An Introduction to the Einstein Toolkit,” *Int. J. Mod. Phys. A*, vol. 28, p. 1340014, 2013.

REFERENCES

- [22] “Repository of the changa code on github,” <https://github.com/N-BodyShop/changa>, accessed: 28.10.2023.
- [23] 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.
- [24] 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.
- [25] J. W. Wadsley, B. W. Keller, and T. R. Quinn, “Gasoline2: a modern smoothed particle hydrodynamics code,” *MNRAS*, vol. 471, no. 2, pp. 2357–2369, Oct. 2017.
- [26] J. G. Stadel, “Cosmological N-body simulations and their analysis,” Ph.D. dissertation, University of Washington, Seattle, Jan. 2001.