Scalable Parallel Astrophysical Codes for Exascale

Code Modules and Kernels

Deliverable number: D1.2

Version 1.1/1.1

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:	Karademir, Geray - LMU
Deliverable nature:	Report (R)
Dissemination level:	Public (PU)
Contractual Delivery Date:	31.12.2023
Actual Delivery Date	08.10.2024
Number of pages:	41
Keywords:	modules and kernels, optimization, scalability
Authors:	Luca Tornatore – INAF Marco Rossazza – UNITO Andrea Mignone – UNITO Stefano Truzzi – UNITO Benoît Commerçon – CNRS Joakim Rosdahl – CNRS Geray Karademir – LMU Georgios Doulis – GUF Khalil Pierre – GUF N. Shukla KU Leuven Robert Wissing – UiO Gino Perna – ES Eva Sciacca – INAF Giuliano Taffoni – INAF
Peer review:	Lubomir, Riha - IT4I@VSB Nitin Shukla - CINECA

History of Changes

Release	Date	Author, Organization	Description of changes
0.1	03.11.2023	Eva Sciacca, INAF	Started the document
0.7	12.12.2023	All authors	Version for reviews
0.81	13.12.2023	Geray Karademir, LMU	Introduction and conclusion
0.9	14.12.2023	Lubomir, IT4I@VSB	internal review
0.91	16.12.2023	Luca Tornatore, INAF	corrections following reviewers
0.92	16.12.2023	All authors	Integration as requested by reviewer 1 & 2
0.93	19.12.2023	Nitin Shukla, CINECA	Internal review
0.94	20.12.2023	Gino Perna , ES; Giuliano Taffoni, INAF	Conclusions and integrations
0.94	21.12.2023	All	Integration as requested by reviwer 2
1.00	21.12.2023	Gino Perna , ES; Giuliano Taffoni, INAF	final version assembled
1.1	16.09.2024	All code owners	Added paragraph on possible re-usage and inter-change of modules across the codes

Scalable Parallel Astrophysical Codes for Exascale

DISCLAIMER

Executive Summary

This document provides high-level descriptions for each code in the SPACE CoE. Within this code description the focus is on the main algorithms, which are used for each codes. These variety of algorithms range from Smoothed-Particle Hydrodynamics, Adaptive-Mesh-Refinements, different Runge-Kutta schemes to Particle in Cell algorithms.

In addition, the modules and kernels identified by each code for optimization are described as well as how the mini-apps will be created for each code. In short each code presented between one and six different regions. The selected regions address the main functions of the different codes, providing large potential for total wall time gains in the light of future optimisations. One aspect all codes and regions focus on is the optimization of the different communication and work load balancing schemes since this is a crucial component towards the exa-scale.

The identification of the modules and kernels is the first step towards the further development of the different codes. The identification and the creation of the mini-apps or alternative testing frameworks will create a benchmark for all future activities in WP2.

Contents

List of Figures

List of Tables

List of Acronyms

SPACE	Scalable Parallel Astrophysical Codes for Exascale
CoE	Centre of Excellence
CT	Constrained Transport
RK	Runge-Kutta
FV	Finite-Volume
HLL	Harte-Lax-van Leer
MHD	MagnetoHydrodynamics
RMHD	Relativistic Magneto-Hydrodynamics
SMP	Shared-Memory Multiprocessing

1 Introduction

The European codes in the Scalable Parallel Astrophysical Codes for Exascale (SPACE) Centre of Excellence (CoE) are among the most used and widespread codes in astrophysics. The SPACE CoE aims to enable these codes to exploit the pre-exascale systems funded by EuroHPC JU efficiently and effectively to enable their transition to the exascale. The goal of this deliverable is to describe the fundamental modules and kernels of the codes in the SPACE CoE for future activities. The selection of these modules and kernels was done by the responsible partners following the insights resulting from the profiling activity described in D2.1. The purpose of their selection is to identify modules and kernels, which are going to be extracted as mini-applications for optimization and co-design, and report on initial performance, scalability, and energy efficiency. The preferred methodology for this extraction will be to disentangle specific infrastructure functionalities and physics from the code.

The general procedure to create a mini app is to isolate and extract a particular module from the code and create a stand-alone application with this module. The mini-app then allows us to check an updated version of the module against the original to a) verify the results and b) compare its performance. However, the exact details of the methodology for each code will be defined by each code owner. A simplified view of the design of such a mini-app is shown in Fig. 1.

```
int main(){

    initalization/read-in();

    tstart = get_time();
    original_results = original_module();
    tend = get_time();
    time_original = tend - tstart;

    tstart = get_time();
    new_results = updated_module();
    tend = get_time();
    time_updated = tend-tstart;

    check_validity(new_results, original_results);

    print << "Change in runtime:" << time_updated/time_original;
}
```

Figure 1: Schematic view of a mini-app. After the initialization and the read-in at first, the old version of the module is executed and secondly with an updated version. Afterwards, the results are verified and, at last, the performances can be compared (in this simple example, only the total run-time is compared). This design will allow a simple and direct comparison of the updated version to its original.

While this approach would be ideal, due to the complexity of the codes involved this setup is not achievable for all codes. To mitigate this complex task an alternative approach is not to extract the kernel, but to remove not needed functionality the code. While this might not create an as clean mini-app as described above, the required functionality of the code will remain intact and allows for much easier access to the required modules. An additional benefit of this approach is that a complex and time consuming back-integration of the updated modules can be omitted since it is already embedded in the larger overall code structure.

The following chapters are organized as one per code, where the responsible code owners explain their code structure and present their selected modules and kernels.

2 OpenGADGET

2.1 High-level description of the code

OpenGADGET is a collisionless N-Body/Lagrangian cosmological code using the smoothed particle hydrodynamics (SPH) computational method to describe the motion of fluids in addition to the gravitational forces. The code allows running simulations in a full cosmological context, i.e. accounting for an expanding background and the presence of matter, both "dark" and baryonic (the ordinary matter), and dark energy. Although the full cosmological context is often the default choice, having a non-expanding background and a setup with only dark or baryonic matter is equally possible.

In addition to the gravitational problem, it also simulates the evolution of the physical properties of the baryonic matter subject to hydrodynamics and other physical effects such as radiative cooling, star formation, energy feedback, radiative transfer, magnetic fields and others that we collectively refer to as "extra physics".

The two most important algorithms in the code are the computation of the gravitational and hydrodynamical forces. These calculations are implemented in OpenGadget by three different algorithms:

ALG 1 **Gravity: Direct Summation + Tree Multipolar expansion**; the particles are organized in a highly-optimized Barnes&Hutt Oct-Tree structure that ensures a $N_p\, logN_p$ retrieval time, where N_p is the number of particles. Being the gravitational force additive, the resultant force to every particle is due to the summation of every other particle. However, since performing a direct summation would result in a prohibitive N_p^2 scaling, only the closest particles' contribution is accounted for particle-wise. The gravitational force due to more distant particles is calculated via the multi-polar expansion of the tree nodes that contain those particles; in fact, distant enough particles can be represented effectively as a single particle of mass equal to the entire mass, with a relative accuracy that depends on the ratio between the spatial extent of the particles distribution and the distance of their centre-of-mass from the target particles. The acceptable accuracy is a code parameter that determines the precise definition of "close" and "distant".

ALG 2 **Gravity: Particle-Mesh**; "very distant" particles could be treated separately, accounting for the average gravitational field due to large-scale matter distribution that results from the spreading of the particles' mass onto a grid. The resulting density distribution is used as a known term of the Poisson equation $\Delta\Phi = 4\pi G\rho$, which is solved via FFT on the mentioned grid. The calculation of $\nabla\Phi$ then conveys the contribution to the gravitational force from the large scales. Since the evolution of the large-scale averaged density field is slower than the evolution of the small-scale field, the update frequency is also minor, and that increases the algorithmic advantage of mixing algorithms ALG 1 and ALG 2 for "close" and "distant" particles.

ALG 3 **Hydro: Smoothed-Particles Hydrodynamics (SPH)**; the hydrodynamics forces among baryonic gas particles are calculated via a state-of-the-art modern implementation of the SPH technique (presented initially by Gingold & Monaghan, 1977).

Every gas particle is considered a sample of the gas distribution in a volume determined by a kernel function $W()$ with compact support centered on the particle itself, whose radius h is called the "smoothing length". This radius is determined by the fact that a given number of other gas particles, referred to as "the neighbouring particles", are found within the volume. The local density of matter is then interpolated locally using the kernel W as a weighting function.

Each neighbour contributes to the local density $\rho_{i,gas}$ at the position of every particle i as $m_j \times W(h, r_{ij}/h$, where r_{ij} is the distance between the particle i and its neighbour particle j. Then, summing over all the neighbours j, $\rho_{i,gas} = \sum_j m_j \times W(h, r_{ij}/h)$.

In turn, the local density $\rho_{i,gas}$ of each particle is used in the interpolation of all the other baryonic physical fields f_b, that is estimated at the position of particle i as $f_b(x_i) = \sum_j f_b(x_j)/\rho_j \times W(h, r_{ij}/h)$.

This algorithm naturally couples with the tree-based ALG 1 since the same API used to retrieve neighbors to estimate the gravitational force is used to determine the local radius h_i for every particle i.

The high-level abstraction of the Gadget code is depicted in the algorithmic view reported in Figure 2. In simple terms, the code consists of two parts. First is the initialisation: reading initial conditions, reading the parameter file, and initializing the required physical modules. Second, an "infinite" time loop that is the body of the code. Each of these time steps consists of a series of subsequent calculations: the estimate of the gravitation acceleration, the hydrodynamical acceleration and the "extra-physics" (with which we indicate all the physical processes for baryons, like the radiative cooling, star-formation, stellar feedback, black-holes feedback, ...). In addition, the domain decomposition, which amounts to the re-distribution of particles to keep the work as balanced as possible, is performed depending on evaluating some global and local conditions.

All local physical processes modeled in the code rely on this general structure. Either they are exquisitely local, requiring computations based only on the properties of a single particle, or they need to evaluate the physical conditions in a limited space region around a target particle's position. For example, the radiative cooling of a single particle depends upon only its physical properties (which, in turn, have been shaped beforehand by other non-local processes), like the density, the temperature, the chemical composition, and possibly an external radiation field. In contrast, the accretion of gas by a black hole requires that it collects the overall physical information around its position by scanning its neighborhood. In the second case, the same routines are used as in the gravity tree, the gas density and the hydrodynamics to find the neighbour particles of a target particle and their values of the needed physical quantities. Finally, there is a last fundamental feature of a massively parallel code: the domain decomposition, which distributes the particles across the different tasks and is the source of the constant re-balancing of the parallel workload

The distribution in OpenGADGET is based on space decomposition through a space-filling Peano-Hilbert curve. After the Peano index of each particle has been calculated and the particles are cross-sorted by their indexes, the entire computational domain can be seen as a one-dimensional set; the segments of this distribution, which correspond to connected space sub-volumes, are used to determine the domain of every MPI task. The computational cost of each segment is then defined as the sum of the computational cost of all the particles within that segment. By iteratively adjusting the segments' boundaries, an optimal work balance can be achieved within the memory constraints. The code favors the work balance instead of the load balance; some MPI tasks may have a higher fraction of low-cost particles, while others may have a smaller quantity of more costly particles within the memory constraints. The minimum number N_D of segments is equal to the number N_T of MPI tasks; however, a finer balance is achieved when a multiple of that is adopted, $N_D = \eta N_T$ so that the computational domain of every task consists of η segments of the one-dimensional distribution based on the Peano-Hilbert curve. Sorting the $\eta \times N_T$ segments by their computational cost and round-robin assigning to different MPI tasks the pairs made of the (`most-intense`, `less-intese`) among the unassigned segments, a better balance among tasks work distribution can be attained.

2.2 Kernels targeted for optimization

Given the above description, we conclude that the essential kernels are the following:

A) Tree Building - all the routines related to the building of the OctTree;

B) Tree Walking - all the routines related to the solution of the k-Nearest Neighbour (kNN) problem, or the approximate kNN problem;

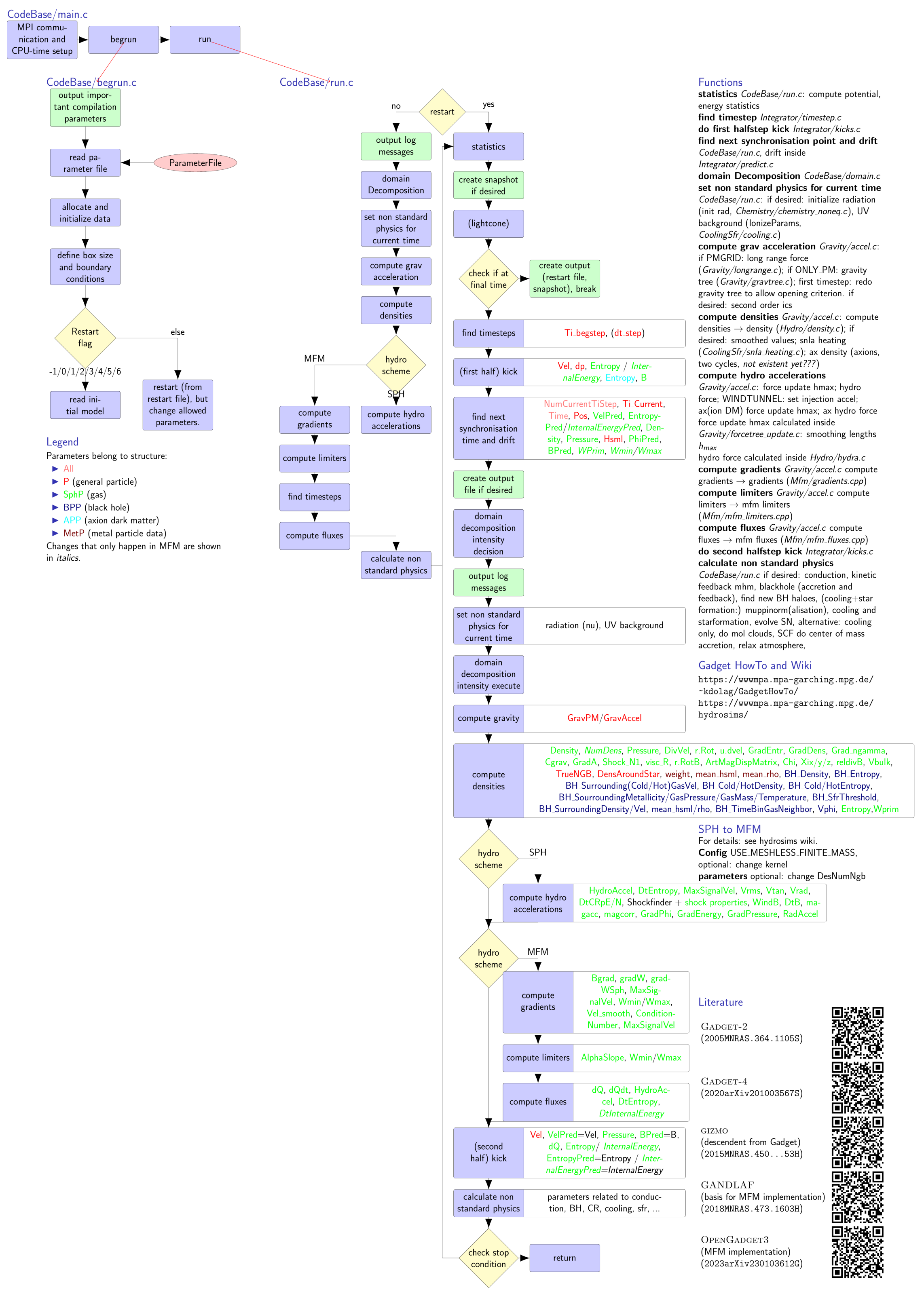
C) Domain Decomposition - all the routines related to the individuation of computational domains and the communication patterns used to exchange data;

D) Gravity Tree - the calculation of the direct and tree-estimated gravitational forces among the particles;

E) Density Loop - the loop for the local density estimation.

We remind the readers that, as detailed in D2.1, we have concentrated the performance tests on the high-level sections responsible for most of the computation and communication. A simplified view, including the profiling regions, is shown in Figure 3. Following the activity of deliverable D2.1, the fundamental code aspects that need to be stressed and improved are as follows:

- the core-efficiency; we need to improve the code's capability of exploiting the instruction-level parallelism (ILP) in terms of both achieving a larger instructions per Cycle (IPC) and a larger vectorization (especially for the larger vector sizes); that may need some re-designing of the data structures to expose better opportunities for vectorization and exploitation of out-of-order capabilities.
- OpenMP scalability; where present, the thread-level scalability exhibits a good/very good behaviour, although there is room for improvement. However, two important regions need significant effort, namely the domain decomposition and the extra-physics, in which the OpenMP support is either not fully developed


CodeBase/main.c
MPI communication and CPU-time setup
begrun
run
CodeBase/begrun.c
output important compilation parameters
read parameter file
ParameterFile
allocate and initialize data
define box size and boundary conditions
Restart flag
else
-1/0/1/2/3/4/5/6
read initial model
restart (from restart file), but change allowed parameters.
CodeBase/run.c
restart
no
yes
output log messages
domain Decomposition
set non standard physics for current time
compute grav acceleration
compute densities
hydro scheme
MFM
SPH
compute gradients
compute limiters
find timesteps
compute fluxes
compute hydro accelerations
calculate non standard physics
statistics
create snapshot if desired
(lightcone)
check if at final time
create output (restart file, snapshot), break
find timesteps
Ti_begstep, (dt_step)
(first half) kick
Vel, dp, Entropy / *InternalEnergy*, Entropy, B
find next synchronisation time and drift
NumCurrentTiStep, Ti_Current, Time, Pos, VelPred, EntropyPred/*InternalEnergyPred*, Density, Pressure, Hsml, PhiPred, BPred, *WPrim*, *Wmin/Wmax*
create output file if desired
domain decomposition intensity decision
output log messages
set non standard physics for current time
radiation (nu), UV background
domain decomposition intensity execute
compute gravity
GravPM/GravAccel
compute densities
Density, *NumDens*, Pressure, DivVel, r.Rot, u.dvel, GradEntr, GradDens, Grad_ngamma, Cgrav, GradA, Shock_N1, visc_R, r.RotB, ArtMagDispMatrix, Chi, Xix/y/z, reldivB, Vbulk, TrueNGB, DensAroundStar, weight, mean_hsml, mean_rho, BH_Density, BH_Entropy, BH_Surrounding(Cold/Hot)GasVel, BH_Cold/HotDensity, BH_Cold/HotEntropy, BH_SourroundingMetallicity/GasPressure/GasMass/Temperature, BH_SfrThreshold, BH_SurroundingDensity/Vel, mean_hsml/rho, BH_TimeBinGasNeighbor, Vphi, Entropy,Wprim
hydro scheme
SPH
compute hydro accelerations
HydroAccel, DtEntropy, MaxSignalVel, Vrms, Vtan, Vrad, DtCRpE/N, Shockfinder + shock properties, WindB, DtB, magacc, magcorr, GradPhi, GradEnergy, GradPressure, RadAccel
hydro scheme
MFM
compute gradients
Bgrad, gradW, gradWSph, MaxSignalVel, Wmin/Wmax, Vel_smooth, ConditionNumber, MaxSignalVel
compute limiters
AlphaSlope, Wmin/Wmax
compute fluxes
dQ, dQdt, HydroAccel, DtEntropy, *DtInternalEnergy*
(second half) kick
Vel, VelPred=Vel, Pressure, BPred=B, dQ, Entropy/ *InternalEnergy*, EntropyPred=Entropy / *InternalEnergyPred*=*InternalEnergy*
calculate non standard physics
parameters related to conduction, BH, CR, cooling, sfr, ...
check stop condition
return

Legend

Parameters belong to structure:

- All
- P (general particle)
- SphP (gas)
- BPP (black hole)
- APP (axion dark matter)
- MetP (metal particle data)

Changes that only happen in MFM are shown in *italics*.

Functions

statistics *CodeBase/run.c*: compute potential, energy statistics
find timestep *Integrator/timestep.c*
do first halfstep kick *Integrator/kicks.c*
find next synchronisation point and drift *CodeBase/run.c*, drift inside *Integrator/predict.c*
domain Decomposition *CodeBase/domain.c*
set non standard physics for current time *CodeBase/run.c*: if desired: initialize radiation (init rad, *Chemistry/chemistry_noneq.c*), UV background (IonizeParams, *CoolingSfr/cooling.c*)
compute grav acceleration *Gravity/accel.c*: if PMGRID: long range force (*Gravity/longrange.c*); if ONLY_PM: gravity tree (*Gravity/gravtree.c*); first timestep: redo gravity tree to allow opening criterion. if desired: second order ics
compute densities *Gravity/accel.c*: compute densities → density (*Hydro/density.c*); if desired: smoothed values; snIa heating (*CoolingSfr/snIa_heating.c*); ax density (axions, two cycles, *not existent yet???*)
compute hydro accelerations *Gravity/accel.c*: force update hmax; hydro force; WINDTUNNEL: set injection accel; ax(ion DM) force update hmax; ax hydro force force update hmax calculated inside *Gravity/forcetree_update.c*: smoothing lengths h_{max}
hydro force calculated inside *Hydro/hydra.c*
compute gradients *Gravity/accel.c* compute gradients → gradients (*Mfm/gradients.cpp*)
compute limiters *Gravity/accel.c* compute limiters → mfm limiters (*Mfm/mfm_limiters.cpp*)
compute fluxes *Gravity/accel.c* compute fluxes → mfm fluxes (*Mfm/mfm_fluxes.cpp*)
do second halfstep kick *Integrator/kicks.c*
calculate non standard physics *CodeBase/run.c* if desired: conduction, kinetic feedback mhm, blackhole (accretion and feedback), find new BH haloes, (cooling+star formation:) muppinorm(alisation), cooling and starformation, evolve SN, alternative: cooling only, do mol clouds, SCF do center of mass accretion, relax atmosphere,

Gadget HowTo and Wiki

`https://wwwmpa.mpa-garching.mpg.de/~kdolag/GadgetHowTo/`
`https://wwwmpa.mpa-garching.mpg.de/hydrosims/`

SPH to MFM

For details: see hydrosims wiki.
Config USE_MESHLESS_FINITE_MASS, optional: change kernel
parameters optional: change DesNumNgb

Literature

GADGET-2
(2005MNRAS.364.1105S)

GADGET-4
(2020arXiv201003567S)

GIZMO
(descendent from Gadget)
(2015MNRAS.450...53H)

GANDLAF
(basis for MFM implementation)
(2018MNRAS.473.1603H)

OPENGADGET3
(MFM implementation)
(2023arXiv230103612G)

Figure 2: General view of OpenGadget's workflow.

or absent.

- MPI scalability; the profiling has shown that the communication efficiency and the parallel load balance can be quite improved in several regions.

  A further inquiry is needed to assess the opportunity for either a re-design of the parallelization in general or local refinements and improvements.

- overlap of communication and computation; that is currently achieved with different levels of efficiency only where the GPU support is developed. However, a general redesign to enhance this aspect is to be considered.

- The profiling done for D2.1 showed that the domain decomposition, dominated by MPI communications, has significant limitations in the strong scaling.

  The reason for low communication efficiency is that over 50% of the run time is spent on an `MPI_Waitall`. In addition, the OpenMP threads are not utilized appropriately, resulting in the limited OpenMP load balance seen during the profiling. This insufficient threading results in poor performance, where the other CPU cores are idle during the MPI operations, done only by the master thread.
  Since domain decomposition is responsible for distributing the workload across the different tasks and gets called at almost every step, improving this kernel would significantly improve the code overall. Therefore, this kernel will need a re-designing of its architecture that includes an efficient threding and a re-design of the calculation-communication patterns. The noticeable improvement is to enhance the work-sharing among the threads so that the master thread can communicate small data chunks while the other threads are processing. A carefully designed new scheme will also cure the large amount of time spent in the MPI Waitall. In particular, the implementation of a stencil decomposition would bring great benefit to this kernel.

According to a best-effort strategy, we also plan to explore the following.

1. the feasibility of introducing a native NUMA AWARENESS so that the code can *(i)* exploit more efficiently the shared memory at the node level by leveraging the DMA possibilities offered by the MPI standard, and *(ii)* reduce the inter-node communication surface by appointing one MPI task per node as a node master that manages the message exchange with other levels. The coupling and enhanced OpenMP parallelization and an explicit NUMA awareness would allow more efficient communication/synchronization and computation overlapping, especially in the domain decomposition and the neighbours' search.

2. the impact of different data models and structures layout, especially on exploiting the CPU's ILP capabilities, the vectorisation, and the communication patterns.

Since operating on the entire code would be difficult and isolating the single kernels requires some coding effort, *we have chosen to have a stripped-down version of the code*, cleaned by all the extra physical modules and experimental options.
Hence, **we obtained a mini-app that includes an essential version of the kernels listed above**, and that allows us to both *(i)* intervene in each of them separately and *(ii)* to act on the general data structures harmonizing their usage throughout all the kernels to test the impact of different data models.

```c
int main(){

    do_initialisation();

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

        write_snapshot_if_desired();

        // exit code when max time is reached
        if ( Time >= maxTime ){
            write_snapshot();
            break ;
        }

        do_first_halfstep_kick();

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

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

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

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

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

        do_second_halfstep_kick();

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

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

Figure 3: Simplified structure of the main function for OpenGadget. Here, the code splitting into the initialization and the main time loop can be clearly seen.

3 PLUTO

PLUTO [1, 2] is designed to integrate a general system of conservation laws that we write as:

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

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

The explicit form of U and $\mathcal{T}(U)$ depends on the physics modules been selected. PLUTO supports 5 different physics modules corresponding to different conservation laws:

- HD (HydroDynamics): it implements the Euler equation of gas-dynamics evolving density, momentum and energy (e.g. $U = \{\rho, \rho\mathbf{v}, E\}$);
- MagnetoHydrodynamics (MHD) (Magneto-HydroDynamics): this implements the ideal or resistive equations of magnetohydrodynamics. In addition to the variables already included in the HD module, it also adds the magnetic field;
- RHD (Relativistic HD): (Special) Relativistic extensioon of the Euler equations;
- Relativistic Magneto-Hydrodynamics (RMHD) (Relativistic MHD): (Special) Relativistic extensioon of the ideal MHD equations;
- ResRMHD (Resistive Relativistic MHD): (Special) Relativistic extensioon of the resistive MHD equations. In this case, both magnetic and electric fields are included and updated in time.

The divergence-free condition can be controlled either by using the divergence-cleaning technique or by the Constrained Transport (CT) approach. Since the latter is a more sophisticated and articulated method, we will describe the high-level description of the code according to the latter framework.

While the components of U are the primary variables being updated, fluxes are computed more conveniently using a different set of physical quantities, which we take as the primitive vector V. Numerical integration of the conservation law above is achieved through shock-capturing schemes using the Finite-Volume (FV) formalism where volume averages evolve over time. Generally speaking, these methods comprise three steps: a reconstruction routine followed by the solution of Riemann problems at the zone edges and a final evolution stage. In PLUTO, this sequence of steps provides the necessary infrastructure of the code; see the schematic diagram in (4): first, volume averages U are more conveniently mapped into primitive quantities V. The left and right states $V^{L,R}_{i+1/2}$ are then reconstructed inside each cell:

$$V^L_{i+1/2} = \mathcal{R}^+(V_i)\,, \qquad V^R_{i-1/2} = \mathcal{R}^-(V_i) \tag{2}$$

where $\mathcal{R}^\pm$ is the reconstructor operator in the positive (+) or negative (−) direction, i stands for the cell center while $i \pm 1/2$ identify the interfaces. Operations along the y- and z- directions are obtained in a similar way. At zone interfaces, a Riemann problem is then solved between adjacent L/R states to obtain a stable numerical flux $F^{(d)}_{i+1/2}$, where $d = x, y, z$. The solution is finally advanced in time in a conservative fashion, so that zone-centered variables are updated through

$$U^{n+1}_c = U^n_c - \frac{\Delta t}{\Delta V}\sum_d \int \mathbf{F}^{(d)} \cdot d\mathbf{S}_d + \Delta t S\,, \tag{3}$$

where $c \equiv (i, j, k)$, ΔV is the cell volume. For staggered variables (used by the CT approach), the update step relies on a discrete form of the Stokes theorem:

$$B^{n+1}_d = B^n_d - \frac{\Delta t}{\Delta S_d}\oint \mathbf{E} \cdot d\mathbf{l}\,. \tag{4}$$

where, using a shorthand notation, d indicates that the surface average of a generic magnetic field component located at zone faces (e.g. B_d with $d = y$ means $B_{y,i,j+1/2,k}$). Likewise, the electric field line integral is done on the boundary of the corresponding face.

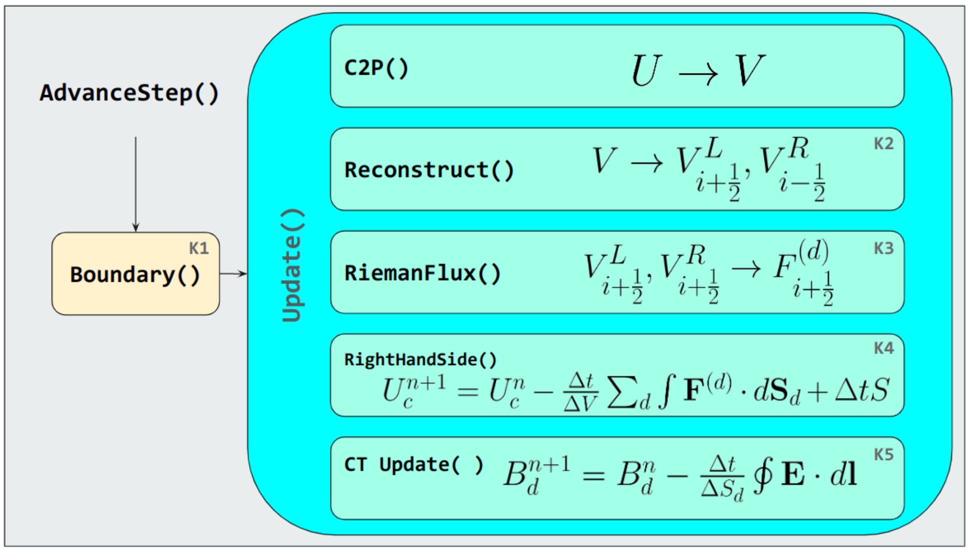

AdvanceStep()
Boundary()
K1
Update()
C2P()
$U \to V$
Reconstruct()
$V \to V^L_{i+\frac{1}{2}}, V^R_{i-\frac{1}{2}}$
K2
RiemanFlux()
$V^L_{i+\frac{1}{2}}, V^R_{i+\frac{1}{2}} \to F^{(d)}_{i+\frac{1}{2}}$
K3
RightHandSide()
$U^{n+1}_c = U^n_c - \frac{\Delta t}{\Delta V}\sum_d \int \mathbf{F}^{(d)} \cdot d\mathbf{S}_d + \Delta t S$
K4
CT Update( )
$B^{n+1}_d = B^n_d - \frac{\Delta t}{\Delta S_d}\oint \mathbf{E} \cdot d\mathbf{l}$
K5

Figure 4: Diagram of the Reconstruct-solve-average (RSA) strategy

3.1 High-level description of the code

The computational expensive part of the code is enclosed in the loop that cycles as long as either the final simulation time has been reached or the maximum number of steps has been reached. The main function executed in the body of the while loop is `AdvanceStep()` which advances the solution array to the next time-level (Eq. 3 and Eq. 4), using a Runge-Kutta (RK) time stepping method of the desired order (at present orders $1, 2, 3$ and 4 are available). For a RK method of order n, the `AdvanceStep()` typically consists of n stages during which the `Boundary()` and `UpdateStage()` are called multiple (that is, n) times. For the 3rd-order RK, as in our selected case, the `AdvanceStep()` function consists of 3 similar stages.

As already mentioned, a single RK stage implements a call to the `Boundary()` function followed by a call to `UpdateStage()`, which evaluates the right-hand sides of Eq. (3) and (4). More specifically, during `UpdateStage()`, one first reconstructs primitive variables (Eq. 2) in order to obtain left and right states (`Reconstruct()` function). These states are then fed into the Riemann solver (`RiemannFlux()` function) which construct the flux function needed to compute the integrals on the right-hand side of Eq. (3). The interface fluxes are then collected together, including also the source term, in the `RightHandSide()` function which does the summation in Eq. (3). Finally, the information obtained during `RiemannFlux()` is further re-processed to construct the electromotive force in Eq. (4) during the `CT_Update()` function. These operations are extended to every dimension, and the high-level structure of the code is shown in 5.

3.2 Kernels targeted for optimization

3.2.1 Kernel 1 - Boundary Conditions (`Boundary()` function)

This is a fundamental routine as it contains basically all the process communications and it is called once per stage. The `Boundary()` function sets both internal (that is, inter-processor) and physical boundary conditions on all of the sides of the computational domain. Its basic purpose is to fill ghost zones for both cell-centred and face-centred data arrays. The type of physical boundary condition at the leftmost or rightmost side of a given dimensions is set by the user at runtime. Possibile options are periodic, outflow (zero-gradient), reflective, axisymmetric, equatorial symmetric or user-defined. For our chosen test, outflow boundary conditions have been selected. Conversely, when a processor does not abut a physical boundary, ghost zone values are exchanged between neighbour processors. This step is repeated dimension by dimension for both cell-centered and staggered data arrays. Currently, our implementation relies on the `SendRecv()` functions of the Message Passing Interface (MPI) library.

```
while (last_step!=1){

    AdvanceStep(){

        /* Predictor step (Euler, RK2, RK3)*/
        Boundary();
        UpdateStage(){

          /* IDIR */
          Reconstruct();
          RiemannFlux();
          RightHandSide();

           /* JDIR */
          Reconstruct();
          RiemannFlux();
          RightHandSide();

           /* KDIR */
          Reconstruct();
          RiemannFlux();
          RightHandSide();

          CT_Update();
        }

        /* Stage #2 (RK2, RK3) */
        Boundary();
        UpdateStage():

        /* Stage #3 (RK3) */
        Boundary();
        UpdateStage():
    }
}
```

Figure 5: Simplified high-level code structure of PLUTO.

3.2.2 Kernel 2 - States reconstruction (`Reconstruct()` function)

At each RK stage, this function is executed once per every dimension (therefore 9 times for a 3D problem using a third-order RK method), and it calculates the left / right primitive states (see Eq. 2) at zone interfaces $i+1/2$ using a suitable reconstruction method which we denote by $\mathcal{R}^{\pm}(.)$. The reconstruction is typically based on piecewise polynomial interpolation subject to monotonicity constraints in order to avoid the Gibbs phenomena in the presence of discontinuities or steep gradients. In alternative to primitive variables, characteristic variables (i.e., obtained by projecting primitive variables onto the eigenvectors of the underlying equations) may also be employed at the cost of substantially increasing the operation count. These methods are essentially one-dimensional and employ information from adjacent zones. While the code offers different reconstruction methods - such as, Linear, Piecewise Parabolic, Weighted Essentially non Oscillatory (WENO), Monotonicity Preserving (MP) algorithm, we select linear (and later on fifth-order WENO and MP reconstruction) for optimization.

3.2.3 Kernel 3 - Riemann Solver (`RiemannFlux()` function)

The Riemann solver represents the most complex single kernel and computationally costly section of the code. This function computes the flux function at a zone interface and, in 3D, it is called 3 times per stage and, for a single step, $3 \times n$ times, where n is the temporal order of the scheme. On input, `RiemannFlux()` takes the left and right primitive states at a zone edge (say $i + 1/2$ or $j + 1/2$ or $k + 1/2$) previously obtained during the reconstruction kernel 3.2.2. PLUTO allows the user to choose different solvers such as the Harte-Lax-van Leer (HLL) Riemann solver, FORCE, HLLC, HLLD or Roe (see, for instance, [3, 4, 5]), although some choices may not be available for all the physics modules. On output, it returns the Godunov flux at the corresponding interface $i + 1/2$.

Also, during this step, we compute and store additional information which will serve during subsequent operations, most notably, i) the maximum wave propagation speed ($\lambda_{\max}$) for explicit time step computation and, in the case of CT scheme, ii) interpolation coefficients and electric field components needed to compute the electro-motive force at zone edges. We point out that different Riemann solvers have, in general, different computational cost and may differ by a factor 3 or even more.

3.2.4 Kernel 4 - Right Hand Side (`RightHandSide()` function)

This routine evaluates the right hand side for cell-centered conservative variables, as depicted formally in Eq. (3). It is invoked 3 times (one per direction) per stage. Interface fluxes are included one direction at a time, according to

$$R = -\frac{\Delta t}{\Delta V}(A_{i+1/2}F_{i+1/2} - A_{i-1/2}F_{i-1/2}) + \Delta t S_i \tag{5}$$

where $A_{i\pm1/2}$ are the right and left interface areas, $F_{i\pm1/2}$ are the Godunov fluxes previously obtained with a Riemann solver, Δt is the time step while S_i is the source term including geometrical contribution and body forces in general.

As already mentioned, during this step, only contributions to density, momentum and energy are effectively considered. The right hand side for staggered magnetic fields is computed later in a the constrained transport (CT) kernel.

3.2.5 Kernel 5 - Constrain transport update

This kernel builds the electric field contribution to the induction equation (right hand side of Eq. 4 and it consists of a discrete version of Stokes theorem.

This kernel can be split essentially into 2 contributions. During the 1st one, information collected during the Riemann solver calls will be brought together to evaluate stable and properly upwinded electric field components at cell edges (see [6]). This kernel is intrinsically multidimensional and cannot be reduced to a sequence of 1D operators. During the 2nd one, the explicit right hand side (analogous to Eq. 5) is constructed using the electric field just evaluated during the 1st contribution:

$$
\begin{aligned}
B_x^{n+1} - B_x^n &= -\frac{\Delta t}{\Delta y \Delta z}\oint(-E^y dy + E^z dz) \approx -\Delta t\left[-\frac{(E^y)_{z_+} - (E^y)_{z_-}}{\Delta z} + \frac{(E^z)_{y_+} - (E^z)_{y_-}}{\Delta y}\right] \\
B_y^{n+1} - B_y^n &= -\frac{\Delta t}{\Delta x \Delta z}\oint(E^x dx - E^z dz) \approx -\Delta t\left[\frac{(E^x)_{z_+} - (E^x)_{z_-}}{\Delta z} - \frac{(E^z)_{x_+} - (E^z)_{x_-}}{\Delta x}\right] \\
B_z^{n+1} - B_z^n &= -\frac{\Delta t}{\Delta x \Delta y}\oint(-E^x dx + E^y dy) \approx -\Delta t\left[-\frac{(E^x)_{y_+} - (E^x)_{y_-}}{\Delta y} + \frac{(E^y)_{x_+} - (E^y)_{x_-}}{\Delta x}\right] .
\end{aligned} \tag{6}
$$

3.3 Mini-apps description

The findings from document D2.1 underscored the computational intensity of these kernels, indicating their potential suitability for GPU vectorization. Our focus in the future will be the GPU-porting, and to validate performance optimization, we opted to address the entire code. This choice is driven by the convenience of working directly on the code and the complexity associated with isolating each kernel into individual applications. Employing profilers and/or measuring execution times will streamline the evaluation of our progress. In particular, Nsight_System (integrated with NVTX) is a tool that we are becoming familiar with and that is proving to be very useful in analyzing the performance of each kernel on the GPU.

4 BHAC

The Black Hole Accretion Code (BHAC) [7, 8, 9, 10] is a multidimensional General Relativistic Magnetohydrodynamics (GRMHD) code that solves the equations of ideal GRMHD in one, two or three dimensions in order to perform (magneto)hydrodynamical simulations of accretion flows onto compact objects in arbitrary stationary space-times (Cowling approximation) using an efficient block based approach. BHAC is build upon the MPI-Adaptive Mesh Refinement-Versatile Advection Code (MPI-AMRVAC). MPI-AMRVAC [11, 12] is a parallel adaptive mesh refinement framework aimed at solving (primarily hyperbolic) partial differential equations (PDEs) by a number of different numerical schemes. The framework supports 1D to 3D simulations, in a number of different geometries (Cartesian, cylindrical, spherical). MPI-AMRVAC is written in Fortran 90 and uses MPI for parallelization. BHAC is second-order accurate and employs a variety of multi-step Runge-Kutta schemes to temporally integrate the cell-average of the conserved variables through the computational grid. For a more detailed description of BHAC see Sec. 6 of [13].

4.1 High-level description of the code

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

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

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

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

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

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

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

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

4.2 Kernels targeted for optimization

4.2.1 Kernel 1 - Primitive reconstruction

The nonlinear inversion of the conservatives to recover the primitive variables at every time-step is the Achilles heel of any relativistic (M)HD code and requires the development of sophisticated schemes with multiple backup strategies. BHAC is no different from other GRMHD codes when it comes to deal with this issue: Two primary inversion strategies are available in BHAC supplemented by a third one in highly magnetized regions. The first two inversion methods can fail under different circumstances, and thus the one can act as a backup strategy for the other. Typically, we first attempt one of them and switch to the other method when no convergence is found. If both fail to converge, then the third one can act as the next layer of backup. Clearly, depending on the details of our setup, the primitive reconstruction process can become the most resource-demanding part of our simulations. This could possibly explain the findings of the profiling performed in deliverable D2.1, where the main cause that limits parallel efficiency is the poor load balance of the region that contains the primitive recovery module. It could be possible that the processing units dealing with the highly magnetized areas may require more effort to complete the primitive recovery process compared to the ones dealing with the weakly magnetized areas. This workload imbalance results in decreased efficiency and performance of the code.

```
subroutine primitive()

    ! Transform conservative variables into primitive ones
    ! (D,S,tau,B)-->(rho,v,p,B,lfac,xi)
    ...
    call getaux()  ! calculate lorentz factor and (auxiliary variable) xi from conservatives
    only

    call primitiven()  ! avoid recursive calling by small values
    ...
  end subroutine primitive
```

The conservative to primitive computations could greatly benefit from a GPU offloading, but this task depends crucially on the possibility of porting the parallel adaptive mesh refinement framework AMRVAC, on top of which BHAC is build, to GPU.

4.3 Mini-apps description

Because of the structure of BHAC and the fact that, in order to reconstruct the primitives at each time-step, we need the corresponding values of the conservatives, it is not possible to isolate the primitive() module from the rest of the code. Therefore, the mini-app will be designed to monitor primitive() through its parent module `advance()`. In this way `primitive()` will be able to access the updated values of the conservatives at every time-step guaranteeing the accuracy of our results. For each call of `advance()`, corresponding to a full time step, `primitive()` is called twice, at each intermediate time step, because of the predictor-corrector method we are using for the time-integration.

5 ChaNGa

5.1 High-level description of the code

ChaNGa [14][15][16] is an N-body and Smoothed Particle Magneto-HydroDynamics (SPMHD) code which is used to study a wide array of astrophysical systems. While the gravity and SPMHD algorithms are based on the gasoline [17] and pkdgrav [18] codes, the unique feature of ChaNGa is it's implementation of the Charm++ framework, which enables highly efficient parallel scaling. Charm++ employs overdecomposition to achieve this. That is, divide the work into many more pieces (chares/tree pieces) than you have processors and let the Charm++ runtime system load balance by appropriately assigning pieces to real processors (see Figure 6 for an overview). During runtime, Charm++ applies dynamic load rebalancing strategies to determine which tree pieces should be migrated to new processors for better load balance. In addition, the SMP mode of Charm++ leverages the shared-memory systems found in high-performance computing environments. Within an SMP process/node, one thread is assigned as the communication thread, responsible for internode communication, while the remaining threads act as worker threads in charge of the processing elements. Multiple Shared-Memory Multiprocessing (SMP) processes can be initiated per network node, the number of which should be based on the CPU architecture and communication load. Charm++ also provides support to execute CUDA kernels on the GPU asynchronously and to manage data transfers between the CPU and GPU.

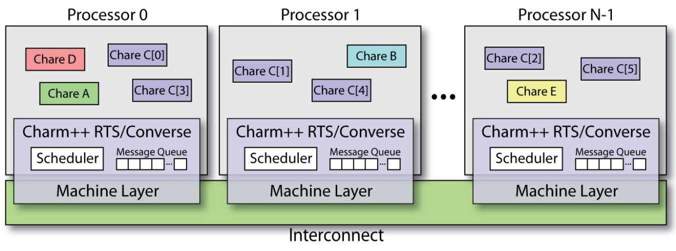


Figure 6: System's View of a Charm++ Application

The central feature of ChaNGa is a tree-based gravity solver using a variant of the Barnes-Hut algorithm. This solver is combined with several other features which include: gravitational interaction between dark matter, stars and gas, Ewald summation to handle cosmological boundary conditions, multi-time stepping, hydrodynamics, magnetic field and several subgrid physics such as, star formation, feedback from supernova, stellar winds, black holes and the radiative cooling of gas.

The five major physics algorithms within ChaNGa are in order of computation and communication load (though this can depend on simulation conditions):

ALG 1 **Gravity: Barnes-Hut tree**; To compute the gravitational force at any point in space given a specific mass distribution, we want to solve Poisson's equation for gravity:

$$\nabla^2 \Phi = 4\pi G \int f dv = 4\pi G \rho \tag{7}$$

Here ∇^2 is the Laplace operator, Φ is the gravitational potential, G is the gravitational constant, ρ is the mass density at each point in space. In an N-body code, the continuous mass distribution is discretized into a finite number of particles. Instead of directly calculating the gravitational influence that each particle has on each other, the Barnes-Hut tree approximation is used, where distant particles are combined into the center of mass of tree cells. This allows one to separate the force calculation into subdivisions of long-range components (tree/multipole) and a short-range component (direct) to the degree which is required by the desired force accuracy. For ChaNGa the global tree is then split into N number of tree pieces (depending on the given overdecomposition) and distributed among all the processors (including different nodes). Particles, or more specifically, groups of particles contained in the tree leaves (often referred to as "buckets"), traverse the tree to compute the gravitational force. This is separated into a local tree traversal part (tree pieces within an SMP process) and a remote tree traversal part (tree pieces in another SMP process).

ALG 2 **Smoothed-Particle Magnetohydrodynamics (SPMHD)**; The governing equations of smoothed particle magneto-hydrodynamics is given by:

$$\frac{dv_a^i}{dt} = \sum_b \frac{m_b}{\rho_a \rho_b} \left(S_a^{ij} + S_b^{ij} \right) \nabla_a^j \overline{W_{ab}} + F_{v,diss}, \tag{8}$$

$$\frac{du_a}{dt} = \frac{P_a}{\rho_a} \sum_b \frac{m_b}{\rho_b} (v_b - v_a) \cdot \nabla_a \overline{W_{ab}} + F_{u,diss}, \tag{9}$$

$$\frac{dB_a}{dt} = \sum_b \frac{m_b}{\rho_b} \left[B_a (v_{ab} \cdot \nabla_a \overline{W_{ab}}) - v_{ab} (B_a \cdot \nabla_a \overline{W_{ab}}) \right] + F_{B,diss} + F_{B,clean} \tag{10}$$

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

$$\rho_a = \sum_b^{N_{smooth}} m_b W_{ab} \tag{11}$$

The term N_{smooth} represents the number of neighbours that each particle uses for interpolation and is user-defined (default 64). To find the neighbours a k-nearest neighbors tree traversal is performed.

ALG 3 **Feedback + Star formation**; Feedback in galaxies refers to the processes by which the energy and matter output from stars and supermassive black holes influence the surrounding interstellar medium (ISM) and intergalactic medium (IGM), thereby regulating further star formation and galaxy evolution. This feedback can be broadly categorized into two types: stellar feedback and AGN feedback. To distribute the energy output generated from both types of feedback, the tree build is leveraged and a similar k-nearest neighbors method is performed as for the SPMHD code. The star formation is based on a stellar module that determines when gas turns into stars/star clusters. It requires local gas properties and the formation of star particles and deletion of gas particles.

ALG 4 **Cosmological boundary/Ewald summation**; The key idea here is to replicate the conditions of an infinite universe within a finite simulation box. One common approach is to use periodic boundary conditions. In this setup, the simulation box is treated as a cell in a tessellated infinite universe. Objects that exit one side of the simulation box are re-introduced on the opposite side. This creates a universe that is effectively infinite for the purposes of the simulation, allowing the study of large-scale structure without the need for an impractically large computational domain. Ewald summation is used to efficiently compute long-range interactions in periodic systems. Ewald summation works by dividing the potential into short-range and long-range components. The short-range component is computed directly in real space, while the long-range component is computed in Fourier space.

ALG 5 **Radiative Cooling**; Ionized gas is subject to cooling processes that dissipate the thermal energy through (collisional excitation and ionization, inverse Compton, recombination and free-free emission). In numerical simulations, cooling processes are included as source terms in the internal energy equation. These are set by a so-called cooling function that is determined by the ionization fractions, density and composition of the gas. Due to the many local properties that affect the ionization rate, we are required to simplify the calculation to reduce computational cost. As such the ionization rate is only calculated based on a handful of chemical species and the cooling function uses pre-calculated tables to determine its value. An effort is being made to improve this in D3.1.

A high-level description of the code has already been given in D2.1, which showcases how each part of the code is called during a timestep.

5.2 Kernels targeted for optimization

The three kernels that we will aim to optimize are:

A) Gravity: Remote communication - exploring improvements to all routines involved in remote tree traversal when performing gravity calculation.

B) SPMHD: Remote communication - similar to the above, but for the routines related to the k-nearest neighbors tree traversal.

C) SPMHD: GPU offloading - Offloading the computation part of the SPMHD routines to the GPU.

During our performance analysis (D2.1) it became clear that ChaNGa suffers from communication imbalance for highly clustered datasets involving both gravity and SPMHD. The merger simulations run during our benchmark represent some of the heaviest computation and communication loads for the code. We will prioritize improving the communication imbalance, as it is a major bottleneck for these kinds of simulations. A further explanation of the issue and potential solutions follows in Section 5.2.1.

Offloading work to the GPU has been done efficiently in ChaNGa for both gravity and cosmological boundary algorithms. For gravity, the full local tree walk is performed on the GPU, which includes both building the interaction lists and processing them. As the local tree walk is the largest computational part of the code, this significantly offloads the CPUs during a timestep, allowing them to focus on the remote tree walks, leading to large overlap in remote and local work. Although many efforts have been performed to also offload the remote tree traversal to GPU [19], it has not performed well with any of the methods developed. The SPMHD part is the next big computational part that would be useful to offload to the GPU. This would in turn include the offloading of the feedback routines as this module also applies a k-nearest neighbors tree traversal similar to that of SPMHD.

We also plan to explore possible enhancements in the vectorization aspect of the ChaNGa code.

5.2.1 Remote communication

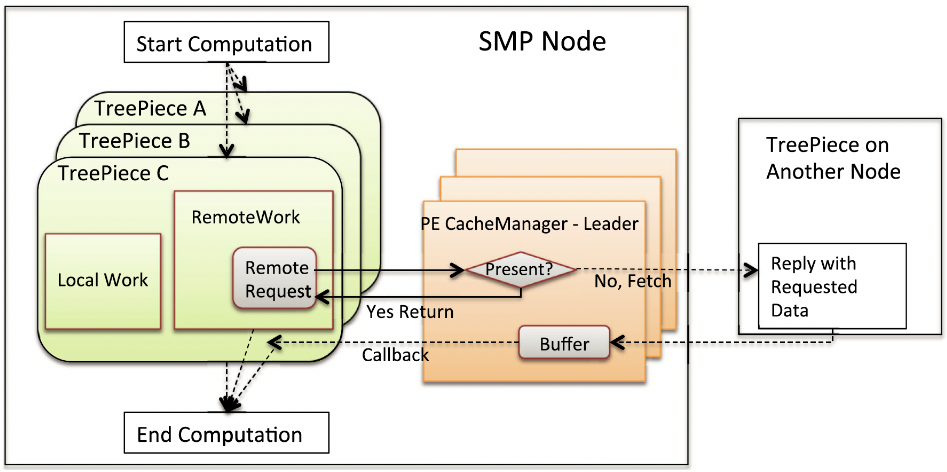


Figure 7: Workflow of the local and remote computation. A local traversal is performed for the tree pieces within the SMP node, while the remaining part of the tree is traversed by requesting communication from remote tree pieces outside the SMP node. Previously requested tree pieces by the SMP node are temporarily saved within the SMP Cache Manager.

During profiling, poor performance scaling was seen in the overall time step at high node counts (see more in D2.1). We attributed this to the communication imbalance produced by the highly clustered dataset. As we have mentioned before in the Introduction, ChaNGa tries to effectively overlap communication and computation at all times. Therefore, the tree traversal is separated into local and remote parts (depicted in Figure 7). During the time the remote tree traversal has sent out requests and is waiting for responses, the local tree traversal can be performed, effectively overlapping communication and computation. The issue arises when dealing with highly clustered simulations, where some tree pieces receive many more requests than others. This substantially slows down the receiving of messages needed for remote traversal. And even though there is overlap between communication and computation, there is insufficient local computation to overlap with the extended delay in receiving messages. This effect is exacerbated as we increase the number of nodes, as this increases the communication imbalance. We found three potential avenues to resolve and improve this issue:

1. **A) Replication of tree pieces-** Another solution is to 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. This proved to be a successful method to reduce the communication imbalance in an older version of ChaNGa [20], but does not exist anymore within the main branch of the ChaNGa code.

2. **B) Topological Routing and Aggregation (TRAM) Charm++ module** - One potential solution to communication issues is to investigate the capabilities of Charm++ TRAM module. This module is designed to automatically aggregate messages and detect collective communication patterns to improve communication within the Charm++ application.

3. **C) Node-wide cache model** - Another solution is to improve the cache manager, to reduce communication volume and improve idle time. Caching the received remote data is an effective way to reduce the volume of communication. ChaNGa already implements a cache as a table of tree-node data hashed by a tree-node key. However, these kinds of hash tables are not ideally suited for shared-memory environments. Since the software cache undergoes frequent updates to accommodate incoming remote requests, parallel accesses by threads on the process must be thread-safe, which proves to be difficult for hash tables of dynamic size. A new shared-memory software cache model for the global tree, was presented by ParaTreeT [21], which is a node-wide cache model with atomic read and write operations.

5.2.2 Isolation of kernels

Due to the asynchronous nature of Charm++ there are no well-defined boundaries to the different regions. Functions belonging to different regions are usually scheduled asynchronously by the runtime, and thus different regions usually blend together. In addition, both the major parts of the code, gravity and SPMHD, relies on the same domain decomposition and tree build to function. Thus extracting and creating a light mini-app becomes challenging and not very useful. However, we do have the ability to disable parts of the full code to isolate certain physics kernels. Such as running only with SPMHD or only with gravity and removing all the additional subgrid physics. The different scientific cases of D1.1 were chosen, such that it allows us to test the performance of the kernels both isolated and in combination with other physics modules. The kernel isolation from the global code is done by the use of preprocessing instructions, which deactivate the parts of the code not used. In total, there will be five versions used for the scientific cases:

Version 1) Gravity only

Version 2) SPMHD only

Version 3) Gravity + SPMHD

Version 4) Gravity + SPMHD + Feedback + Star formation + Cooling

Version 5) Gravity + SPMHD + Feedback + Star formation + Cooling + Cosmological Boundary

The function of each of these modules is listed in 5.1.

6 FIL

FIL is a module of the Einstein Toolkit (ET) that solves the General Relativistic Magneto-Hydrodynamic (GRMHD) equations in 3-dimensions. FIL simulates magnetic fields in dynamical spacetimes, allowing for the study of binary neutron star and black hole-neutron star mergers. It is written in the C++ programming language and is parallelized with MPI+OpenMP. Further information on both FIL and the ET can be found here [22] and here [23], respectively.

6.1 High-level description of the code

The ET is built around the Cactus framework a general framework for the development of portable modular applications, where programs are split into modules known as Thorns. As mentioned above FIL is a Thorn of the ET. The Cactus framework acts as the interface between different Thorns. Some important Thorns are; Carpet, responsible for the adaptive mesh refinement (AMR) grid simulations take place on; Antelope, the space-time evolution Thorn; method of lines (MoL), provides the numerical time integrators; TOVSolver, integrates the TOV equations to provide the initial data for the neutron stars; and the Einstein base Thorns, AMDBase, Hydrobase and TmunuBase define the grid functions[1] for the basic space time variables. Respectively the base Thorns define the 3 metric, the hydrodynamic variables and the stress energy tensor. Having these variables and more defined by the Einstein base Thorns is key to the Cactus framework. It allows for a clearly defined interface between different Thorns. Analysis, initial data and evolution Thorns will all be working on the same well defined grid functions. Antelope, the space-time evolution Thorn, can be used to solve the BSSN, CCZ4 and Z4c formulations of the Einstein's equations. More can be found on the different formulations here [24]. It is important to understand that FIL evaluates the MHD variables and Antelope evaluates the equivalent space-time variables, both sets of variables are passed to the method of lines Thorn which evolves both sets forward in time. Each Thorn has the following directory structure:

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

The COPYRIGHT, README, doc/, test/ and par/ are optional; doc/ will typically be used to store documentation related to the Thorn, and par/ can have example parameter files for the Thorn. The src/ directory will store the code of the Thorn as well as the make.code.defn file which tells the ET which files to compile. FIL's source code can be found in the Relastro_dev/IllinoisGRMHD/ directory. The .ccl files are the files which directly interact with the cactus framework. The configuration.ccl file contains inter-Thorn build dependencies and is optional; interface.ccl defines Thorn-wide variables, grid functions and shared functions; param.ccl defines all parameters and sets their default values and schedule.ccl takes care of all the function scheduling and controls the global storage of all grid functions.

For a full description of the algorithm FIL uses, see section 7.3 in D2.1, what follows is a brief description of the key elements. To evolve the MHD variables the system of ideal MHD equations is written in flux conservation form:

$$\partial_t \mathbf{U} + \partial_i \mathbf{F}^i = \mathbf{S} \tag{12}$$

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

Where **U** is the vector of conserved variables, **F** is the physical fluxes into a volume element and **S** are the source terms within a given volume element. The conserved variables have no physical meaning they are constructed from the primitive variables which correspond to physical quantities such as density, pressure, magnetic field strength etc. FIL evaluates the conservative variables, not the primitives, at each substep of the 4th order Runge Kutta routine (RK4) provided by the MoL thorn. To perform the substeps the source and the flux terms in equation 12 need to be evaluated. This is done by the `Illinois_GRMHD_driver_evaluate_MHD_RHS_type` function. During a time step the primitive variables need to be recovered from the conservative variables multiple times. The conservatives are related to the primitive variables by a non linear set of equations. To recover the primitive variable's a Newton–Raphson-based root finder is needed to recover the primitive variables. The algorithm for recovering the primitive varibales is found in the `illinoisGRMHD_conserv_to_prims` function. At the start of each run of FIL the Cactus schedule is outputted: a complete list of every function, the order it is called in and some of the program structure such as loops. Bellow is an edited version of the scheduler output where effort has been made to reduce the size considerably and slight modifications have been made for the ease of reading.

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

##################################################
#Calculate initial data
##################################################
Margherita_EOS::Margherita_setup_polytrope              #Setup piecewise polytropic EOS.
ADMBase_InitialData                                     #ADMBase initial data
Hydrobase::HydroBase_Initial                            #HydroBase initial data
TOVSolver::TOV_C_Integrate_RHS                          #Integrate the 1d equations for the
    TOV star
TOVSolver::TOV_C_Exact                                  #Set up the 3d quantities for the TOV
    starr
Seed_Magnetic_Fields_BNS::Seed_Magnetic_Fields_Privt    #Set up binary neutron star seed
    magnetic fields.
TmunuBase::SetTmunu                                     #Calculate the stress-energy tensor
Antelope::Antelope_MoLRegister                          #Register the evolution variables with
     MoL

Boundary::BoundaryConditions                            #Set up boundary conditions

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


BNSTrackerGen::BNSTrackerGen_Track_Stars                #Track the stars position

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

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

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

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

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

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

6.2 Kernels targeted for optimization

6.2.1 Kernel 1 - Carpet

The ET is the evolution framework that FIL uses, it controls the AMR grid the simulations are done on as well as provide other routines for the running of the simulations. The precursor to FIL, IllionisGRMHD, was originally evolution framework agnostic. However over subsequent iterations IllinoisGRMHD became more and more dependent on the ET, by the time FIL was developed the ET was the only evolution framework that FIL could use. The profiling showed that the ET was the main cause of slowdown, poor scaling and load balancing issues which had a knock on effects on the other kernels. One key question is whether FIL will continue to use the ET as its evolution framework. When it comes to accelerating the performance of FIL one of the key areas of interest is the AMR grid controlled by Carpet. The developers of the ET are looking to release CarpetX late 2024 which will be GPU accelerated, however initial results do not show the acceleration one would expect. There are other AMR grids such as AMReX that have already been released and are GPU compatible that are of interest, however porting FIL to a different framework will be a significant task. More investigation into the particular areas of poor performance from the ET is hence required.

6.2.2 Kernel 2 - Driver evaluate MHD RHS

The `Illinois_GRMHD_driver_evaluate_MHD_RHS_type` function is called at each sub step of the RK4 method. The driver is responsible for evaluating the flux (MHD) and the source (RHS) in equation 12. To this end, the driver performs successive calls to several functions. The most time consuming functions are:

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

The `Illinois_GRMHD_driver_evaluate_MHD_RHS_type` function has been chosen for optimisation as it takes the longest amount of computational time of all the FIL routines and as it is a purely computational kernel it is a good candidate for SIMD vectorization. GPU acceleration will also be explored.

6.2.3 Kernel 3 - Conservative to primitive solver

As mentioned in section 6.1 the calculation of primitive variables from the conservative is not a trivial task and requires the use of a 2D Newton-Raphson root-finding algorithm, implemented in the `IllinoisGRMHD_conserv_to_prims` function. Because of truncation and interpolation errors, calling the 2D Newton-Raphson solver naively may incur unphysical results. To prevent this, the `IllinoisGRMHD_conserv_to_prims` function performs a series of checks to make sure the conservative variables are in a valid range before the root finder is called. If the conservative values are outside of this range they are minimally modified. Sometimes the root finder will still fail to find a root, this is very rare and almost always occurs in a low-density environment, such as the atmosphere of a neutron star or the interior of a black hole. In such cases the pressure is manually set to guarantee inversion. Once the primitives are successfully calculated, they are checked for physicality (make sure fluid speed remain sub-liminal for example) and the evolution algorithm can continue. Again as this kernel is purely a computational kernel, it is a good candidate for GPU acceleration as well as SIMD vectorization. The conservative to primitive solver was found to be a fairly small section of the overall compute time during the profiling however it is felt that this kernel will become a more significant part of the compute time in regions with high magnetic fields which were not present in the profiling.

6.3 Mini-apps description

Due to the structure of FIL and its coupling to the ET it is not possible to extract the kernels of interest because all kernels depend in multiple ways on the others. Therefore, the mini app will be designed to monitor the kernels of interest via the `ScheduleTraverse` function which runs the main time loop and contains the AMR grid, the evaluate MHD and RHS function, and the conservative to primitive solver.

7 iPic3D

7.1 High-level description of the code

iPic3D is a semi implicit Particle in Cell code developed in C++ to study various phenomena that are observed in collisionless space plasma.

The core of the code is encapsulated within the iPic3D::c_Solver class, here instantiated as KCode. The simulation workflow begins with initialization (Init method), where the solver is configured based on the provided command-line arguments (argc, argv). This initialization includes setting up the computational domain, initializing physical parameters, and preparing the simulation environment.

```cpp
#include <iomanip>
#include "iPic3D.h"

using namespace iPic3D;

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

  iPic3D::c_Solver KCode;
  bool b_err = false;

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

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

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

  for (int i = KCode.FirstCycle(); i <= KCode.LastCycle(); i++) {

    if (KCode.get_myrank() == 0) cout << " ======= Cycle " << i << " ======= " << endl;

    /* ------------------------------------------------------ */
    /* 2- Calculate fields and move particles                 */
    /*    Exit if there is a memory issue with the particles */
    /* ------------------------------------------------------ */

    KCode.UpdateCycleInfo(i);
    KCode.CalculateField();

    b_err = KCode.ParticlesMover();

    if (!b_err) KCode.CalculateBField();
    if (!b_err) KCode.GatherMoments();
    if ( b_err) i = KCode.LastCycle() + 1;

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

    KCode.WriteOutput(i);
    KCode.WriteConserved(i);
    KCode.WriteRestart(i);

  }

  KCode.Finalize();

  return 0;
}
```

Listing 1: High level structure of iPic3D

After initialization, the code injects boundary particles (InjectBoundaryParticles) and gathers initial moments (GatherMoments). The initialization procedures first set up the foundational simulation infrastructure, including allocating memory for the spatial grid and particle arrays, initializing the timestep counters, parsing input parameters to set the domain decomposition and resolution, and generating the initial conditions for the electromagnetic fields and plasma particles based on analytic functions or Restart files. Critical kernels also inject new plasma particles at the boundary surfaces to maintain appropriate particle influx and execute high-performance interpolation operations to accumulate charge density, current density, and pressure tensor attributes from the discretized particles onto the defined grid points for the field solve.

The main loop of the simulation iterates over a predefined number of cycles (time steps), as determined by the FirstCycle and LastCycle methods of the KCode object. The main simulation loop progresses over a series of discrete timesteps. Each cycle first invokes the field solver module, which numerically integrates Maxwell's equations using a finite-difference time-domain method to advance the electric and magnetic field values defined on the grid based on the charge density, current density, and constitutive electromagnetic relations. The subsequent particle pusher module computes the Lorentz force for each simulation macroparticle based on interpolation of the updated grid fields and integrates the equation of motion to determine their new positions and velocities. Additional sub-steps recalculate current density from the particle momenta, re-solve magnetic differential equations, and collect charge density and current density by finite-size particle-in-cell weighting operations.

Each iteration of the main loop involves several key steps: updating cycle information (UpdateCycleInfo), calculating the electromagnetic field (CalculateField), moving the particles based on the newly calculated fields (ParticlesMover), and recalculating the magnetic field (CalculateBField). The ParticlesMover function is particularly critical as it updates the positions and velocities of the particles based on the fields and can return an error flag (b_err) if there is a memory issue. If such an error occurs, the loop is prematurely terminated.

Interspersed I/O routines output detailed electromagnetic field and particle states in a consistent format, along with diagnostic metrics tracking energy conservation. Restart dumps provide contingency against disruption while post-processing tools generate visualizations. This overall workflow produces a self-consistent kinetic approximation of plasma microphysics by the interaction of discrete particles with continually updated electromagnetic fields.

7.2 Kernels targeted for optimization

7.2.1 Kernel 1 - Gather Moments

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

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

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

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

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

}
```

This C++ function GatherMoments() is part of iPic3D specifically developed for calculating and updating physical quantities (moments) based on particle data. The function begins by updating field information with EMf-¿updateInfoFields(grid, vct, col), which likely involves preparing electromagnetic fields and other relevant physical properties for the current simulation state. This is followed by resetting the densities to zero with EMf-¿setZeroDensities(), preparing for fresh calculations in the current cycle.

The core computational task in this function is the loop over species (for (int i = 0; i ¡ ns; i++)), where part[i].interpP2G(EMf, grid, vct) is called. This method represents the interpolation of particle properties (such as charge and current densities) into the grid nodes, a crucial step in PIC simulations. This interpolation is computationally intensive as it involves processing potentially large numbers of particles and mapping their properties onto a spatial grid, a task that requires significant computation, especially for high-resolution grids or large particle counts. This is evident from the profiling done in D2.1.

After interpolation, EMf-sumOverSpecies(vct) aggregates these interpolated values across all particle species, which is essential for calculating net densities and currents on the grid. Additionally, there is a conditional block for handling a specific case (Dipole), which adds a constant charge to a planet model, suggesting a specialized scenario within the simulation framework.

Given the nature of these operations, especially the particle-to-grid interpolation and the summing over species, this function is likely a significant contributor to computational time in the overall simulation, as evident from profiling. These operations are both data- and computation-intensive, involving multiple passes over potentially large arrays representing particles and grid nodes. Thus, we chose this as a kernel.

Regarding GPU offloading, this function is indeed a good candidate. The particle-to-grid interpolation and density summation are parallelizable tasks since they involve operations over independent particles or grid nodes. GPUs, with their high parallel processing capabilities, are well-suited for such tasks. Offloading these computations to a GPU could significantly accelerate the simulation by leveraging the GPU's ability to handle many operations simultaneously, especially beneficial when dealing with large-scale simulations characteristic of plasma physics.

7.2.2 Kernel 2 - Particle Mover

```
  bool c_Solver::ParticlesMover() {

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

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

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

The provided module ParticlesMover() is a critical component of iPic3D code primarily responsible for updating the position and velocity of particles based on the electromagnetic fields. The computational intensity of this function is significant as it involves complex operations for each particle in the simulation. The motion update typically requires evaluating the electromagnetic forces acting on the particles and then updating their states accordingly. Given that simulations can involve a large number of particles, this process becomes computationally demanding. This is clearly evident from the profiling of the code.

Regarding its suitability for GPU offloading, ParticlesMover is indeed an excellent candidate. The nature of particle updates, where each particle's movement can be computed independently of others, lends itself well to parallelization. GPUs, with their ability to handle thousands of simultaneous threads, can significantly accelerate this process. By offloading the particle movement computations to a GPU, each thread can handle the update for a single particle or a small group of particles, leading to a substantial reduction in computation time.

7.2.3 Kernel 3 - Calculate Field

```
  void c_Solver::CalculateField() {
```

```

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

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

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

}
```

The module CalculateField() function in the iPic3D code is designed to calculate the electric field. The presence of MPI_Barrier(MPI_COMM_WORLD) shows the synchronization among different processes in a parallel computing environment, ensuring that all processes reach this point before proceeding, which is essential for consistent and accurate field calculations across the computational domain. The profiling of the code shows that this function is also a significant contributor to the total computation time in the simulation. Field calculation, especially in three-dimensional simulations with fine grids or complex geometries, requires substantial computational resources.

With respect to GPU offloading, this function is a promising candidate. The computation of electromagnetic fields, particularly the solution of Maxwell's equations over a grid, can benefit greatly from parallel processing.

7.3 Mini-apps description

Given the intricacies of particle-in-cell (PIC) simulations, it is indeed challenging to compartmentalize the simulation process into separate mini-applications, especially considering the inter-dependencies and sequential nature of the core computational routines / kernels- ParticlesMover, GatherMoments, and CalculateField. In **GatherMoments**, the properties of the particles are interpolated onto the grid, a process essential for calculating aggregate physical quantities like charge and current densities. These densities are then used in **CalculateField** to update the electromagnetic fields via Maxwell's equations. Subsequently, **ParticlesMover** uses these updated fields to advance the particles in time and space. This sequence of operations is not just linear but also cyclic, as the outcome of the particle movement again influences the next cycle's moment gathering and field calculations.

Creating separate mini-applications for each of these kernels would disrupt this critical sequence, as each mini-app would operate in isolation, lacking access to the dynamically updated data from the other parts of the simulation. This separation would lead to inconsistencies and inaccuracies in the simulation results. Therefore, a more feasible approach is to encapsulate these kernels within a single mini-application. This unified approach ensures that the data flow and dependencies among the kernels are maintained, preserving the integrity and accuracy of the simulation. The mini-app can still focus on optimizing and studying these kernels, but within a cohesive framework that reflects the interconnected nature of PIC simulations.

8 Ramses

RAMSES [25, 26] is an Adaptive-Mesh-Refinement (AMR) code which is used to study astrophysical fluid dynamics and the formation of structures in the Universe. It is based on an oct-tree structure, where parent cells are refined into children cells on a cell-by-cell basis following some user-defined criteria. RAMSES can deal with 1D, 2D and 3D Cartesian grids.

For hydrodynamics, RAMSES integrates the equations of fluid dynamics in their conservative form. This system can be written in the canonical form

$$\frac{\partial \mathbb{U}}{\partial t} + \nabla \cdot \mathbb{F}(\mathbb{U}) = \mathbb{S}(\mathbb{U}). \tag{13}$$

Here the vector $\mathbb{U} = (\rho, \rho\mathbf{u}, E)$ contains the conservative variables: density ρ, velocity $\mathbf{u}$, and total energy $E = e + 1/2\rho u^2$ with e the internal energy. The flux vector $\mathbb{F}(\mathbb{U}) = (\rho\mathbf{u}, \rho\mathbf{u} \otimes \mathbf{u} + P\mathbb{I}, \mathbf{u}\,(\mathrm{E} + P))$ is a linear function of $\mathbb{U}$, and uses the primitive variables ρ, $\mathbf{u}$ and pressure P. $\mathbb{S}(\mathbb{U})$ represents the source terms, e.g. the gravitational force contribution. This system is closed using a perfect gas equation of state $e = P/(\gamma - 1)$ with γ the adiabatic index.

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

$$\frac{\mathbb{U}_i^{n+1} - \mathbb{U}_i^n}{\Delta t} \times V_i = \mathbb{F}_{i+1/2}^{n+1/2} S_{i+1/2} - \mathbb{F}_{i-1/2}^{n+1/2} S_{i-1/2}, \tag{14}$$

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

RAMSES can also handle the evolution of particles, such as stars, Dark Matter (DM) and sink particles, whose evolution is integrated using a Cloud-in-Cell (CIC) interpolation. The same CIC interpolation is used to deposit the mass of the particles onto the grid to solve for the gravitational potential. For gravity, RAMSES can either use a Conjugate Gradient algorithm or a multigrid solver [27]. For each solver, the gravitational potential is solved level-by-level. The time integration can be also accelerated using the adaptive-time-step implementation, in which each AMR level ℓ evolves with its own time-step which satisfies a global synchronization point at the end of the coarser time-step $\Delta t_{\ell_{\min}} = 2^{\ell_{\max}} \Delta t_{\ell_{\max}}$.

RAMSES uses adaptive time integration, where each AMR level is evolved with its own Courant-based timestep, over which the next finer level is recursively subcycled twice and all levels are synchronized after the coarsest timestep.

8.1 High-level description of the code

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

The heart of the RAMSES code is the recursive routine `amr_step`, which handles the AMR hierarchy from the finer to the coarser levels. Inside `amr_step`, the state of the vector $\mathbb{U}$ at time n is copied from the global vector `uold` into the temporary global vector `unew`. Then, the values of the vector $\mathbb{U}$ are updated in every cell to time $n+1$ by the second-order Godunov solver in the subroutine `godunov_fine` using the pre-state `uold` to update the post-state `unew`. After hydrodynamic update, MPI domains need to share their updated values to their neighboring MPI domains. This is done in the `make_virtual_reverse` and `make_virtual_fine` routines, which employ asynchronous point to point MPI communications. After this, all MPI processes are synchronized, `unew` is copied back into `uold` and the next iteration can start with the updated `uold` vector.

```fortran

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

!---------------------------------------------------
recursive subroutine amr_step(istep,ilevel,icount)

 ! Build communication patterns and refine cells flagged for refinement

 if(poisson)then
    ! Compute the density field at level ilevel using
    ! the CIC scheme
    call rho_fine(ilevel) ! Poisson source term
 endif

 !---------------
 ! Gravity update
 !---------------
 if(poisson)then
    ! Compute gravitational potential
    if(multigrid) then
        call multigrid_fine(ilevel)
    else
        call phi_fine_cg(ilevel)
    end if

    ! Compute gravitational acceleration on the grid cells
    call force_fine(ilevel,icount)

    ! Synchronize particles for gravity
    if(pic)then
        call synchro_fine(ilevel)
    end if
 end if

 call set_unew(ilevel)

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

    call compute_godunov(ilevel)! Hydro solver

    call make_virstual_reverse(ilevel) ! Update direct MPI neighbors

    call set_uold(ilevel)

    if(pic)then
        call move_fine(ilevel) ! Move particles
    end if


    call make_virtual_fine(ilevel) ! Send new states to MPI neighbors ghost regions


 if ...
    stop ! exit application
 end if

end subroutine amr_step
```

Listing 2: High level structure of RAMSES

8.2 Kernels targeted for optimization

The first set of kernels we have selected is based on the performance measurements we obtained in D2.1. They correspond to the main time-consuming parts of the code for this test with hydro and MPI communications. Then we identified the kernels for the particle-grid interaction (CIC interpolation). These kernels share the same structure as many other kernels in RAMSES. Once a good strategy of optimization is found for the selected kernels, it will be propagated to the other similar ones. This list can be updated throughout the duration of the project once other use cases are properly benchmarked.

8.2.1 Kernel 1 - Hydrodynamical solver

Kernel 1 corresponds to the predictor-corrector Godunov solver (equation 14). This is a computational step, without I/O or MPI communication. It corresponds to the core of the hydrodynamical kernel, which can be extended to the MHD case, as well as radiative transfer. RAMSES uses an unsplit scheme, so that all directions are treated in a single call to the Godunov solver. Each cell of the computational domain is scanned using loops with a low level of internal vectorization. Octs, i.e., groups of $2^{N_{\rm dim}}$ cells, with $N_{\rm dim}$ the number of dimensions, are first gathered into groups of size `nvector` and then sent to the main routine of the Godunov kernel. The size `nvector` is an input parameter, its optimal value depending on each architecture. Finding optimal `nvector` values is a first basic optimization that we can perform.

```
subroutine godunov_fine(ilevel)
    ! Loop over active grids by vector sweeps
    ncache=active(ilevel)%ngrid
    do igrid=1,ncache,nvector
        do i=1,ngrid
            ind_grid(i)=active(ilevel)%igrid(igrid+i-1)
        end do
     call godfine1(ind_grid,ngrid,ilevel) ! hydro update on nvector groups of $3^{Ndim}$ octs
  end do
end subroutine godunov_fine(ilevel)
```

Listing 3: Basic vectorisation loop in the Kernel 1

From D2.1 results on the first benchmarks, we observed that this region is well optimized on CPU architectures. The challenge now is to port this code to advanced hardware platforms, focusing on different types of GPU accelerators or ARM processors with potentially longer vector lanes (SVE). Note that this strategy for vectorization (`do igrid=1,ncache,nvector`) is employed in more than 90 loops in various parts of the RAMSES code where AMR cells are scanned, for example, to compute the gravitational potential, the conversion of gas into stars or stellar feedback. Therefore, any improvement might result in an important optimization of a significant portion of the code.

8.2.2 Kernel 2 & 3 - MPI communications for the synchronization of the variables

Kernels 2 & 3 correspond to MPI communications between MPI domains. We gather them since they have the same global structure and communication patterns. Kernel 2, `make_virtual_fine`, represents a direct communication of the updated states, like hydrodynamical quantities, from one process to its neighbors. Once the global update is completed (hydro, gravity, feedback, etc...) and all values in the computational domain are synchronized in time, the new hydrodynamical states have to be communicated between the domains ghost regions in order to proceed to the next iteration. Only ghost regions are affected by this communication.

Kernel 3, `make_virtual_reverse`, is the most tricky and constraining communication between MPI processes in RAMSES. For instance, once all the hydrodynamical fluxes have been computed in kernel 1, the hydrodynamical fluxes between neighboring MPI domains have to be communicated to update the hydrodynamical states. This is done to avoid computing a flux twice at level cell interfaces (only the flux on the right face of a cell is calculated). Kernel 3 consists of updating an MPI process from its direct neighbors. It has to be optimized carefully, in particular to guarantee total energy conservation and when adaptive time steps are used since cells at a coarser level should not evolve in the sequence of two consecutive time steps at the finer level.

These two kernels are currently the principal bottleneck regarding performance scaling when the number of MPI processes is too large: the ratio of the surface of the boundary region versus the size of the MPI domain increases, and too much time is spent on MPI communication. The regions contain asynchronous (Isend and Irecieve) communication operations and the `MPI_Waitall` operation which is also sensitive to the load balancing between MPI domains. The following is a summary of the high-level code structure of these kernels.

```fortran
subroutine make_virtual_fine(xx,ilevel)
  ! ------------------------------------------------------------------
  ! This routine communicates virtual boundaries among all cpu's.
  ! at level ilevel for any double precision array in the AMR grid.
  ! ------------------------------------------------------------------

  ! Receive all messages
  countrecv=0
  do icpu=1,ncpu
    ncache=reception(icpu,ilevel)%ngrid
    countrecv=countrecv+1
    call MPI_IRECV(reception(icpu,ilevel)%u,ncache*twotondim, &
    & MPI_DOUBLE_PRECISION,icpu-1,tag,MPI_COMM_WORLD,reqrecv(countrecv),info)
  end do

  ! Gather emission array
  do icpu=1,ncpu
    emission(icpu,ilevel)%u=xx(emission(icpu,ilevel)
  end do

  ! Send all messages
  countsend=0
  do icpu=1,ncpu
    call MPI_ISEND(emission(icpu,ilevel)%u,ncache*twotondim, &
            & MPI_DOUBLE_PRECISION,icpu-1,tag,MPI_COMM_WORLD,reqsend(countsend),info)
  end do

  ! Wait for full completion of receives
  call MPI_WAITALL(countrecv,reqrecv,statuses,info)

  ! Scatter reception array
  do icpu=1,ncpu
    xx(reception(icpu,ilevel)=reception(icpu,ilevel)%u
  end do

  ! Wait for full completion of sends
  call MPI_WAITALL(countsend,reqsend,statuses,info)

end subroutine make_virtual_fine
```

Listing 4: MPI communications to update ghost regions of neighboring MPI domains

These kernels have been selected for their potential optimization of MPI communication and load balancing. The first proposed optimization is on the intra-node communications. Second, we will evaluate the MPI3 standard to improve overlapping of communication and computation. Then, a sorting of the octs in MPI domains is envisioned in order to separate internal octs, i.e. the ones that do not share a common boundary with another MPI process, and boundary octs. This last optimization will affect not only kernels 2 and 3 but also the calls to kernel 1, so it will be difficult to evaluate the degree of optimization without running the three kernels in the same use case.

8.2.3 Kernel 4 - Particle-mesh interaction

Kernel 4 corresponds to the mass deposition from particles (stars, dark matter) onto the grid, using the cloud in cell (CIC) method, prior to the computation of the gravitational potential. It is a testbed for the optimization of the particle-mesh interaction kernels (not all use CIC), used, for instance, to move particles, to convert gas into stars or to apply star formation feedback. Kernel 4 is called within the `rho_gine(ilevel)` routine which also integrates calls to kernels 2 and 3.

```fortran

subroutine rho_fine(ilevel)

  !------------------------------------------------------------------
  ! This routine computes the density field at level ilevel using
  ! the CIC scheme.
  !-------------------------------------------------------

  ! Compute density due to current level particles
  if(pic)then
```

```
     call rho_from_current_level(ilevel)
  end if
  ! Update boudaries
  call make_virtual_reverse(rho(1),ilevel)
  call make_virtual_fine   (rho(1),ilevel)

end subroutine rho_fine
!################################################################################
!##########################################################
subroutine rho_from_current_level(ilevel)
  !------------------------------------------------------------------
  ! This routine computes the density field at level ilevel using
  ! the CIC scheme from particles
  !------------------------------------------------------------------

  ! Loop over cpus
  do icpu=1,ncpu
    ! Loop over grids
    do jgrid=1,numbl(icpu,ilevel)
        npart1=numbp(igrid)  ! Number of particles in the grid
        do jpart=1,npart1
            ip=ip+1
            if(ip==nvector)then
                !------------------------------------------------------------------
                ! Compute the density field at level ilevel using
                ! the CIC scheme on groups of nvector particles
                !------------------------------------------------------------------
                cic_amr(ind_cell,ind_part,ind_grid_part,x0ig,ip,ilevel)
            end if
        end do
        ! End loop over particles
    end do
    ! End loop over grids
  end do
  ! End loop over cpus

end subroutine rho_from_current_level
```

Listing 5: High level code structure for the CIC interpolation of the particles mass onto the grid

Like kernel 1, this kernel is well optimized on CPU architectures, hence the challenge is the same: porting this code on advanced hardware platforms. Note that the same strategy for vectorization is also employed for the particle loops. This loop is present in more than 20 places in the entire code where the particles are scanned. Therefore, any improvement might result in an important optimization of a significant portion of the code.

8.3 Mini-apps description

RAMSES is a big code consisting of about 90 thousand lines, with a lot of physics included and a high entanglement of fundamental kernels in each of the physical modules. Indeed, many of the developments have been done through copy-paste of the main features and loops of the aforementioned kernels. As such, it is not easy to create light-isolated mini-apps. We will isolate the kernels from the global code using preprocessing instructions and conditional instructions which deactivate the parts of the code that are not used. Then we will extract the core parts of the kernels, which will be identified thanks to a fine grain performance analysis. Once a significant optimization of the kernels is reached, we will progressively implement it in all the parts of the code that share the same basis as the four kernels we selected.

9 Modules interoperability across codes

The interoperability of scientific codes is a highly advisable goal for every community targeting a class of problems in science, and it should be promoted as a standard practice.
Using community-developed kernels across codes or pipelines owned by different groups may offer the advantage of a larger user community and more intensive and extensive tests on a larger variety of conditions, besides reducing the total time spent in development because of the smaller redundancy.

From a broad perspective, it is possible to individuate two types of "modules": those that either do or do not need to interact with the peculiar data structures of a code. The modules that perform specific operations on/from a subset of variables could be more easily engineered as inter-operable and re-usable, for instance, in the form of libraries called from within the codes. A prominent example of that is the linear algebra. In the domain of this project, typical candidates are all calculations made on the value of the variables of a single resolution element (RE, either a particle or a cell). Valid examples are a star-formation algorithm that decides whether the gas in a resolution element will form stars or not or a cooling function that determines the energy loss only based on the values of physical properties available at the moment of calculation.
On the other hand, operations that tightly interact with the backbone of a code, for instance, the domain decomposition, are more difficult to be extracted and offered as inter-operable modules.

Within SPACE, two broad categories of codes are included: fluid- (or mesh-) based codes (PLUTO, BHAC, FIL & RAMSES) and particle-base codes (CHANGA and OpenGadget). By the nature of their fundamental algorithms, these two classes of codes could hardly share any module.

In the following, we detail the effort that we are making in the SPACE collaboration to *(i)* foster the exchange of know-how and knowledge in the development of modules that solve common problems, *(ii)* produce some modules that could be actually shared among codes of the same type.

9.1 Grid codes

Fluid codes are typically constructed using either finite volume or finite difference techniques which - in their basic form - follow a standard workflow made by a sequence of computationally intensive kernels. While these kernels cannot - in the way they are coded - be easily re-used across codes, sharing them still allows code developers to adopt similar optimization strategies and development approaches, in the attempt to improve performance and scalability. To this end, we have identified three such common modules which are briefly described in the following.

- `Reconstruction`: these have already been described above see, e.g., 3.2.2, 4.2.1 and 6.2.2. The reconstruction kernel is typically employed to recover the left and right interface values from the volume averages of the solution itself. As codes may rely on algorithms with different spatial order (e.g. linear, parabolic or cubic interpolation) or approach (e.g. conservative vs. primitive vs. characteristic variables), a general re-usable module among codes would be an overwhelming complex task. However, the optimization strategies (e.g. faster vs slower indices, local array allocation, data transposition, physical constrains such as density or pressure positivity) can be easily identified ad shared among code developers.

- `Riemann Solver`: used to obtain a consistent numerical flux by solving (to various degree of approximation) the Riemann problem between adjacent left and right states. The Riemann solver can be considered the heart of a finite-volume code. While a different solvers are available, the most basic ones (e.g. the Rusanov-Lax-Friedrichs solver) can be shared among different groups.

- `ConsToPrim`: In order to correctly capture discontinuous features, astrophysical fluid / plasma codes evolve conservative quantities such laboratory density, momentum and energy density. However, the recovery of pressure and velocity from these quantities is a crucial step in the algorithm. While this is a relatively simple procedure in the case of classical (Newtonian) flows, the relativistic counterparts require special care and an iterative algorithm is typically employed. While this kernel should allow for some flexibility (mainly in the way unphysical values are handled during runtime), the general designing and optimization strategies on GPU can be shared by codes having relativistic solvers (e.g. PLUTO, FIL & BHAC).

9.2 Particle codes

Both particle-based codes within the SPACE project, CHANGA and OpenGadget, are based on tree algorithms and follow a logically similar workflow: update the domain decomposition if needed, calculate the gravitational force, calculate the hydrodynamic force, compute physical processes (cooling, star formation, etc.), and update positions and velocities. This workflow summary is quite simplified; for instance, the integration scheme is a kick-drift-kick method, such that the updates of positions and velocities are performed at different times; the time-stepping is hierarchical and as such not all the particles are active every time; the gravitational force is split between long-range and short-range interactions. For more information we re-direct the reader to the detailed description of each code.
Due to the similarity between the two codes, they are good candidates to share some of the modules that are being optimized in the project, as follows:

- `Cooling Function` Matter in a gaseous or plasma state, under typical conditions in cosmological simulations of large-scale structure evolution or galaxy formation, may lose or gain energy through radiative cooling and heating. This includes various physical processes such as excitations, de-excitations, collisions, etc. The net energy balance is determined by physical properties such as density, temperature, chemical composition and the radiation background. The effectiveness of physical processes and which ones are active depends as well on the same properties (eminently on the temperature). The modelling of these physical processes can require very sophisticated codes, and the associated calculations can be extremely demanding from the computational point of view.
  Within SPACE, we are developing new sophisticated modules and exploring the feasibility of using online ML techniques.
  This will result in a module adopted by both codes as a common result of a joint development effort.

- `Stellar Evolution` Stars are a fundamental tracker of a galaxy's evolution. The formation of stars depends heavily on local physical conditions, which are, in turn, shaped by a wide range of large-scale physical processes. Subsequently, their feedback, in terms of energy and chemical elements ejected by supernovae explosions and stellar evolution, strongly impacts the physical evolution of the surrounding environment. The star formation algorithm is crucial for every code simulating galaxy evolution. However, once the "stars" are released in the simulation, the evolution of the stellar populations that they represent depends on a set of physical quantities (the age, the chemical composition, the mass) and assumptions (the supernovae model, the initial mass function, the lifetime function, etc.). As such, it may be rendered a "Module" shared by different codes. Even those codes do not need to be Lagrangian. We built such a fully thread-safe module, and we will release it after complete testing.

- `Tree` The tree-related data structures and routines are the core of every Lagrangian code in this domain. As detailed in the code descriptions in D1.1 and in this deliverable, the tree routines offer a multitude of "services" related to the spatial ordering of the particles: to name a few, domain decomposition is built upon the insight given by the tree nodes and different types of neighbourhood of a given target particle, used in several modules, are retrieved using the tree.
  In principle, the data needed to build the tree and run the relevant routines are just a few: the position and the mass. However, since different "types" of particles typically exist in a code (gas, dark matter, stars, black holes, etc.), this information may also be needed because the neighbours needed for a given operation may be of a precise type. In addition, several other code-specific information are typically added because that allows for several optimizations that are strictly related to the code's architecture. For these reasons, producing a "universal" tree code that a code could import as a library is not an obvious and well-defined task, especially considering that CHANGA relies on a PGAS approach, while OpenGadget does not. However, the SPACE collaboration fosters a close exchange of knowledge, know-how and best practices between the two Lagrangian codes.

10 Conclusions

This deliverable has detailed the process of selecting fundamental modules and kernels within these codes, highlighting their extraction as mini–applications for the purpose of optimization and co-design. The preferred methodology, aimed at disentangling infrastructure functionalities and physics from the code, offers a systematic approach to enhancing performance, scalability, and energy efficiency.

While the ideal procedure involves isolating and creating standalone applications for each module, the intricate nature of certain codes necessitates an alternative approach, that we detailed in the codes sections of the document. The strategic removal of unnecessary functionalities within the existing code structure emerges as a pragmatic solution, ensuring continued access to primary modules while potentially simplifying the integration of updated functionalities. This approach, although not as pristine as creating individual mini-applications, retains the essential functionality within the code and streamlines access to required modules, presenting a feasible avenue for advancement.

Moreover, this approach minimizes the arduous back-integration process, as the updated modules are already embedded within the overarching code structure.

The deliverable describes the different codes in the SPACE CoE and their modules and kernels in greater detail.

For this task, each code presented a high-level description of the implemented algorithms and, from a comparison with the other codes, a large variety of different approaches and architectures can be seen. The only similarity between the codes is that they consist of a main loop iterating through the different steps until the final step is reached, but the computations done within each step differ vastly between the different codes. A consequence of these differences between the codes leads to the large variety of kernels and modules which have been chosen for optimization by each code. In the following table all modules and kernels are listed:

Code	**Modules and kernels**
OpenGADGET	Tree Building, Tree Walking, Domain Decomposition, Gravity Tree, Density Loop
PLUTO	Boundary Conditions, States reconstruction, Riemann Solver, Right Hand Side, Constrain transport update
BHAC	Primitive reconstruction
Changa	Gravity, SPMHD, Feedback, Star formation, Cooling, Cosmological Boundary
FIL	Carpet, Driver evaluate MHD RHS, Conservative to primitive solver
iPic3D	Particle Mover, Calculate Field
Ramses	Hydrodynamical solver, MPI communications, Particle-mesh interaction

Table 1: Table listing all modules and kernels for each code identified for optmization.

As it is summarized in Table 1, each code focuses on one to six different modules and kernels. One large aspect of the selected regions is the communications between different tasks, which will be of great importance for the exa-scale.

In conclusion, the key deliverable for this report encompasses the comprehensive description of the codes and the identification of the modules and kernels earmarked for optimization. With the achievement of these tasks, this report sets the foundations for the optimization activities to be pursued in WP2. This meticulous groundwork serves as a cornerstone, facilitating the delineation of methodologies for module extraction and establishing a roadmap toward enhanced performance, scalability, and energy efficiency.

References

[1] A. Mignone, G. Bodo, S. Massaglia, T. Matsakos, O. Tesileanu, C. Zanni, and A. Ferrari, "PLUTO: A Numerical Code for Computational Astrophysics," *The Astrophysical Journal Supplement Series*, vol. 170, no. 1, pp. 228–242, May 2007.

[2] A. Mignone, C. Zanni, P. Tzeferacos, B. van Straalen, P. Colella, and G. Bodo, "The PLUTO Code for Adaptive Mesh Computations in Astrophysical Fluid Dynamics," *The Astrophysical Journal Supplement Series*, vol. 198, no. 1, p. 7, Jan. 2012.

[3] A. Harten, P. Lax, and B. Leer, "On upstream differencing and godunov-type schemes for hyperbolic conservation laws," *SIAM Review*, vol. 25, no. 1, pp. 35–61, 1983. [Online]. Available: https://doi.org/10.1137/1025002

[4] T. Miyoshi and K. Kusano, "A multi-state HLL approximate Riemann solver for ideal magnetohydrodynamics," *Journal of Computational Physics*, vol. 208, no. 1, pp. 315–344, Sep 2005.

[5] A. Mignone, M. Ugliano, and G. Bodo, "A five-wave Harten-Lax-van Leer Riemann solver for relativistic magnetohydrodynamics," *Monthly Notices of the Royal Astronomical Society*, vol. 393, pp. 1141–1156, Mar. 2009.

[6] A. Mignone and L. Del Zanna, "Systematic construction of upwind constrained transport schemes for MHD," *Journal of Computational Physics*, vol. 424, p. 109748, Jan. 2021.

[7] 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.

[8] 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.

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

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

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

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

[13] SPACE, "Deliverable D2.1: Performance profiling and benchmarking," https://www.space-coe.eu/files/SPACE_D2_1_Performance_profiling_and_benchmarkingfinal.pdf, December 2023.

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

[15] 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.

[16] 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.

[17] J. W. Wadsley, B. W. Keller, and T. R. Quinn, "Gasoline2: a modern smoothed particle hydrodynamics code," *Monthly Notices of the Royal Astronomical Society*, vol. 471, no. 2, pp. 2357–2369, Oct. 2017.

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

[19] J. Liu, M. Robson, T. Quinn, and M. Kulkarni, "Efficient gpu tree walks for effective distributed n-body simulations," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 24–34. [Online]. Available: https://doi.org/10.1145/3330345.3330348

[20] 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.

[21] J. Hutter, J. Szaday, J. Choi, S. Liu, L. Kale, S. Wallace, and T. Quinn, "Paratreet: A fast, general framework for spatial tree traversal," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 762–772.

[22] 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.

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

[24] L. Baiotti and L. Rezzolla, "Binary neutron star mergers: a review of Einstein's richest laboratory," *Rept. Prog. Phys.*, vol. 80, no. 9, p. 096901, 2017.

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

[26] "Ramses repository on GitLab," https://bitbucket.org/rteyssie/ramses/src/master/.

[27] T. Guillet and R. Teyssier, "A simple multigrid scheme for solving the Poisson equation with arbitrary domain boundaries," *Journal of Computational Physics*, vol. 230, no. 12, pp. 4756–4771, Jun. 2011.