MNRAS 000, 1–30 (2024) Preprint 28 May 2024 Compiled using MNRAS LATEX style file v3.3

The Shamrock code: I- Smoothed Particle Hydrodynamics on GPUs.

T. David--Cléris,[1]★ G. Laibe,[1,2] and Y. Lapeyre[1]
[1]ENS de Lyon, CRAL UMR5574, Universite Claude Bernard Lyon 1, CNRS, Lyon, F-69007, France.
[2]Institut Universitaire de France

Accepted XXX. Received YYY; in original form ZZZ

ABSTRACT

We present Shamrock, a performance portable framework developed in C++17 with the SYCL programming standard, tailored for numerical astrophysics on Exascale architectures. The core of Shamrock is an accelerated parallel tree with negligible construction time, whose efficiency is based on binary algebra. The Smoothed Particle Hydrodynamics algorithm of the Phantom code is implemented in Shamrock. On-the-fly tree construction circumvents the necessity for extensive data communications. In a Sedov blast test performed with 65 billion of particles, Shamrock achieves a single time step in just 7 seconds using the 1024 MI250X GPUs of the Adastra Cluster. This equates to processing 9 billion particles per second, with 64 million particles per MI250X. The parallel efficiency across the entire cluster is $\sim 92\%$. The code is publicly distributed on Github under the open source CeCILL license.

Key words: Methods: numerical

1 INTRODUCTION

The study of the formation of structures in the Universe is a field in which non-linear, non-equilibrium physical processes interact at many different scales, requiring ever greater computing resources to simulate them, right up to Exascale (one quintillion operations per second). To increase energy efficiency with acceptable CO_2 emissions, recent super computers have been designed with specialised hardware such as ARM central processing units (CPUs) or graphics processing units (GPUs). Those involve multiple computational units that perform the same operation on multiple data simultaneously through specific instructions (Single Instruction Multiple Data, or SIMD parallel processing). This type of hardware differs radically from standard x86 CPUs, requiring a complete rewrite of CPU-based codes.

Considerable efforts have recently been invested into developing codes adapted to the new hybrid architectures aimed at Exascale (e.g. Idefix: Lesur et al. 2023 ; Parthenon: Grete et al. 2022 ; Quokka: Wibking & Krumholz 2022). The performance of those codes is conditioned by the rate at which data involved in the solver can be prepared, explaining the efficiency of grid-based Eulerian codes developed to date. For example, the multiphysics Godunov code Idefix uses a fixed grid, so no overhead is required when executing the numerical scheme. On the other hand, simulating moving disordered particles on Exascale architectures is a tremendous challenge, regardless of whether they are tracers for Eulerian methods, super particles for Lagrangian methods or interpolation points for Fast Multiple Moments. The rule of thumb is that performance decreases when the number of neighbours increases and when they are unevenly distributed.

Our code Shamrock is a performance portable framework aiming at hybrid CPU-GPU multi-node Exascale architectures (Sect. 2). The design of Shamrock makes it appealing for with particle-based methods such as Smoothed Particle Hydrodynamics (e.g. Hopkins 2015; Price et al. 2018; Springel et al. 2021), while remaining inherently compatible with any distribution of numerical objects (grids, particles) and numerical schemes (grid-based or Lagrangian). Our strategy in Shamrock is that the tree used for neighbour search is never updated, unlike in existing codes. Instead, we are aiming for a highly efficient fully parallel tree algorithm that allows on-the-fly building and traversal, for any distribution of cells or particles. The specific nature of GPU architectures calls for a different design from the state-of-the-art methods developed for CPUs (e.g. Gafton & Rosswog 2011). The simulation domain undergoes an initial partitioning into sub-domains, fostering communication and interface exchange through an outer layer of MPI parallelism presented in Sect. 3. The core of Shamrock is its inner layer of parallelism, which consists in distributing the operations performed for the hydrodynamical solver on a sub-domain over the GPUs using the SYCL standard. The overall performance of Shamrock hinges on the performance on neighbour finding *on a single GPU*. Hence the need for a tree building and traversal procedure that doesn't bottleneck the hydrodynamical time step. In Sect. 4, we first present a tree algorithm that has the required level of performance for any number of objects compatible with current GPU capabilities. It combines state-of-the-art algorithms on Morton codes (Morton 1966; Lauterbach et al. 2009) with specific optimisations, a key feature being the Karras algorithm (Karras 2012). The resulting tree building time is negligible. We detail the subsequent implementation of a Smoothed Particle Hydrodynamics solver (SPH) in Shamrock in Sect. 5, and present the results obtained on standard astrophysical tests in Sect. 6. Our implementation is almost identical to that of the Phantom code, facilitating performance assessments and comparisons on both one and multiple GPUs (Sect. 7). We discuss potential future directions for Shamrock in Sect. 8.

★ E-mail: timothee.david--cleris@ens-lyon.fr

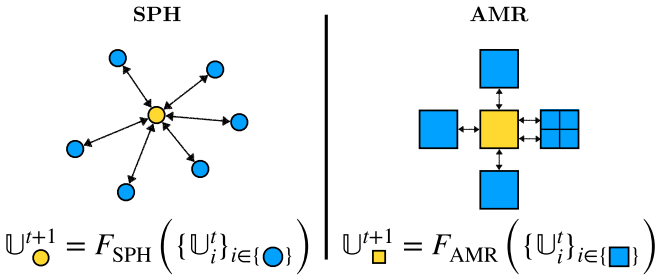

Figure 1. Numerical integration of an hydrodynamic quantity $\mathbb{U}$ involves finding neighbours i (particles, cells), then adding their contributions according to the chosen solver F.

2 THE SHAMROCK FRAMEWORK

2.1 Modular computational fluid dynamics

Computational fluid dynamics consists in discretising a physical system of partial differential equations, alongside specifying initial and boundary conditions. Deterministic numerical schemes can be viewed as being the combination of neighbour finding and specified arithmetic (see Fig. 1), and an algorithm capable of operating on neighbours can provide a generic framework for implementing schemes frequently used in astrophysics, such as Lagrangian Smoothed Particle Hydrodynamics (SPH), Eulerian Adaptive Mesh Refinement (AMR) grid-based methods, or others, within the same structure. This is the very purpose of SHAMROCK: to abstract optimised neighbour search, in a way that is versatile enough that the user only needs to provide functionality to write new schemes with minimal changes. Fig.2 sketches the SHAMROCK framework: a collection of libraries connected by standardised interfaces, where models (CFD solvers or analysis modules) are implemented atop these libraries.

2.2 Multi-GPUs architectures: choice of languages and standards

Modern computer hardware harnesses graphics processing units (GPUs) as computing accelerators. A typical compute node configuration consists of several GPUs connected to a CPU via a PCI express or other proprietary interconnect. Each GPU is equipped with its network card, enabling direct communications from one GPU to another with the Message Passing Interface (MPI) protocol. The GPUs themselves are specialised hardware capable of exploiting the full bandwidth of their high speed memory in tandem with a high compute throughput using SIMT (Single instruction, Multiple Threads) and SIMD (Single Instruction Multiple Data). This design renders GPUs more potent and energy-efficient than CPUs, especially for processing parallelized tasks involving simple, identical operations. Performing simulations on architectures comprising thousands of GPUs introduces several challenges: evenly distributing the workload among the available GPUs (load balancing problem), communicating data between domains to perform the computation while moving the communication directly to the GPU if possible (communication problem), structuring and organising the workload on the GPU into GPU-executed functions called *compute kernels* to make the best use of hardware capabilities (algorithmic problem). The first two points are are common issues associated with MPI, while the third is specific to GPU architecture, raising the question of choosing an appropriate backend.

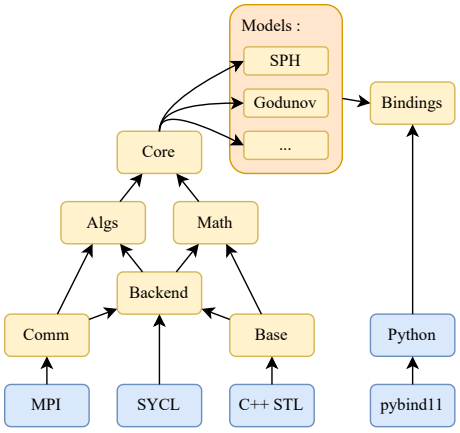

Figure 2. Internal structure of SHAMROCK: functionalities for calculating neighbour finding are organised in different layers of abstraction, enabling the independent treatment of any numerical scheme (Models).

GPU vendors have developed various standards, languages and libraries to handle GPU programming, the most widely used to date for scientific applications being CUDA and ROCM , which are vendor-specific. To address the issue of portability, libraries and standard have been created to enable the same code to be used on any hardware from any vendors. Current options include KOKKOS (Trott et al. 2021), OPENACC and OPENMP (target). The SYCL standard, released by Khronos in 2016, is a domain-specific embedded language compliant with C++17, which is compiled to the native CUDA , ROCM or OPENMP backend. With a single codebase, one can directly target directly any GPUs or CPUs from any vendors, eliminating the need for separate code paths for each supported hardware. To date, the two main SYCL open source compilers are ADAPTIVECPP (Alpay & Heuveline 2020; Alpay et al. 2022), and OPENAPI/DPC++, which is maintained by Intel. Among other heterogeneous parallelisation libraries, we use the SYCL standard to develop SHAMROCK, since it offers robustness, performance (Markomanolis et al. 2022), portability (Deakin & McIntosh-Smith 2020; Jin & Vetter 2022) and potential for durability. SYCL compilers can also generally compile directly to a native language without significant overhead, delivering near-native performance on Nvidia, AMD and Intel platforms (e.g. tests with GROMACS, Alekseenko & Páll 2023; Abraham et al. 2015; Alekseenko et al. 2024). Since C++ code written using SYCL is compiled directly to the underlying backend (CUDA or ROCM or others), we harness direct GPU communication and use vendor libraries directly in the code.

2.3 Elements of software design

Software design of SHAMROCK relies on

- A modular organisation of the code structured around interconnected cmake projects,
- Python bindings provided through the use of pybind (Jakob et al. 2024),
- Version control development for forking and branching (Git),
- A comprehensive, automated test library handling multiple configurations of compilers, targets and versions,
- Automated deployment of code across machines by the mean of environment scripts,
- A user-friendly PYTHON frontend for versatility.

Further details are provided in App. A

3 DOMAIN DECOMPOSITION & MPI

3.1 Simulation box

The three-dimensional volume on which a numerical simulation is performed can be embedded in a cube, whose edges define axes for Cartesian coordinates. This cube is often referred as an Aligned Axis Bounding Box, or *AABB* , particularly within the ray-tracing community. The box is parametrised by two values, $r_{\min}$ and $r_{\max}$, which are chosen to represent the minimum and maximum possible coordinates inside the cube in all three dimensions. For convenience, we shall refer to this *AABB* as the *simulation box* of Shamrock. Within this box, coordinates can be mapped to a grid of integers, by subdividing the simulation box coordinates into $N_{\rm g}$ grid points on each axis, where $N_{\rm g}$ is a power of two. In pratice, we use $N_{\rm g} = 2^{21}$ or $N_{\rm g} = 2^{42}$ (see Sect. 3.5).

3.2 Patch decomposition

The first level of parallelisation in Shamrock consists of dividing the simulation box into elementary volumes, or subdomains, which are then distributed across nodes of a computing cluster. For convenience, we shall further refer to these subdomains as *patches*. In Shamrock, patches are constructed following a procedure of recursive refinement. Starting from the simulation box, patches are divided into eight patches of equal sizes by splitting in two equal parts the original patch on each axis. The resulting structure is an octree, where each node is either a leaf or an internal node with eight children. The patches managed by Shamrock are the leaves of this octree. We call this structure the *patch octree* of Shamrock. The patch octree is similar to the structure of a three-dimensional grid that has been adaptively refined (AMR grid). The cells of this AMR grid would correspond to the patches of Shamrock. Similar to an AMR grid, patches can be dynamically subdivided or merged.

To each patch p, we associate an *estimated load* $\mathcal{W}_p$, which is an estimate of the time required to perform the computational load on the patch. The load depends *a priori* on the type of simulation chosen by the user (e.g. fixed or refined grids, particles, see Sect. 3.5). If the estimated load of a patch exceeds a maximum threshold ($\mathcal{W}_p > \mathcal{W}_{\max}$), the patch is subdivided. If the estimated load is below a minimum threshold ($\mathcal{W}_{\min}$), the patch is flagged for a merge operation. In Shamrock, patch merging is performed when all eight patches corresponding to the same node in the patch octree are flagged. To avoid cycling between subdivisions and merges, we enforce $\mathcal{W}_{\max} > 4\mathcal{W}_{\min}$. Hence, the decomposition of the simulation box into patches is only controlled by the values of $\mathcal{W}_{\min}$ and $\mathcal{W}_{\max}$. Shamrock maps several patches to a given MPI rank in a dynamical manner. We call this decomposition an *abstract domain decomposition*. In practice, we find that 10 patches per MPI rank provides a compromise between the level of granularity required for effective load balancing and the overheads associated with patch management.

3.3 Data Structure

Each patch in Shamrock is associated with two types of information. The first type is the *patch metadata*, which encompasses the current status, location and identifier of the patch. The second, called *patch data*, comprises the data pertaining to the fields processed by the patch.

3.3.1 Patch metadata

Within Shamrock, metadata is synchronised across all MPI ranks. This synchronisation is made possible by the use of a class of small size (80 bytes when compiled). The metadata of a Shamrock patch is represented in the code with the following class

```
template<u32 dim>
struct Patch{
    u64 id_patch;
    u64 pack_node_index;
    u64 load_value;

    std::array<u64,dim> coord_min;
    std::array<u64,dim> coord_max;

    u32 node_owner_id;
};
```

In this class, `u32` denotes 32 bits unsigned integers and `u64` their 64 bits variants, `id_patch` the patch unique identifier, `load_value` the estimated load of a patch (see Sect. 3.2), `coord_min` and `coord_max` represent edges of the AABB patch on the integer grid, `node_owner` the MPI rank owning the current patch. Finally, `pack_node_index` is an additional field used to specify that a patch aims to reside in the same MPI ranks as another one (see section 3.4 for more details). We also provide a dedicated MPI type to facilitate the utilisation of collective operations on patch metadata.

3.3.2 Patch data

The *patch data* of a patch is a list of fields related to a collection of objects (cells or particles). A field can contain one or multiple values per object, as long as the number of values per object is constant. The first field, so-called the *main field* in Shamrock, must have one value per object and store the positions of every object in the patch. Domain decomposition and load balancing are executed based on the positions stored in the main field. When a patch is moved, split or merged, the corresponding operations are applied to the other fields as well. This ensures that communications are implicitly modified when the layout of the data is changed, eliminating the need for direct user intervention. For efficient implementation of new physics, the fields stored in the patch data can encompass a wide range of types (scalar, vector, or matrices, with float, double, or integer data), arranged in any order. This versatility is enabled by representing the patch data as a `std::vector` of `std::variant` encompassing all possible field types. This aspect is abstracted from the user, as only field identifiers and types are required. One example of such use is

```
PatchData & pdat = ...
// get the layout of the patch data
PatchDataLayout &pdl = pdat.pdl;
// get id of the field (name and type specified)
// f64_3 is a 3 dimensional double precision vector
const u32 ivxyz      = pdl.get_field_idx<f64_3>("vxyz");
// get the field at this id
PatchDataField<f64_3> & vxyz =
                          pdat.get_field<f64_3>(ivxyz);
```

3.3.3 Patch scheduler

In Shamrock, a single class is responsible of managing patches, distributing data to MPI ranks and processing the refinement of the patch grid. This class contains the patch octree, patch metadata, and

patch data. It is referred internally as the `PatchScheduler`. This class is only controlled by four parameters: the patch data layout, which specifies the list of fields and the corresponding number of variables, the split criterion $\mathcal{W}_{\max}$ and the merge criterion $\mathcal{W}_{\min}$ that control patch refinement, and the load balancing configuration. The patch scheduler is designed to operate as a black box for the user. The user calls the `scheduler_step` function, which triggers the scheduler to execute merge, split, and load balancing operations. The `scheduler_step` is called at the beginning of every time step in practice. Multiple 'for each' functions are provided in Shamrock as abstractions for iterating over patches. An example of such use is

```
PatchScheduler & scheduler = ...

scheduler.for_each_patchdata(
    // the c++ lambda contain the operation
    // to perform on the patches
    [&](const Patch & p, PatchData &pdat) {
        // do someting on the patch
    }
);
```

These abstractions shield the end user from interactions with the MPI layer. The strategy is as follows: one does not need to be aware of which patches reside on which MPI ranks. Indeed, operations are conducted solely through 'for each' calls to the patches, and the scheduler handles the other tasks.

3.4 Scheduler step

Fig. 3 illustrates a single scheduler step in Shamrock. During this step, patch data are exclusively processed on their current MPI rank, while patch metadata and the patch tree remain unchanged over all MPI ranks.

3.4.1 *Synchronising metadata*

The initial operation conducted during a scheduler step consists in synchronising the metadata across the MPI ranks. This operation, named `vector_allgatherv` in Shamrock, is implemented as an extension of the MPI primitive `MPI_ALLGATHER_V` (see fig.4). Given a `std::vector` in each MPI ranks, `vector_allgatherv` returns on all ranks the same `std::vector` made by concatenating the input vectors in each ranks. We create an MPI type for the patch metadata, and use `vector_allgatherv` to gather all the metadata of all patches on all MPI ranks. This operation returns the list containing the metadata of all patches in the simulation (the step 'Metadata sync' in Fig.3).

3.4.2 *Listing requests*

The operation 'Get requests' depicted in Fig.3 provides the list of identifiers for patches requiring merging or splitting. A patch splits when its estimated load exceeds the split criterion (see Sect.3.2). If all children of a node in the patch tree meet the merge criterion, they merge, resulting in the parent node being marked for pending child merge and consequently transitioning into a tree leaf.

3.4.3 *Patch splitting*

Subsequent split operations on the metadata and the patch tree are carried out in each MPI rank. If the MPI rank holds the patch data

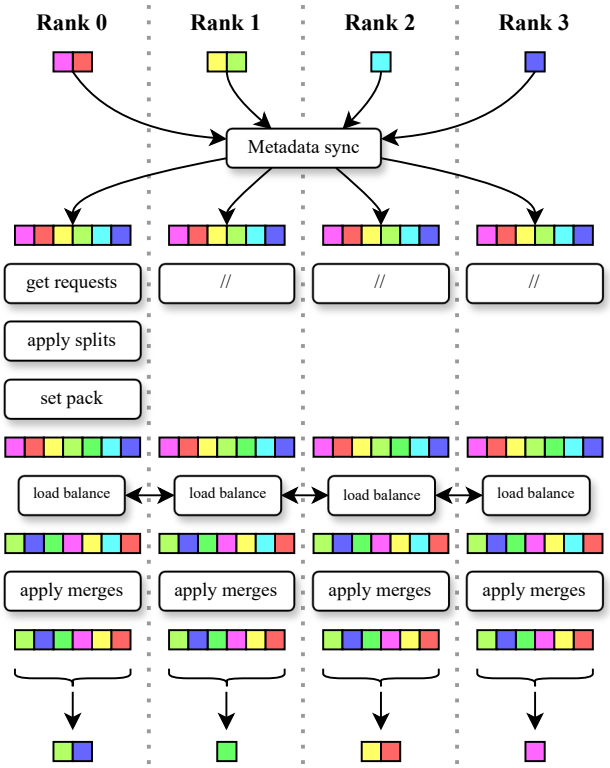

Figure 3. Illustration of a scheduler step. Initially, a synchronisation of the patch metadata occurs across all MPI ranks, resulting in each rank possessing an identical list of all patch metadata. Subsequently, each MPI rank generates a list of split and merge requests. Split requests are then executed, followed by setting the packing index. The subsequent operation consists in performing load balancing on all patches. Finally, merge requests are carried out to complete the step.

associated with the patch being split into eight new patches, the patch data is then subdivided into eight new patch data objects corresponding to the eight newly formed patches.

3.4.4 *Collecting information on ranks*

The *pack index* is a list containing necessary information indicating whether a given patch a must reside in the same MPI rank as another patch b. After having executed patch splitting, we then go through the list of merge operations along the MPI ranks. We use an identifier that denotes the parent of the eight merging children patches. With the exception of the first child, all the other children patches have their pack index set to the index of the first child in the global metadata list. This signifies that the seven other children patches must be in the same MPI rank as the first one, which enables the merge operation to be conducted at a later stage. The pack index is used during the subsequent load balancing step.

3.4.5 *Load balancing*

Performing load balancing consists of grouping patches in chunks, and distributing the chunks appropriately over the MPI ranks for their computational charge to be as homogeneous as possible. Load balancing is performed in four sub-steps:

- The load balancing module receives a list of metadata that includes estimated computational loads. Here, a strategy for patch

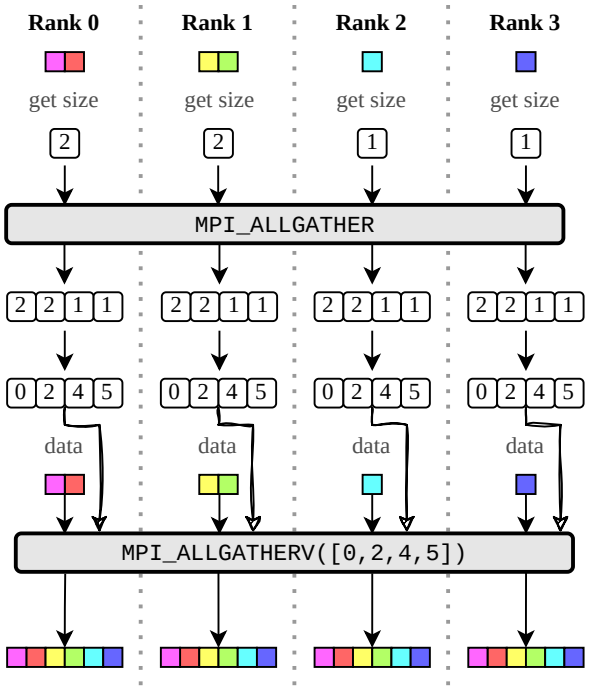

Figure 4. Illustration of the steps performed in a `vector_allgatherv` operation. Firstly, the size of each sent vector is retrieved. Secondly, all MPI ranks gather the sizes sent by each other MPI ranks. An exclusive scan is performed on this list to obtain the offset at which the data of each MPI rank will be inserted in the final vector. Finally, a `MPI_ALLGATHER_V` is called using the provided offsets to retrieve the gathered list in each MPI rank.

reorganisation of patches is computed (see Sect. 3.5 for the details of the load balancing procedure). The code returns a list specifying the novel MPI rank assignment for each patch. When compared to the current owner of each patch, this list identifies necessary changes, indicating if a patch must move from one MPI rank to another. Additionally, this list incorporates the pack index described above.

- The patch reorganisation encoded in the list of changes is subsequently implemented. We iterate through the list of changes a first time. If the sender MPI rank matches the MPI rank of the current process, it sends the corresponding patch data using a non-blocking send of the serialized data (see 3.7 for details).
- We then go through the list of changes for a second time. If the receiver patch matches the MPI rank of the current process, we execute an MPI non-blocking receive operation to obtain the corresponding patch data in the new rank.
- Finally, we finish by waiting for all MPI operations to complete, thereby concluding the load balancing step.

Generating the list of changes accounts for the pack index. As such, patches intended for merging together are in the same MPI rank after the load balancing operation.

3.4.6 Patch merging

Merge operations require for the eight children patches to be in the same MPI rank, which is garanteed by the packing in the load balancing step. Similarly to split operations, merges are executed across both metadata and the patch tree within all MPI ranks. Merging is also applied to the patch data on the MPI rank that owns the data.

3.5 Load balancing strategies

The load balancing module generates the list of owner of each patch determined by abstract estimates of its required computational load. Load balancing is processed consistently across all MPI ranks for identical inputs. The load balancing module initially utilises the list of all patch metadata, with the estimated load values as input. Patches are then arranged along a Hilbert curve, which is subsequently segmented into contiguous chunks of adjacent patches. The objective of optimal load balancing is to identify a collection of chunks wherein the workload is distributed as evenly as possible across all MPI ranks.

To achieve this, various load balancing strategies are dynamically evaluated in Shamrock (e.g. analytic decomposition, round-robin method), and the one found to be the most effective is selected. The computational overhead involved in assessing the benefits of different load balancing strategies is minimal, since it relies on simple estimations. This process yields a list specifying the new MPI ranks for each patches. This list is compared against the current distribution of patches to generate the change list when load balancing is applied.

3.6 Patch interactions

3.6.1 Interaction criteria

For a given collection of objects (cells or particles), we can establish a condition indicating whether objects i and j interact, and define a Boolean interaction criterion $\gamma_{o/o}(i,j)$ to signify this condition. For example, in the Smoothed Particle Hydrodynamics method, $\gamma_{o/o}$ is defined as true when particle i is within the interaction radius of particle j, or vice versa.

A first generalisation of this object-object criterion is an object-group criterion, which describes if there is interaction between an object and a group of objects. A necessary condition for such a criterion is

$$\gamma_{o/g}(i,\{j\}_j) \Leftarrow \bigvee_j \gamma_{o/o}(i,j)$$

This condition formally expresses the fact that if the interaction criterion is fulfilled for any object in the group, it must also hold true for the entire group. Failure to meet this condition would imply the possibility of interaction with an element of the group without interaction with the group as a whole, which is incorrect. Both the object-object and group-object criteria are used in the tree traversal step. 4.12. Another extension of the aforementioned criteria is the group-group interaction criterion, which similarly satisfies the following condition

$$\gamma_{g/g}(\{i\}_i,\{j\}_j) \Leftarrow \bigvee_i \gamma_{o/g}(i,\{j\}_j)$$

This latter condition is used to manage ghost zones (see Sect. 3.6.2) and perform two-stages neighbour search (see Sect. 4.14).

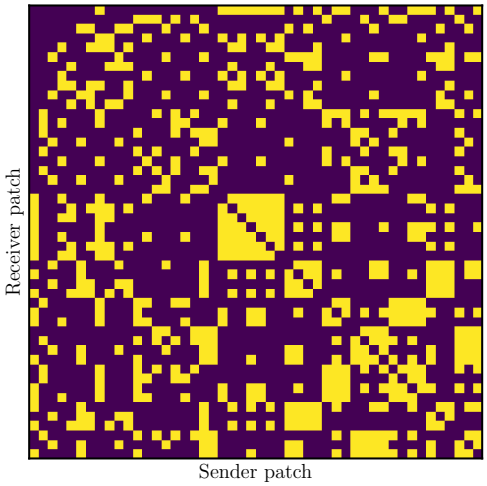

Figure 5. Matrix of the interaction graph between patches extracted from an SPH simulation of a protoplanetary disc, involving several hundred patches. Empty patches have been excluded from the graph as they do not meet any interaction criteria due to their emptiness. The resulting interaction matrix is symmetrical and sparse. This visualisation was generated using a debug tool in Shamrock, which creates a dot graph representing the ghost zones of the current time step, which can be rendered in its matrix form here showed.

3.6.2 *Interaction graph*

Patches themselves are objects that can interact. Their interaction is handled using a group-group interaction criterion $\gamma_{g/g}$. The interaction criterion of an empty patch is always false. After assessing the interaction status between all pairs of patches, we define the *interaction graph* of the patches by considering the list of links such that the group-group interaction criterion $\gamma_{g/g}$ is true. Fig. 5 shows an example of such a graph of interactions between patches.

3.6.3 *Interfaces and ghost zones patch*

We define the *interface between two patches* as the smallest set of individual objects for which the group-group interaction criterion between their parent patches is satisfied. To reduce communications between interacting patches, we communicate not the entire patch content to its neighbour, but only their interfaces. These communicated interfaces consequently manifest as ghost extensions for the neighbouring patch and are therefore called *ghosts zones* of patches. The graph of ghost zones between patches is the same as the interaction graph, with links between vertices representing the ghost zone of one patch being sent to another patch.

3.7 Serialisation

In Shamrock, all communications are serialised, i.e. converted into a stream of bytes to reduce the MPI overhead by performing less operations, and shield the user from the MPI layer. To send a patch ghost zone, data are initially packed into a byte buffer. Communication patterns and operations remain therefore unchanged, regardless of the communication content. In particular, the addition of a field simply adds extra data to the serialisation without altering the communication process.

```
// Data to be serialized
std::string str = "exemple";
sycl::buffer<f64_3> buffer = ...;
u32 buf_size = buffer.size();

SerializeHelper ser;

// Compute byte size of header and content
SerializeSize bytelen =
      ser.serialize_byte_size<u32>()
    + ser.serialize_byte_size<f64_3>(buffer.size())
    + ser.serialize_byte_size(test_str);

// Allocate memory
ser.allocate(bytelen);

// Write data
ser.write(buf_size);
ser.write_buf(buffer, n2);
ser.write(test_str);

// Recover the result
sycl::buffer<u8> res = ser.finalize();
```

```
// The byte buffer
sycl::buffer<u8> res = ...;

// Give the buffer to the helper
shamalgs::SerializeHelper ser(std::move(res));

// Recover buffer size
u32 buf_size;
ser.load(buf_size);

// Allocate buffer and load data
sycl::buffer<f64_3> buf (buf_size);
ser.load_buf(buf, buf_size);

// Read the string
std::string str;
ser.load(recv_str);
```

Serialisation in Shamrock relies on a split header data approach. Individual values are stored in the header on the CPU, while buffer data is stored on the device (CPU or GPU). This organisation ensures that individual value reads incurs minimal latency, thus avoiding high GPU load latency. The entire buffer is only assembled on the device at the end of the serialisation procedure. During deserialisation, the header is initially copied to the CPU. To circumvent constraints imposed by the CUDA backend, all reads and writes are adjusted to 8-byte length.

3.8 Sparse MPI communications

In hydrodynamical simulations, interactions among objects are predominantly local, resulting in each patch being connected to only a limited number of other patches in the interaction graph. A crucial element of communication management in Shamrock is to uphold this sparsity. With synchronised metadata, each MPI ranks holds information of the MPI rank to which every patch belongs. We therefore group communication between patches involving the same pair of MPI ranks in a single *patch message* (see Fig. 6). The graph corresponding to patch messages to be communicated is also sparse (rank $i \mapsto$ rank j). We therefore apply a MPI operation that extends `MPI_Alltoall` to accommodate a sparse graph structure. The operation, referred to as *sparse all-to-all*, is structured as depicted in Fig. 7. Initially, we compile the list of communications to be ex-

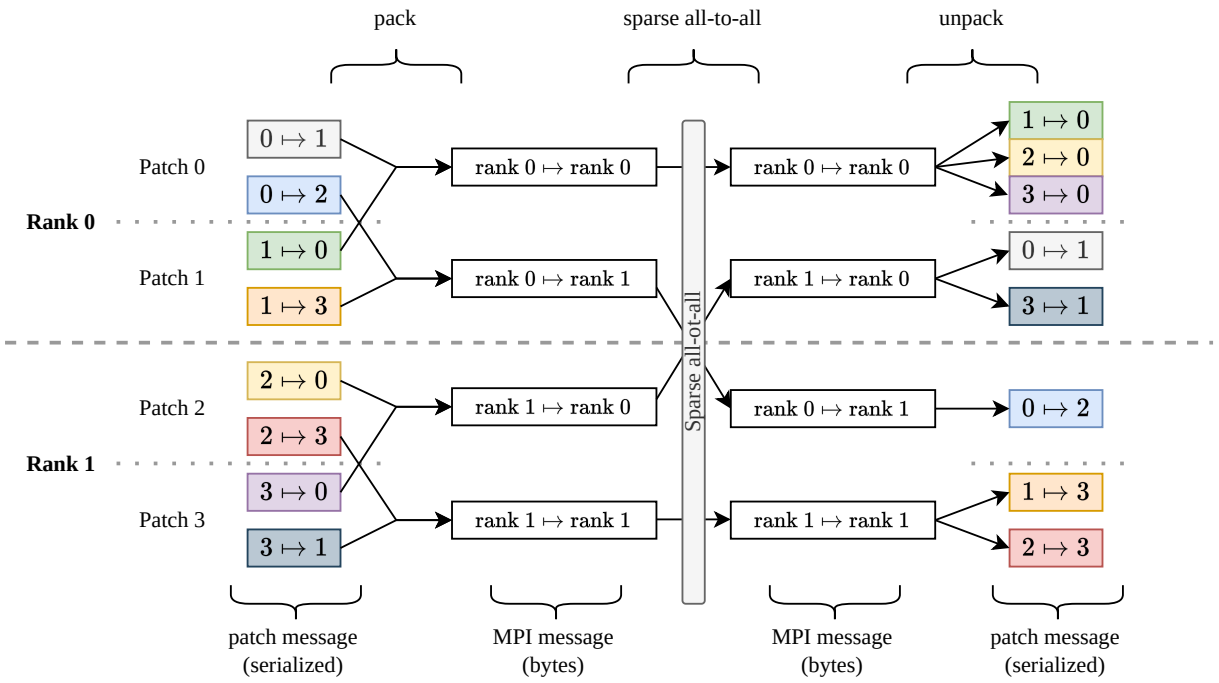

Figure 6. Illustration of the behaviour of a MPI sparse communication of patches in Shamrock. The first step consists of packing communication between a same pair of MPI ranks together. Subsequently, a sparse all-to-all operation is executed (see Fig.7). Finally, the received buffers are unpacked.

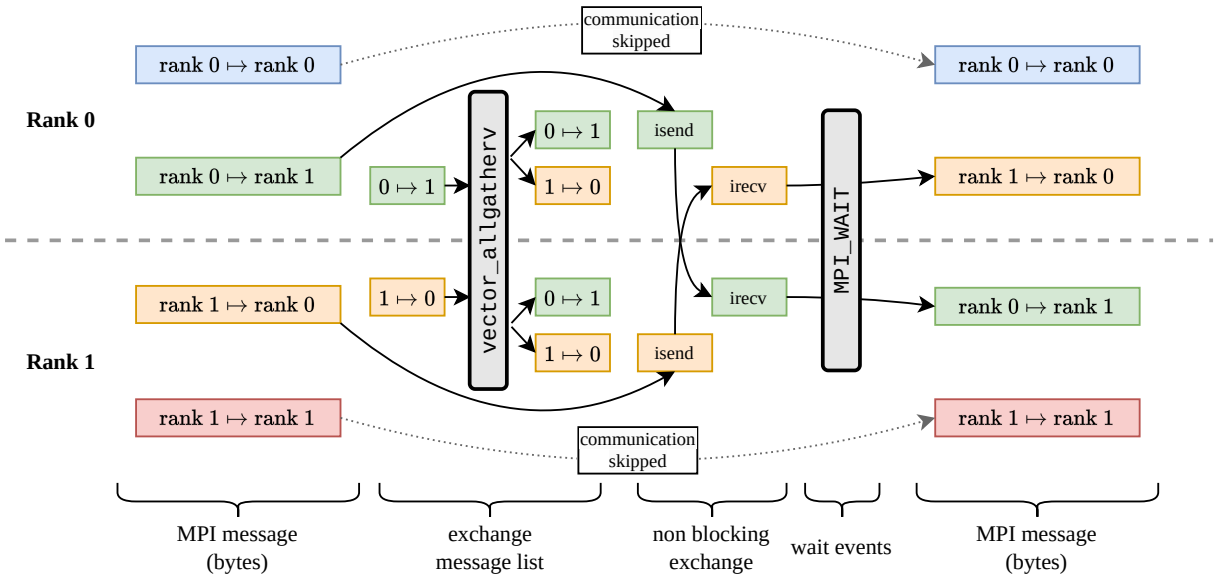

Figure 7. Illustration of the behaviour of a MPI sparse all-to-all communication in Shamrock. Firstly, a `vector_allgatherv` is performed on the list of communication. Subsequently, each rank executes a non-blocking send of its data. To prepare for receiving, a non-blocking receive is launched for every incoming message. The operation is concluded by waiting for all non-blocking operation to finish. Communications for an MPI rank to itself are skipped by simply relocating the data within the rank.

ecuted on each node. Subsequently, on each node, we go through the communication list and execute a non-blocking MPI send if the sender's rank matches the current MPI rank. Following this, on each node, we go through the communication list once more and initiate a non-blocking MPI receive if the recipient's rank aligns with the current MPI rank. Finally, we conclude the operation by invoking a MPI wait on all non-blocking communication requests. Exchanges within the patch ghost zone graph are finalised once the sparse all to all operation is completed. If the sender's MPI rank matches that of the recipient, the communication is disregarded, and an internal memory move is executed instead. Another approach could involve using an MPI reduce operation to count the number of messages received, and trigger the corresponding number of non-blocking receives with `MPI_ANY_SOURCE`. Given the limited number of communications, we observe no practical distinction between the two methods in practice. Moreover, the former approach is easier to debug and optimise,

Table 1. List of symbols used in Sect. 4.

Symbol	Definition	Meaning
x, y, z	Sect. 4.1	particle coordinates
X, Y, Z	Sect. 4.1	integer particle coordinates
β	Sect. 4.1	bit count
$\mathtt{X}_0\mathtt{X}_1\cdots\mathtt{X}_{\beta-1}$	Sect. 4.1	binary representation of X
m	$\mathtt{X}_0\mathtt{Y}_0\mathtt{Z}_0\cdots$	generic Morton code
$m_1 \equiv 0101$	Sect. 4.2	Morton code example 1
$m_2 \equiv 0111$	Sect. 4.2	Morton code example 2
$\delta\,(a, b)$	eq.2	Karras δ operator
$\mathtt{clz}(a)$	Sect. 4.4	count leading zeros
$a\hat{}b$	Sect. 4.4	bitwise XOR operator
a & b	Sect. 4.5	bitwise AND operator
$a \ll b$	Sect. 4.5	left bitshift operator
$\mathbf{r}_i$		position of particle i
m_i		Morton code of particle i
$\{\mu_i\}_i$	$\mu_i = m_{\epsilon_i}$	sorted Morton codes
ϵ_i	sort : $\epsilon_i \mapsto i$	sort inverse permutation
ξ_i	Sect. 4.9	Morton-keep mask
$\mathtt{id}_i$		indexes of kept Morton codes
$\mu_{\mathrm{leaf},i}$		tree leaf Morton codes

since it eliminates the need for sorting data to ensure determinism in the list of received messages.

4 THE SHAMROCK TREE

Specific notations used in this Section are given in Table 1.

4.1 Morton codes

In hydrodynamic simulations, physical fields are represented on a discrete set of elementary numerical elements such as grid cells or interpolation points (or *numerical objects* for a generic terminology). The positions of these objects are represented by coordinates, usually stored as floating point numbers such as (x, y, z) in three dimensions. These coordinates are usually sampled on a 3D integer grid, which in turn can be mapped onto a 1D integer fractal curve. The Morton space-filling curve, also called *Morton ordering*, is commonly used for this purpose since it has a natural duality with a tree structure (e.g. Samet 2006, see below). In practice, Morton ordering can be constructed from a list of 3D positions as follows. First, the real coordinates in each dimension are remapped over the interval $[0, 1)^3$ (note the exclusion of the value 1) by doing $x \mapsto (x - x_{\min})/(x_{\max} - x_{\min})$, and a similar procedure is applied for y and z respectively. This unit cube is then divided into a 3D grid of $(2^\beta)^3$ elements, where β is the number of bits used to represent integers. Within this grid, the objects possess integer coordinates $(X, Y, Z) \in [0, 2^\beta - 1]^3$. These integer coordinates are noted in their binary representation $X = \mathtt{X}_0\mathtt{X}_1\mathtt{X}_2\cdots$, where $\mathtt{X}_i$ denote the value of the i^{th} bit (the same convention also applies for Y and Z). The Morton space-filling curve comprises a sequence of integers, called *Morton codes* (or Morton numbers), defined through the following construction in a binary basis: the Morton code m of each object is obtained by interleaving the binary representation of each coordinate $m \equiv \mathtt{X}_0\mathtt{Y}_0\mathtt{Z}_0\mathtt{X}_1\mathtt{Y}_1\mathtt{Z}_1\mathtt{X}_2\mathtt{Y}_2\mathtt{Z}_2\cdots\mathtt{X}_{\beta-1}\mathtt{Y}_{\beta-1}\mathtt{Z}_{\beta-1}$.

By default, and unless specified otherwise, Morton codes are presented in binary notation, while other integers are expressed in decimal hereafter. A Morton code can also be interpreted as an ordered position on an octree with $\beta+1$ levels, or alternatively as a position in a binary tree with $3\beta+1$ levels (Fig. 8). To illustrate this duality, let us consider the first bit $\mathtt{X}_0$ of a Morton code. If $\mathtt{X}_0 = 0$, the integer coordinate X belongs to the half space where $X < 2^{\beta-1} - 1$. If $\mathtt{X}_0 = 1$, the integer coordinate X belongs to the other half space $X \geq 2^{\beta-1} - 1$. The following bits Y_0 and Z_0 divide the other dimensions in a similar way. The next sequence of bits X_1, Y_1, Z_1 subdivides the subspace characterised by X_0, Y_0, Z_0 in a similar manner, and the construction of a tree follows recursively. After going through all bits and reaching $\mathtt{X}_{\beta-1}\mathtt{Y}_{\beta-1}\mathtt{Z}_{\beta-1}$, one is left with the exact position in the space of integer coordinates. This tree structure consists of nested volumes where each parent volume encompasses all its children, forming as such a Bounded Volume Hierarchy (BVH).

4.2 Prefixes

A *prefix* is the sequence of the first $\gamma \leq \beta$ bits of a Morton code. One defines the *longest common prefix* of two Morton codes a and b as the sequence of matching bits starting from X_0 until two bits differ. As an example, the longest common prefix of $m_1 \equiv \mathbf{01}01$ and $m_2 \equiv \mathbf{01}11$ is $\mathbf{01}$. The longest common prefix of two Morton codes gives the minimal subspace of the integer 3D grid that contains the two Morton codes. The number of bits used to represent the longest common prefix of a and b, called the *length of the longest common prefix of a and b*, is denoted $\delta\,(a, b)$.

4.3 Bounding boxes

We use the terminology *prefix class* to refer to a set of Morton codes that have common prefixes. The longest common prefix of any pair of elements in a prefix class is at least of length γ (or equivalently, for any pair of Morton code a, b in the prefix class, $\delta(a, b) \geq \gamma$).

Each prefix class corresponds to an axis aligned bounding box in the space of integer positions, having for generic coordinates $[x_{\min}, x_{\max}) \times [y_{\min}, y_{\max}) \times [z_{\min}, z_{\max})$ (Fig. 8). We refer to the set of three integers representing the lengths of the edges of this bounding box as *the size of the bounding box*. Mathematically,

$$\mathbf{s}(\gamma) = \left\{2^{\beta-\lfloor\gamma/3\rfloor}, 2^{\beta-\lfloor(\gamma-1)/3\rfloor}, 2^{\beta-\lfloor(\gamma-2)/3\rfloor}\right\}, \tag{1}$$

where $\lfloor\cdot\rfloor$ denotes the floor function of a real number. Indeed, for a given γ, the Morton construction divides the x-axis $\lfloor\gamma/3\rfloor$ times, the y-axis $\lfloor(\gamma-1)/3\rfloor$ times and the z-axis, $\lfloor(\gamma-2)/3\rfloor$ times. The exclusion of the upper bounds in the bounding box ensures that the size on each coordinate axis is a power of 2. Similarly, we define the largest common prefix class between two Morton codes a, b as the prefix class corresponding to the longest common prefix between a and b. The size of the corresponding bounding box is then denoted $\mathbf{s}(a, b) = \mathbf{s}\,(\delta(a, b))$.

4.4 Longest common prefix length

The length of the longest common prefix of two Morton codes a and b is given by (Karras 2012)

$$\delta\,(a, b) \equiv \mathtt{clz}\,(a\hat{}b)\,. \tag{2}$$

Eq. 2 involves two binary operators. The first one is the bitwise XOR $\hat{}$ operator (Exclusive OR), that returns the integer formed in binary by zeros where the bits match and ones when they differ. As an example,

$$m_1\hat{}m_2 = 0010, \tag{3}$$

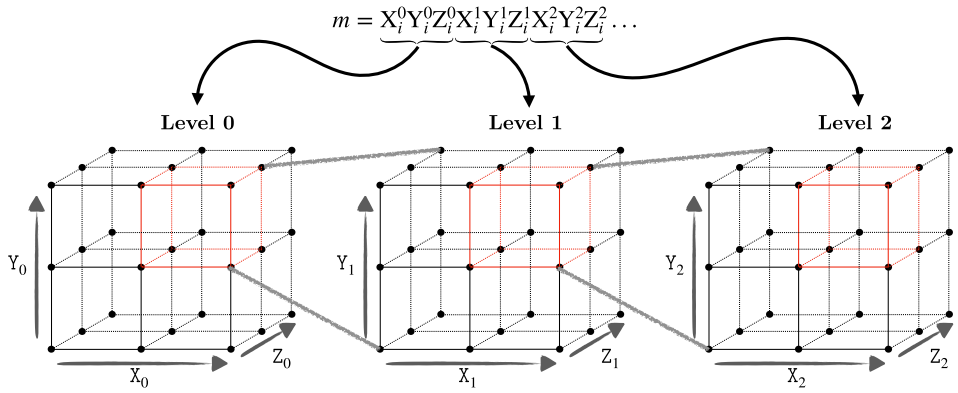

Figure 8. Illustration of the duality between Morton codes and the structure of an octree. 3 bits can describe the procedure of dividing a cube into eight smaller cubes. Repeating the procedure with triplets of additional bits produces an octree.

since m_1 and m_2 differ only by their third bit. The second operator is Count Leading Zeros. `clz` operates on a binary integers and returns the numbers of zeros preceding the first 1 in the binary representation. As an example,

$$\texttt{clz}(0010) = 2. \tag{4}$$

Following this example, the longest common prefix of $m_1 \equiv \mathbf{01}01$ and $m_2 \equiv \mathbf{01}11$ is **01** and is of length 2. Eqs. 3–4 allow performance, since instructions `clz` and `XOR` use only one CPU or GPU cycle on modern architectures. Getting the length of the longest common prefix take only 2 cycles with such procedure (e.g. a `xor` followed by `lzcnt` on Intel Skylake architectures).

4.5 Finding common prefixes

To find the longest common prefix between two Morton codes a and b, we first construct a mask c, which is an integer where the first $p = \delta(a, b)$ bits are set to 1 while the remaining bites are set to 0. For example, applying this mask to the two Morton codes m_1 and m_2 from our previous example yields 1100. To generate the mask, we take advantage of the bitwise shift-left operator. The bitwise shift-left operator $a \ll i$ returns the binary representation of a where the bits are shifted by i^{th} bits to the left, and zeros are introduced in place of non existing bits. Consider u, the integer having only ones in binary representation (i.e $u = 2^\beta - 1$, where β is the size of the binary representation). c is obtained with the following binary operation $u \ll (\beta - \delta(a, b))$. In our previous example, $\beta = 4$ and $\beta - \delta(m_1, m_2) = 2$ gives $1111 \ll 2 = 1100$. Consider now the bitwise `AND` operator, denoted by &, that returns the integer formed in binary by ones where the bits match and zeros when they differ (& is the bitwise negation of the bitwise `XOR` operator). When applying the bitwise `AND` between m_1 or m_2 and the mask, the result is a binary number where the first bits are the prefix and the subsequent bits are zeros. As an example, applying the bitwise `AND` between m_2 and the mask yields $\mathbf{01}11$ & $1100 = \mathbf{01}00$.

4.6 Getting coordinates sizes of bounding boxes

Consider the prefix class formed by Morton codes whose longest common prefix with a (or equivalently b) is $\delta(a, b)$. This prefix class is a set of binary numbers whose smaller and larger values, denoted p_0 and p_1 respectively, are given by

$$p_0(a, b) \equiv \left(2^\beta - 1 \ll \beta - \delta(a, b)\right) \;\&\; a, \tag{5}$$

$$p_1(a, b) \equiv \left(2^\beta - 1 \ll \beta - \delta(a, b)\right) \;\&\; a + \left(2^{\beta - \delta(a,b)} - 1\right). \tag{6}$$

These two Morton codes correspond to two integer coordinates, denoted $\mathbf{p}_0$ and $\mathbf{p}_1$, that are the coordinates of the lower and upper edges of the bounding box, respectively. The size of the bounding box corresponding to this prefix class is

$$\mathbf{s}(a, b) = \mathbf{p}_1(a, b) - \mathbf{p}_0(a, b) + (1, 1, 1). \tag{7}$$

4.7 Binary radix tree

A binary radix tree is a hierarchical representation of the prefixes of a list of bit strings, corresponding here to the binary representation of integers (e.g. Lauterbach et al. 2009; Karras 2012). The tree is defined by a set of hierarchically connected nodes, where nodes without children are called leaf nodes or *leaves* (light orange circles on Fig. 9), and the others ones are called *internal nodes* (blue circles on Fig. 9). The binary radix tree is a complete binary tree: every internal node has exactly two children. As such, a tree having n leaves has exactly $n - 1$ internal nodes. This property allows to know lengths of tables in advance, making it particularly beneficial for GPU programming where dynamic allocation is not feasible. One commonly acknowledged downside of this tree structure is the challenge of efficient hierarchical construction (Lauterbach et al. 2009; Karras 2012). The deeper one goes down the tree, the longer the length of the common prefix of the Morton codes. The corresponding bounding boxes for each node in the tree are nested and become progressively smaller as one goes down the tree, while the length of the common prefixes increases. The corresponding radix tree forms a Bounding Volume Hierarchy, since each child in the bounding box is contained within the box of its parents.

4.8 Karras algorithm

The Karras algorithm overcomes this difficulty with a fully parallel algorithm that constructs a binary radix tree, in which the list of bit strings is a sorted list of Morton codes without any duplicates Karras (2012). Fig. 9 shows a typical binary radix tree constructed by the Karras algorithm. The integers within the light orange circles represent the indices of the Morton codes in the sorted list, and they also are the indices of the corresponding leaves in the tree. The

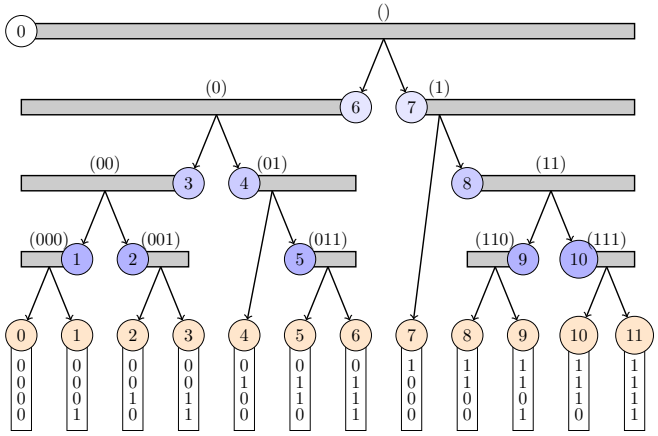

Figure 9. The Karras Algorithm associates a structure of radix tree to a sorted list of unduplicated Morton codes (depicted here within rectangles). Within this tree, nodes can be either leaves, denoted by integers in light orange circles, or internal nodes, indicated by integers in blue circles. The grey bars represent the ranges of Morton codes covered by each internal node. Common prefixes of these Morton codes are shown in brackets.

integers within the blue circles denote the indices of the internal nodes, their value come as a by-product of the algorithm. The grey bars denote intervals of leaf indices corresponding to any Morton code contained in the sub-tree beneath an internal node that shares the same common prefix. In Fig. 9, these prefixes are shown in brackets over the grey bars. The deeper the position in the tree, the longer the prefix. Consider the subset of Morton codes associated to an internal node. The Karras algorithm divides this list in two sub-lists (arrows Fig. 9), each with a longer common prefix compared to the original. Such a split is unique. Several tables are associated to the construction of Fig. 9. Those are presented on Table 2. The split associated to an internal node provides two numbers called the indices of the left and right children respectively, denoted as `left-child-id` and `right-child-id` respectively. *A priori*, these indices can correspond to either an internal node or a leaf. This distinction is encoded by the value the integers `left-child-flag` and `right-child-flag`, where a 1 means that the corresponding child is a leaf and a 0, an internal node. The grey bar of an internal node has two ends. One corresponds to the index of the node itself, while the other is stored in `endrange`. Although this value is of no use in the construction of the tree itself, will be important later for calculating the sizes of a bounding box associated to a prefix class and for iterating over objects contained in leafs. The Karras algorithm performs dichotomous searches to compute the values of Table 2 in parallel, with no prerequisites other than the Morton codes (we refer to the pseudo-code of the algorithm in Karras 2012 for details). Its efficiency relies firstly on the ability to pre-allocate tables before building the tree, and secondly on the sole use of the δ operator defined in Sect. 4.4, which requires just 2 binary operations on dedicated hardware.

4.9 Removal of duplicated codes

As mentioned in Sect. 4.8, the Karras algorithm requires a sorted list of Morton codes without duplicates (line 1-5 in alg.1). To achieve this, we go through the list of sorted Morton codes and compute a mask to select the Morton codes to retain. The list of Morton codes without duplicates corresponds to the leaves of the tree obtained after applying the Karras algorithm to construct the radix tree.

Internal cell id	0	1	2	3	4	5	6	7	8	9	10
`left-child-id`	6	0	2	1	4	5	3	7	9	8	10
`right-child-id`	7	1	3	2	5	6	4	8	10	9	11
`left-child-flag`	0	1	1	0	1	1	0	1	0	1	1
`right-child-flag`	0	1	1	0	0	1	0	0	0	1	1
`endrange`	11	0	3	0	6	6	0	11	11	8	11

Table 2. Tables corresponding to the tree shown on Fig. 9 as returned by the Karras algorithm.

Algorithm 1: Removal mask initialisation and reduction algorithm

Data: $\{m_i\}_{i\in[0,n)}$ The morton codes.
Result: $\{\xi_i\}_{i\in[0,n)}$ The mask list.

```
// Flag removal of duplicates
1  for i in parralel do
2      if i == 0 then
3          ξ_i ← true;
4      else
5          ξ_i ← not(m_i = m_{i-1});

// Reduction passes
6  for n_red reduction steps do
7      for i in parralel do
           // Get kept morton codes indexes
8          i_{-1} ← i − 1;
9          while (ξ_{i_{-1}} = false & i_{-1} ≥ 0) do
10             i_{-1} ← i_{-1} − 1;
11         i_{-2} ← i_{-1} − 1;
12         while (ξ_{i_{-2}} = false & i_{-2} ≥ 0) do
13             i_{-2} ← i_{-2} − 1;
14         i_{+1} ← i + 1;
15         while (ξ_{i_{+1}} = false & i_{+1} < N_morton) do
16             i_{+1} ← i_{+1} + 1;
           // Reduction criterion
17         δ_0 ← δ(μ_i, μ_{i_{+1}});
18         δ_{-1} ← δ(μ_{i_{-1}}, μ_i);
19         δ_{-2} ← δ(μ_{i_{-2}}, μ_{i_{-1}});
20         if not(δ_0 < δ_{-1} & δ_{-2} < δ_{-1}) & ξ_i = true then
21             ξ_i ← true;
22         else
23             ξ_i ← false;
```

4.10 Reduction

In certain situations, an object may interact with a large number of neighbours, resulting in multiple leaves containing these neighbours for the object. One such situation arises frequently in a Smoothed Particle Hydrodynamics solver, where each particle typically interacts with an average of $\sim$ 60 neighbours. One optimisation strategy to speed up the tree traversal consists in reducing the number of leaves containing these 60 neighbours by grouping some leaves at the lower levels of the tree before applying the Karras algorithm. We have integrated a so-called step of *reduction* to achieve this. The resulting tree mirrors the initial one, but with grouped leaves.

Algorithm 2: Leaf object iteration

Data: $\texttt{id}_i$ The leaf index map.
i the leaf index we want to unpack

for $j \in [id_i, id_{i+1})$ **do**
  $k \leftarrow \epsilon_j$ // index map of the sort
  $\mathcal{F}(k)$

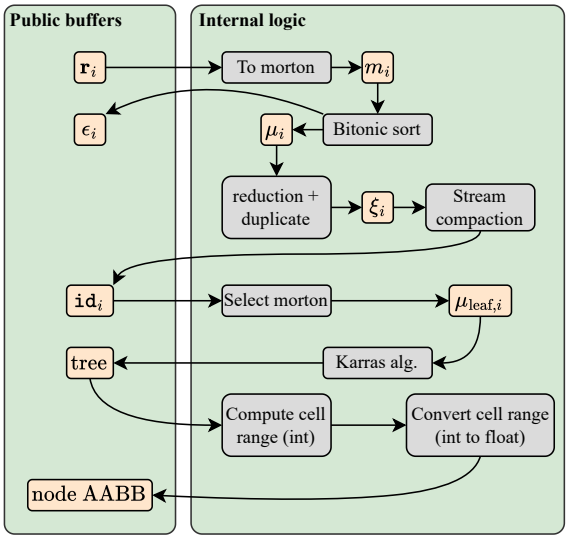

Figure 10. Flowchart illustrating the tree-building procedure, indicating the interdependence between each algorithm (grey boxes) and the related buffers (orange boxes). The internal logic box corresponds to the part of the algorithm inaccessible to the user. Buffers depicted outside this box are structures used in other parts of the code.

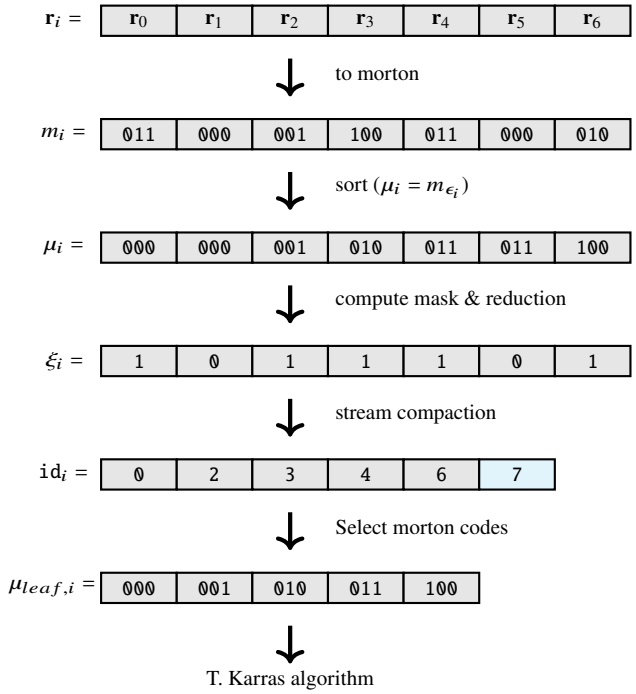

Figure 11. The cyan slot in the $\texttt{id}_i$ row is the total lenght of the input array. ϵ_i is the resulting permutation applied by the sort algorithm.

To perform reduction, we require a criterion determining when two leaves, each containing two Morton codes, can be removed to yield the internal cell positioned just above them. This procedure is carried out using Alg.1: if a Morton code constitutes the second leaf of a shared parent, then it is removable. This property is implemented in the radix tree by verifying when $\delta(\mu_{i-2}, \mu_{i-1}) < \delta(\mu_{i-1}, \mu_i) > \delta(\mu_i, \mu_{i+1})$. When this condition is satisfied, the Morton code i is removable. The reduction step modifies the Morton tree list associated to the initial tree built. The tree is therefore already reduced when it is built and has never had any additional nodes.

4.11 Tree building

Fig. 10 outlines the tree building algorithm of Shamrock. initially, Morton codes are generated from coordinates and efficiently sorted while eliminating duplicates. Morton tables are then prepared and pre-processed (a summary of these steps is sketched in Fig. 11) before filling the values characterising the tree as in Table 2. The lengths associated to the coordinates of the cells are finally calculated. The algorithms described in this section are implemented using C++ metaprogramming, enabling versatile use of any kind of spatial coordinates in practice.

Compute Morton codes

Morton codes are calculated entirely in parallel (step `"To Morton"` in Fig.10). Initially, a buffer storing the positions of the elementary numerical elements is allocated. These positions are mapped to an integer grid following the procedure described in Sect. 4.1. The construction is tested by appropriate sanity checks. The resulting integer coordinates are converted to Morton codes in a Morton code buffer (m_i in Fig.10).

Sort by Key

Initially, the list of Morton codes corresponding to the positions of elementary numerical elements is unsorted. A key-value pair sorting algorithm is therefore used to sort the Morton codes while keeping track of the original index of the object within the list. For this task, we use a GPU Bitonic sorting algorithm that we have re-implemented using `Sycl`. The Bitonic algorithm is simple and its performance is not heavily reliant on the hardware used (step `"Bitonic sort"` in Fig.10, see e.g. Batcher 1968; Nassimi & Sahni 1979). While more efficient alternatives have been suggested in the literature, our observation is that they are more difficult to implement and are not as portable across architectures (e.g. Arkhipov et al. 2017; Adinets & Merrill 2022).

Reduction

From the sorted list of Morton codes, we remove duplicates and apply reduction with a procedure in two steps. In the first step, we generate a buffer of integers where each value is 1 if the Morton code is retained at a given index and 0 otherwise. This information is stored in a buffer called Keep Morton flag buffer (ξ_i in Fig.10). In the second step, we use this buffer to perform a stream compaction algorithm (e.g. Blelloch 1990; Horn 2005, see example in Fig. 11) to construct simultaneously two lists: a list of Morton codes without duplicates, and the list of the indices of the preserved Morton code prior stream compaction. The stream compaction algorithm heavily depends on an internal exclusive scan algorithm. This algorithm, when applied

to the array $\{a_i\}_{i\in[0,n]}$ returns the array $\{\sum_{j=0}^{i-1} a_j\}_{i\in[1,n]}$ and 0 when $i = 0$. In our case, we implemented the single-pass prefix sum with decoupled look-back algorithm (Merrill & Garland 2016).

Compute tree tables

At this point, we have a set of Morton codes sorted without duplicates. We then apply the Karras algorithm described in Sect. 4.8 to generate in parallel the tables from which the properties of the tree can be reconstructed (listed on Table 2).

Compute tree cell sizes

We define a *tree cell* as the bounding box that corresponds to the Morton codes of the leaves under a given node. This node can either be an internal node or a leaf. Tree cells are therefore the geometric representation of the tree, and needs to be computed for neighbour finding (see Sect. 4.12). In practice, it is sufficient to compute the boundaries of the edges of the cell $[x_{\min}, x_{\max}) \times [y_{\min}, y_{\max}) \times [z_{\min}, z_{\max})$ (Sect. 4.6).

Algorithm 3: Compute tree cell sizes

Data: `morton` The morton code buffer.
Result: `bmin`, `bmax`, the bounds of the cells.

1 $m_1 = \texttt{morton}[i]$
2 $m_2 = \texttt{morton}[\texttt{endrange}[i]]$
3 $\sigma = \delta_{\text{karras}}(m_1, m_2)$
4 $\mathbf{f}_0 = \mathbf{s}(\sigma)$
5 $\mathbf{f}_1 = \mathbf{s}(\sigma + 1)$
6 $\texttt{mask} = \texttt{maxint} << (\texttt{bitlen} - \sigma)$
7 $\mathbf{p}_0 = (\text{morton} \rightarrow \text{real space})\ (\texttt{m}[i]\ \&\ \texttt{mask})$
8 $\texttt{bmin}[i] = \mathbf{p}_0$
9 $\texttt{bmax}[i] = \mathbf{p}_0 + \mathbf{f}_0$
10 **if** *left child flag*$[i]$ **then**
11 $\quad \texttt{bmin}[\texttt{rid}[i] + N_{\text{internal}}] = \mathbf{p}_0$
12 $\quad \texttt{bmax}[\texttt{lid}[i] + N_{\text{internal}}] = \mathbf{p}_0 + \mathbf{f}_1$
13 **if** *right child flag*$[i]$ **then**
14 $\quad \texttt{tmp} = \mathbf{f}_0 - \mathbf{f}_1$
15 $\quad \texttt{bmin}[\texttt{rid}[i] + N_{\text{internal}}] = \mathbf{p}_0 + \texttt{tmp}$
16 $\quad \texttt{bmax}[\texttt{lid}[i] + N_{\text{internal}}] = \mathbf{p}_0 + \texttt{tmp} + \mathbf{f}_1$

Alg. 3 provides the procedure to compute the size of tree cells, using the vector position $\mathbf{s}$ defined by Eq. 7 and the quantities p_0 and p_1 defined by Eqs. 5 – 6. For internal cells that have leaves as children, the boundary of the edges can be calculated by incrementing the value of $\delta(a, b)$ by one unity and using the new value in Eqs. 5 – 6. This gives the expected result for a left child, an extra shift being added for the right child (cf. line 14 of Alg.3).

4.12 Tree traversal

Each cell, leaf or internal of the tree constructed by the procedure described above consists of an axis-aligned bounding boxes and containing several numerical objects. Searching for the neighbours of an object a therefore requires checking the existence of an interaction between a cell of the tree c and the object a, using the object-group interaction criterion $\gamma_{o/g}(a, c)$. Per construction, if the criterion is true for a child cell, it is also true for its parent. Neighbour finding requires therefore starting from the root node and going down the tree, checking at each step whether the interaction criterion is still verified or not. The result is a set of retained tree leaves, that are likely to contain neighbours. The set of neighbours of a given object is then obtained by verifying the object-object interaction criterion on each object in each of the targeted leaves. The algorithmic procedure for these steps is detailed in Alg. 4. It is based on the property that the depth of the tree is shorter than the length of the Morton code representation. This allows a stack of known size to be used to traverse the tree, which can be added at compile time and run on the GPU since there is no dynamic memory allocation. The first step in the algorithm is to push the root node onto the stack. In each subsequent step, we pop the node on top of the stack, and we check whether or not it interacts with the object. If it does, and if it is an internal node, we push its children onto the top of the stack and move on to the next step. Otherwise, if it is a leaf, we iterate through the objects contained in the leaf (Alg. 2), and check the object-object interaction criterion for each object in the given leaf. In the source code of Shamrock, we abstract Alg. 4 under the `rtree_for`. It can be called from within a kernel on the device and can be associated with any interaction criteria. It will then provide an abstract loop over the objects found using the criteria.

Algorithm 4: Tree traversal

Data:
$depth$: The maximal tree depth, N_{inode} : The number of internal nodes in the tree, $\{lchild_{id,j}\}_{j\in[0,N_{inode})}$, $\{rchild_{id,j}\}_{j\in[0,N_{inode})}$, $\{lchild_{flag,j}\}_{j\in[0,N_{inode})}$, $\{rchild_{flag,j}\}_{j\in[0,N_{inode})}$

// Setup index stack
1 $i \leftarrow depth - 1$;
2 $s \leftarrow \{err\}_{i\in[0,depth)}$;
// Enqueue the root node
3 $s_i \leftarrow 0$;
4 **do**
$\quad$ // Pop top of the stack
5 $\quad j = s_i$;
6 $\quad s_i = err$;
7 $\quad i \leftarrow i + 1$;
$\quad$ // Check if interaction
8 $\quad \alpha \leftarrow \gamma_{o/g}(\ldots, j)$;
9 $\quad$ **if** α **then**
$\quad\quad$ // If the current node is a leaf
10 $\quad\quad$ **if** $j \geq N_{inode}$ **then**
$\quad\quad\quad$ // Iterate on objects in leaf
11 $\quad\quad\quad$ leaf object iteration(j);
12 $\quad\quad$ **else**
$\quad\quad\quad$ // Push node childs on the stack
13 $\quad\quad\quad lid \leftarrow lchild_{id,j} + (N_{inode}) * lchild_{flag,j}$;
14 $\quad\quad\quad rid \leftarrow rchild_{id,j} + (N_{inode}) * rchild_{flag,j}$;
15 $\quad\quad\quad i \leftarrow i - 1$;
16 $\quad\quad\quad s_i = rid$;
17 $\quad\quad\quad i \leftarrow i - 1$;
18 $\quad\quad\quad s_i = lid$;
19 $\quad$ **else**
$\quad\quad$ // Gravity
20 $\quad\quad$ leaf exclude case(j);
21 **while** $i < depth$;

4.13 Direct neighbour cache

Using neighbour search directly is technically feasible, but conducting it repeatedly would result in substantial costs due to its intricate logic. Moreover, executing computations within the core of a device kernel with extensive branching would negatively impact performance. To circumvent these issues, we instead build a neighbour cache when traversing the tree, and then reuse this cache for subsequent computations on the particles. The benefits are twofold: firstly, it increases performance for the reasons outlined above, and secondly, it decouples neighbour finding from calculations carried out on the particles, enabling optimisation efforts to be better targeted. Conversely, using such an approach means that we store an integer index for each pair of neighbours, which in SPH is roughly 60 times the number of particles. The memory footprint therefore increases significantly. Taking everything into account, we opt for the neighbour caching strategy due to its better performance and extensibility.

Algorithm 5: Neighbour caching

Data: N : The number of objects to build cache for, $\gamma_{o/g}$: the object-group interaction criterion, $\gamma_{o/o}$ the object object interaction criterion.
Result: $\{\xi_i\}_{i\in[0,N+1)}$ The offset map. $\{\Xi_i\}_{i\in[0,N_{neigh})}$ The neighbour id map.

$\{c_i \leftarrow 0\}_{i\in[0,N+1)}$;
// First pass to count neighbours
for $i \in [0, N)$ ***in parallel*** **do**
  $c \leftarrow 0$;
  for $j \leftarrow$ `rtree_for`$[\gamma_{o/g}(i, \dots)]$ **do**
    if $\gamma_{o/o}(i, j)$ **then**
      $c \leftarrow c + 1$;
  $c_i \leftarrow c$;
// c_i contain the neighbours counts
$\{\xi_i\}_{i\in[0,N+1)} \leftarrow$ `exclusive scan`$(\{c_i\}_{i\in[0,N+1)})$;
// ξ_i contain the neighbour map offset
$N_{neigh} \leftarrow c_N$;
$\{\Xi_i \leftarrow 0\}_{i\in[0,N_{neigh})}$;
// Second pass to get neighbours ids
for $i \in [0, N)$ ***in parallel*** **do**
  $off \leftarrow \xi_i$;
  for $j \leftarrow$ `rtree_for`$[\gamma_{o/g}(i, \dots)]$ **do**
    if $\gamma_{o/o}(i, j)$ **then**
      $\Xi_{off} \leftarrow j$;
      $off \leftarrow off + 1$;

We start by allocating a buffer to store the neighbour count for each object. We perform an initial loop over all the objects and do a tree traversal for each of them to obtain the neighbour counts. We then perform an exclusive scan, which gives the offset used to write in the neighbour index map from our neighbour count buffer. The neighbour count buffer has an extra element that is set to zero at its end, this allows us to obtain the total number of neighbours in this slot after the exclusive scan. A final loop writes the indexes of the neighbours to the neighbour index map. Details of this procedure are given in Alg. 5. We can use a procedure similar to the one used for the tree leafs in Alg. 2 to iterate over the neighbours stored in the neighbour cache, as depicted in Alg. 6.

Algorithm 6: Neighbour cache usage

Data: $\{\xi_i \leftarrow 0\}_{i\in[0,N+1)}$ the offset map, $\{\Xi_i\}_{i\in[0,N_{neigh})}$ the neighbour cache

for $j \in [\xi_i, \xi_{i+1})$ **do**
  $k \leftarrow \Xi_j$ // index of neighbour
  $\mathcal{F}(k)$

4.14 Two-stages neighbour cache

The procedure described in Sect. 4.13 consists of a direct neighbour cache, in the sense that for each object we search directly for its neighbours. A more sophisticated approach, likely to improve performance in most cases, involves splitting the direct case into two stages. In the first step, we search for the neighbours of each tree leaf using the group-group interaction criterion and the group-object criterion. In the second step, we first determine in which leaf the object is, then use the leaf neighbour cache to find the neighbour of the object. The first step only searches for neighbours within the leaves of the tree, while the second step produces the same result as in the direct case. In a two-stages neighbour search, tree traversal is performed once per tree leaf, instead of once per object. When combined with tree reduction, this approach can decrease the number of tree traversals performed by a factor of ten. On the flip side, adopting a two-stage neighbour caching approach increases the number of kernels to be executed on the device and the allocation pressure (temporarily, as the first step is discarded at the end, the memory footprint is unchanged compared to the direct case, but temporary allocation can introduce additional latency). Overall, we observe that two-stages neighbour caching generally improves computational efficiency, and when combined with tree reduction, this strategy ultimately yields the best performance.

5 SMOOTHED PARTICLE HYDRODYNAMICS IN SHAMROCK

5.1 Equations of motion

Smoothed Particle Hydrodynamics (SPH), initially introduced by Lucy (1977); Gingold & Monaghan (1977), is a Lagrangian approach widely employed in astrophysics. It is used for its capacity to handle complex geometries, adapt resolution to follow mass, address free boundary conditions, and offer an alternative approach to grid-based methods for validating nonlinear solutions. In a Lagrangian form, the equations of motion for compressible inviscid hydrodynamics are

$$\frac{\mathrm{d}\rho}{\mathrm{d}t} = -\rho\nabla\cdot\mathbf{v}, \tag{8}$$

$$\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\frac{\nabla P}{\rho} + \mathbf{f}, \tag{9}$$

$$\frac{\mathrm{d}u}{\mathrm{d}t} = -\frac{P}{\rho}\nabla\cdot\mathbf{v}, \tag{10}$$

where ρ, $\mathbf{v}$, P and u denote the density, the velocity, the pressure and the specific internal energy of the fluid respectively. The system of equations is closed through the equation of state of the fluid. In SPH, the equations Eqs. 8 – 10 are smoothed and discretised across moving interpolation points, also called SPH particles. Density estimates for

each particle a are obtained following

$$\rho_a = \sum_b m_b W_{ab}(h_a), \tag{11}$$

$$W_{ab}(h_a) = \frac{C_{\rm norm}}{h_a^3} f\left(\frac{|\mathbf{r}_a - \mathbf{r}_b|}{h_a}\right), \tag{12}$$

$\mathbf{r}_a$, m_a, and h_a represent the position, fixed mass, and smoothing length of particle a, respectively. When h_a is itself a function of the density, Eqs. 11 – 12 should be solved consistently (see Sect. 5.4). In practice, particles are semi-regularly arranged. The interpolation kernel W is a bell-shaped function that converges weakly towards a delta Dirac distribution when the smoothing length h goes to zero. $C_{\rm norm}$ is a normalisation constant for the kernel, calculated for a three dimensional domain of simulation. In Shamrock, typical functions f with finite compact supports, such as Schoenberg (1946) B-splines like M_4, M_5, M_6 , or Wendland functions like C_2, C_4, C_6 (see e.g., Wendland 1995), are implemented. Although Gaussian kernels are excellent for SPH they would be too costly for large simulations, motivating the choice of compactly supported function to ensure computational efficiency (see Morris (1996); Price (2012); Dehnen & Aly (2012) for details). We define l_a, the interaction radius of SPH particle a, as $l_a = R_{\rm kern} h_a$, where $R_{\rm kern}$ is the radius of f, the kernel generator function. Eq. 11 is a smoothed integrated form of the continuity equation Eq. 8. Eq. 11 provides an estimate of the local expansion rate of an elementary volume of the fluid dV

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathrm{d}V = (\nabla \cdot \mathbf{v})\ \mathrm{d}V, \tag{13}$$

without defining volumes explicitly. Indeed,

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{m_a}{\rho_a}\right) = -\frac{1}{\rho_a}\frac{\mathrm{d}\rho_a}{\mathrm{d}t}\left(\frac{m_a}{\rho_a}\right). \tag{14}$$

As introduced by Price (2012), evolution of internal energy Eq. 10 is subsequently calculated according to

$$\frac{\mathrm{d}u_a}{\mathrm{d}t} = \frac{P_a}{\rho_a^2}\frac{\mathrm{d}\rho_a}{\mathrm{d}t} = \frac{P_a}{\rho_a^2 \Omega_a}\sum_b m_b \mathbf{v}_{ab} \cdot \nabla W_{ab}(h_a), \tag{15}$$

$$\Omega_a = 1 - \frac{\partial h_a}{\partial \rho_a}\sum_b m_b \frac{\partial W_{ab}(h_a)}{\partial h_a}, \tag{16}$$

where $\mathbf{v}_{ab} \equiv \mathbf{v}_a - \mathbf{v}_b$, and the pressure P_a is related to the density ρ_a and other variables through the equation of state. The term Ω_a arises from the fact that h_a is chosen in practice to be a function of the density and as such, depends on the positions of the particles (Monaghan 2002; Springel & Hernquist 2002).

Equations of motion for the SPH particles can be derived from a Lagrangian (e.g. Monaghan & Price 2001; Price 2012)

$$L = \sum_b m_b \left[\frac{1}{2}\mathbf{v}_b^2 - u_b(\rho_b, s_b)\right], \tag{17}$$

giving a parallel to its continuous counterpart (Seliger & Whitham 1968)

$$\mathcal{L} = \iint \rho_0 \left[\frac{1}{2}\left(\frac{\partial \mathbf{x}_i}{\partial t}\right)^2 - u\right] \mathrm{d}\boldsymbol{\alpha}\mathrm{d}t. \tag{18}$$

From a variational principle, one obtains

$$\frac{\mathrm{d}\mathbf{x}_a}{\mathrm{d}t} = \mathbf{v}_a, \tag{19}$$

$$\frac{\mathrm{d}\mathbf{v}_a}{\mathrm{d}t} = \sum_b m_b \left(\frac{P_a}{\rho_a^2 \Omega_a}\nabla_a W_{ab}(h_a) + \frac{P_b}{\rho_b^2 \Omega_b}\nabla_a W_{ab}(h_b)\right). \tag{20}$$

The variational method guarantees that Eqs. 19 – 20 preserve total linear momentum, angular momentum and energy conservation up to machine precision.

Hydrodynamic shocks are not captured by Eq. 20. To address this issue, a Von Neumann shock viscosity with linear and quadratic terms (Von Neumann & Richtmyer 1950; Landshoff 1955; Margolin & Lloyd-Ronning 2022) is employed for velocities to circumvent this limitation, in the SPH solver we use the shock capturing terms from Price & Federrath (2010); Lodato & Price (2010) modified from the original formation of Monaghan (1997a). The extended equation of motion becomes

$$\frac{\mathrm{d}\mathbf{v}_a}{\mathrm{d}t} = \sum_b m_b \left(\frac{P_a + q^a_{ab}}{\rho_a^2 \Omega_a}\nabla_a W_{ab}(h_a) + \frac{P_b + q^b_{ab}}{\rho_b^2 \Omega_b}\nabla_a W_{ab}(h_b)\right), \tag{21}$$

where

$$q^a_{ab} = \begin{cases} -\frac{1}{2}\rho_a v_{{\rm sig},a} \boldsymbol{v}_{ab} \cdot \hat{\boldsymbol{r}}_{ab}, & \boldsymbol{v}_{ab} \cdot \hat{\boldsymbol{r}}_{ab} < 0 \\ 0 & \text{otherwise,} \end{cases}, \tag{22}$$

$$v_{{\rm sig},a} = \alpha_a^{\rm AV} c_{s,a} + \beta |\mathbf{v}_{ab} \cdot \hat{\mathbf{r}}_{ab}|\ ,\ \alpha_a^{\rm AV} \in [0, 1]. \tag{23}$$

To properly capture energy discontinuities, a shock conductivity (also known as artificial conductivity) is employed for the internal energy (e.g. Noh 1987; Margolin & Lloyd-Ronning 2022). Eq. 15 extends to (e.g. Chow & Monaghan 1997; Price 2012, 2008)

$$\frac{\mathrm{d}u_a}{\mathrm{d}t} = \frac{P_a + q^a_{ab}}{\rho_a^2 \Omega_a}\sum_b m_b \mathbf{v}_{ab} \cdot \nabla_a W_{ab}(h_a) + \Lambda_{\rm cond}, \tag{24}$$

where

$$\Lambda_{\rm cond} = \sum_b m_b \beta_u v^u_{\rm sig}(u_a - u_b)\frac{1}{2}\left[\frac{F_{ab}(h_a)}{\Omega_a \rho_a} + \frac{F_{ab}(h_b)}{\Omega_b \rho_b}\right], \tag{25}$$

$$v^u_{\rm sig} = \sqrt{\frac{|P_a - P_b|}{(\rho_a + \rho_b)/2}}, \tag{26}$$

using $F_{ab}(h_a) = \hat{\mathbf{r}}_{ab} \cdot \nabla_a W_{ab}(h_a)$. We use the symbol β_u to represent the shock conductivity parameter instead of the conventional α_u. This change clarifies that α_u is related to the quadratic part of the artificial viscosity $\beta|\mathbf{v}_{ab} \cdot \hat{\mathbf{r}}_{ab}|$, rather than the linear part $\alpha_a^{\rm AV} c_{{\rm s},a}$ (Von Neumann & Richtmyer 1950; Noh 1987; Margolin & Lloyd-Ronning 2022). Writing Eq. 21&24 with a shock viscosity expressed as a modified pressure ensures consistent application of the corresponding terms in both the velocity and energy equations.

5.2 Shock detection

To provide shock detection to enable shock viscosity only in regions of interest we use the method from Cullen & Dehnen (2010) that was implement in Phantom (Price et al. 2018), which is an improved version of the Morris & Monaghan (1997) switch (see Price 2008, 2012). The value of the shock viscosity parameter α_a is evolved using

$$\frac{\mathrm{d}\alpha_a}{\mathrm{d}t} = -\frac{(\alpha_a - \alpha_{{\rm loc},a})}{\tau_a}. \tag{27}$$

The targeted value of the shock viscosity parameter $\alpha_{{\rm loc},a}$ is defined using

$$\alpha_{{\rm loc},a} \equiv \min\left(10 A_a \frac{h_a^2}{c^2_{{\rm s},a}}, \alpha_{\rm max}\right), \tag{28}$$

where

$$A_a \equiv \xi_a \max\left[-\frac{\mathrm{d}}{\mathrm{d}t}\left(\nabla\cdot\boldsymbol{v}_a\right), 0\right], \tag{29}$$

is the shock indicator and ξ_a is the corrective factor (Balsara 1995)

$$\xi \equiv \frac{|\nabla\cdot\boldsymbol{v}|^2}{|\nabla\cdot\boldsymbol{v}|^2 + |\nabla\times\boldsymbol{v}|^2}. \tag{30}$$

The rising time $\tau_a \equiv h_a/(c_{s,a}\sigma_d)$ is parameterised by the decay parameter $\sigma_d = 0.1$, a typical value for practical cases. In practice, $\alpha_a(t)$ is set directly to $\alpha_{\mathrm{loc},a}$ if $\alpha_{\mathrm{loc},a} > \alpha_a(t)$. Similarly to the approach used in Phantom (Cullen & Dehnen 2010), we use SPH derivatives that are exact to the linear order are used to compute

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\nabla\cdot\boldsymbol{v}_a\right) = \sum_i \frac{\partial a_a^i}{\partial x_a^i} - \sum_{i,j}\frac{\partial v_a^i}{\partial x_a^j}\frac{\partial v_a^j}{\partial x_a^i}, \tag{31}$$

where for a given field ϕ, this accurate SPH derivative is

$$R_a^{ij}\frac{\partial \phi_a^k}{\partial x_a^j} = \sum_b m_b\left(\phi_b^k - \phi_a^k\right)\frac{\partial W_{ab}\left(h_a\right)}{\partial x^i}, \tag{32}$$

where,

$$R_a^{ij} = \sum_b m_b\left(x_b^i - x_a^i\right)\frac{\partial W_{ab}\left(h_a\right)}{\partial x^j} \approx \delta^{ij}. \tag{33}$$

Inverting R_a^{ij} and applying it to Eq. 32 provides the desired derivative.

5.3 SPH interaction criterion

Eq. 21 shows that two SPH particles interact when their relative distance is inferior to the maximum of their interaction radius. Formally, the object-object interaction criterion between two particles a and b is

$$\gamma_{o/o}(a,b) \equiv \left\{|\mathbf{r}_a - \mathbf{r}_b| < \max(l_a, l_b)\right\}. \tag{34}$$

Consider now a group of SPH particles, and let us embed them in an axis-aligned bounding box (AABB). Consider another SPH particle. A necessary condition for the latter particle to interact with the AABB is: it resides within the volume formed by extending the AABB in all directions by the maximum of all interaction radii of particles inside the AABB, or, a ball centered on the particle, with a radius equal to its interaction radius, intersects the AABB. Formally, the interaction criterion between the particle and the AABB of particles is therefore

$$\gamma^1_{o/g}(a,\{b\}_{b\in AABB}) \equiv \left(r_a \in AABB \oplus l_{AABB,b}\right) \vee \Big(B(\mathbf{r}_a, l_a)\cap AABB \neq \emptyset\Big), \tag{35}$$

where $B(\mathbf{r}_a, l_a)$ is a ball centred on $\mathbf{r}_a$ and of diameter $2l_a$, $l_{AABB,b}$ is the maximum interaction radius of the particles in the AABB, $\max_{b\in AABB}(l_b)$, $AABB\oplus l$ is the operation that extends the AABB in every direction by a distance l and $\vee$ is the boolean `or` operator. Consider now the ball centred on $\mathbf{r}_a$ with a diameter of $2l_a$. We denote $AABB(\mathbf{r}_a, l_a)$ the square AABB with a side length of $2l_a$, centred at $\mathbf{r}_a$, ensuring that it encompasses the ball. Replacing the original ball by this AABB in Eq. 35 yields the following group-object criterion

$$\gamma^2_{o/g}(a,\{b\}_{b\in AABB}) \equiv \left(r_a \in AABB \oplus l_{AABB,b}\right) \vee \Big(AABB(\mathbf{r}_a, l_a)\cap AABB \neq \emptyset\Big). \tag{36}$$

Though less stringent than that of Eq. 35, this criterion is easier to handle in practice. Indeed, one can show that (App. B)

$$AABB_1 \oplus h \cap AABB_2 \neq \emptyset \Leftrightarrow AABB_1 \cap AABB_2 \oplus h \neq \emptyset. \tag{37}$$

Let $\mathrm{AABB}_{1\mathrm{e}}$ and $\mathrm{AABB}_{2\mathrm{e}}$ denote the extended version of AABB_1 and AABB_2, extended by the distance h in all three directions respectively. Eq. 37 asserts that if $\mathrm{AABB}_{1\mathrm{e}}$ intersects AABB_1, it is equivalent for AABB_1 to intersect $\mathrm{AABB}_{2\mathrm{e}}$. Applied on Eq. 36, Eq. 37 guarantees that the object-group interaction criterion can be rewritten by moving the contribution of the interaction radius of the particle a to the term corresponding to the AABB in the second brackets, as follows

$$\gamma^2_{o/g}(a,\{b\}_{b\in AABB}) \equiv \left(r_a \in AABB \oplus l_{AABB,b}\right) \vee \Big(AABB(\mathbf{r}_a,0)\cap AABB\oplus l_a \neq \emptyset\Big), \tag{38}$$

$$\equiv \left(r_a \in AABB \oplus l_{AABB,b}\right) \vee \left(r_a \in AABB \oplus l_a\right) \tag{39}$$

$$\equiv \left[r_a \in AABB \oplus \max\left(l_{AABB,b}, l_a\right)\right]. \tag{40}$$

The three criteria discussed above satisfy the hierarchy

$$\gamma^2_{o/g}(a,\{b\}_{b\in AABB}) \Leftarrow \gamma^1_{o/g}(a,\{b\}_{b\in AABB}) \Leftarrow \bigvee_b \gamma_{o/o}(a,b). \tag{41}$$

Finally, one can extend the first form of $\gamma^2_{o/g}$ to the following group-group interaction criterion

$$\gamma_{g/g}(AABB_1, AABB_2) \equiv \Big([AABB_1 \oplus l_{AABB_1,a}]\cap AABB_2 \neq \emptyset\Big) \vee \Big(AABB_1 \cap [AABB_2 \oplus l_{AABB_2,b}] \neq \emptyset\Big). \tag{42}$$

Using Eq. 37 similarly as for Eq.36 we obtain the form of the group-group interaction criterion used in Shamrock,

$$\gamma_{g/g}(AABB_1, AABB_2) \equiv \Big(AABB_1 \cap [AABB_2 \oplus \max(l_{AABB_1,a}, l_{AABB_2,a})] \neq \emptyset\Big). \tag{43}$$

In summary, the interaction criteria used for SPH in Shamrock are:

- Object-object criterion :

$$\gamma_{o/o}(a,b) = |\mathbf{r}_a - \mathbf{r}_b| < R_{\mathrm{kern}}\max(h_a, h_b)$$

- Object-group criterion :

$$\gamma^2_{o/g}(a,\{b\}_{b\in AABB}) = \left[r_a \in AABB \oplus R_{\mathrm{kern}}\max\left(h_{AABB,b}, h_a\right)\right]$$

- Group-group criterion :

$$\gamma_{g/g}(AABB_1, AABB_2) = \Big(AABB_1 \cap [AABB_2 \oplus R_{\mathrm{kern}}\max(h_{AABB_1,a}, h_{AABB_2,a})] \neq \emptyset\Big)$$

5.4 Adaptive smoothing length

In astrophysics, a typical choice consists in choosing h_a in a way that the resolution follows the density

$$\rho(h) = m\left(\frac{h_{\mathrm{fact}}}{h}\right)^3, \tag{44}$$

where h_{fact} is a tabulated dimensionless constant that depends on the kernel (e.g. $h_{\mathrm{fact}} = 1.2$ for the M_4 cubic kernel). This specific

form also implies that the averaged number of neighbours within the compact support of a given SPH particle is roughly constant throughout the simulation. Eq. 44 must itself be consistent with the definition of density Eq. 12, since h depends on ρ and vice versa. Achieving this requires for density and smoothing length to be calculated simultaneously, by minimising the function

$$\delta\rho = \rho_a - \rho(h_a). \tag{45}$$

This approach allows an accurate use of $\rho(h_a)$ in the algorithms rather than calculating the SPH sum. In practice, the iterative procedure is conducted with a Newton-Raphson algorithm. The steps outlined in Alg. 7 describe the iterative procedure used to update the smoothing length. A technicality related to ghost zones arises

Algorithm 7: Smoothing length update

Data: h_a^n The smoothing lengths at timestep n, χ The ghost zone size tolerance.
Result: h_a^{n+1} The smoothing lengths at timestep $n+1$.

$\{\epsilon_a \leftarrow -1\}_a$;
// Use a copy of h_a^n to do iterations
$\{h_a \leftarrow h_a^n\}_a$;
// Outer loop for ghost exchange
while $\min_a(\epsilon_a) = -1$ **do**
  ... exchange ghosts positions with tolerance χ ...;
  // Inner loop for Newton-Rahpson
  while $\max_a(\epsilon_a) > \epsilon_c$ **do**
    for a ***in parallel*** **do**
      // Compute the SPH sum
      $\rho_a \leftarrow \sum_b m_b W_{ab}(h_a)$;
      // Newton-Rahpson
      $\delta\rho \leftarrow \rho_a - \rho(h_a)$;
      $d\delta\rho \leftarrow \sum_b m_b \frac{\partial W_{ab}(h_a)}{\partial h_a} + \frac{3\rho_a}{h_a}$;
      $h_a^{n+1} \leftarrow h_a - \delta\rho/d\delta\rho$;
      // Avoid over/under-shooting
      if $h_a^{n+1} > h_a\lambda$ **then**
        $h_a^{n+1} \leftarrow h_a\lambda$;
      else if $h_a^{n+1} < h_a/\lambda$ **then**
        $h_a^{n+1} \leftarrow h_a/\lambda$;
      $\epsilon_a \leftarrow |h_a^{n+1} - h_a|/h_a^n$;
      // Exceed ghost size
      if $h_a^{n+1} > h_a^n\chi$ **then**
        $h_a^{n+1} \leftarrow h_a^n\chi$;
        $\epsilon_a \leftarrow -1$;

during this procedure. The size γ_{12} of the ghost zone separating two adjacent patches, P_1 and P_2, is determined by the group-group interaction criterion between these patches

$$\gamma_{12} = \max\left(\max_{\{a\}} h_a, \max_{\{b\}} h_b\right). \tag{46}$$

where a and b stem for indices of particles in P_1 and P_2 respectively. In Shamrock, the size γ_{12} is increased by a safety factor χ, termed as the *ghost zone size tolerance*. This factor acknowledges that ghost zone structures should withstand fluctuations in smoothing lengths throughout the iterative process. With this tolerance, the smoothing length can fluctuate by a factor of χ during density iterations without necessitating Shamrock to regenerate the ghost zones. In practice, we first exchange the ghost zones using a tolerance $\chi = 1.1$, then iterate until all particles converge to the consistent smoothing length or exceed the ghost zone size tolerance. If the latter occurs, we restart the process from the beginning with the updated smoothing length. We find that this almost rarely arises, except during the initial time step when the smoothing length is converged for the first time. Alg. 7 shows that in Shamrock, we use an additional safety factor, denoted as λ, to prevent over- and undershooting throughout the iterations. Without this correction, the iterative procedure may yield unstable negative smoothing lengths. In practice, we use $\lambda = 1.2$.

5.5 Time stepping

5.5.1 Leapfrog integration

By construction, standard SPH is conservative and achieves second-order accuracy in space in smooth flows. To ensure consistency, time integration in Shamrock employs a symplectic second-order leapfrog integrator, or 'Kick-drift-kick' (e.g. Verlet 1967; Hairer et al. 2003):

$$\boldsymbol{v}^{n+\frac{1}{2}} = \boldsymbol{v}^n + \frac{1}{2}\Delta t \boldsymbol{a}^n, \tag{47}$$

$$\boldsymbol{r}^{n+1} = \boldsymbol{r}^n + \Delta t \boldsymbol{v}^{n+\frac{1}{2}}, \tag{48}$$

$$\boldsymbol{v}^* = \boldsymbol{v}^{n+\frac{1}{2}} + \frac{1}{2}\Delta t \boldsymbol{a}^n, \tag{49}$$

$$\boldsymbol{a}^{n+1} = \boldsymbol{a}\left(\boldsymbol{r}^{n+1}, \boldsymbol{v}^*\right), \tag{50}$$

$$\boldsymbol{v}^{n+1} = \boldsymbol{v}^* + \frac{1}{2}\Delta t\left[\boldsymbol{a}^{n+1} - \boldsymbol{a}^n\right], \tag{51}$$

where $\mathbf{r}^n$, $\mathbf{v}^n$ and $\mathbf{a}^n$ denote positions, velocities and acceleration at the $n^{-\text{th}}$ time step Δt. In the scheme presented in Price et al. 2018, a combined iteration is used to calculate the acceleration $\boldsymbol{a}^{n+1}$ and update the smoothing length at the same time. To minimise the amount of data communicated, we separate the acceleration and the smoothing length update. In Shamrock, the smoothing length is calculated after applying Eq. 49. Only positions are required for the smoothing length iteration. Once these iterations are complete, we first calculate Ω_a using Eq. 16, then exchange the ghost zones with the required fields, including Ω_a subsequently used in derivative computations. Similar to the approach used in Phantom, we use the correction applied to the velocity, calculated during the correction step of the leapfrog, as a reference to check that the resulting solution is reversible over time. The correction applied at the end of the leapfrog scheme is as follows

$$\Delta\mathbf{v}_i = \frac{1}{2}\Delta t\left[\boldsymbol{a}_i^{n+1} - \boldsymbol{a}_i^n\right]. \tag{52}$$

We use the result of Eq. 52 to verify that the maximum correction does not exceed a fraction ϵ_v of the mean square correction

$$\max_i\left(|\Delta\mathbf{v}_i|/\sqrt{\frac{1}{N}\sum_j |\Delta\mathbf{v}_j|^2|}\right) < \varepsilon_\mathrm{v}. \tag{53}$$

In practice, we set the value $\varepsilon_\mathrm{v} = 10^{-2}$. If any particles fail to meet this criterion, we recalculate the acceleration and apply the correction step again with $\boldsymbol{v}^* \leftarrow \boldsymbol{v}^{n+1}$ instead.

5.5.2 Choice of the timestep

The value of the explicit time step is governed by the Courant-Friedrich-Levy stability condition (Courant et al. 1928). Following

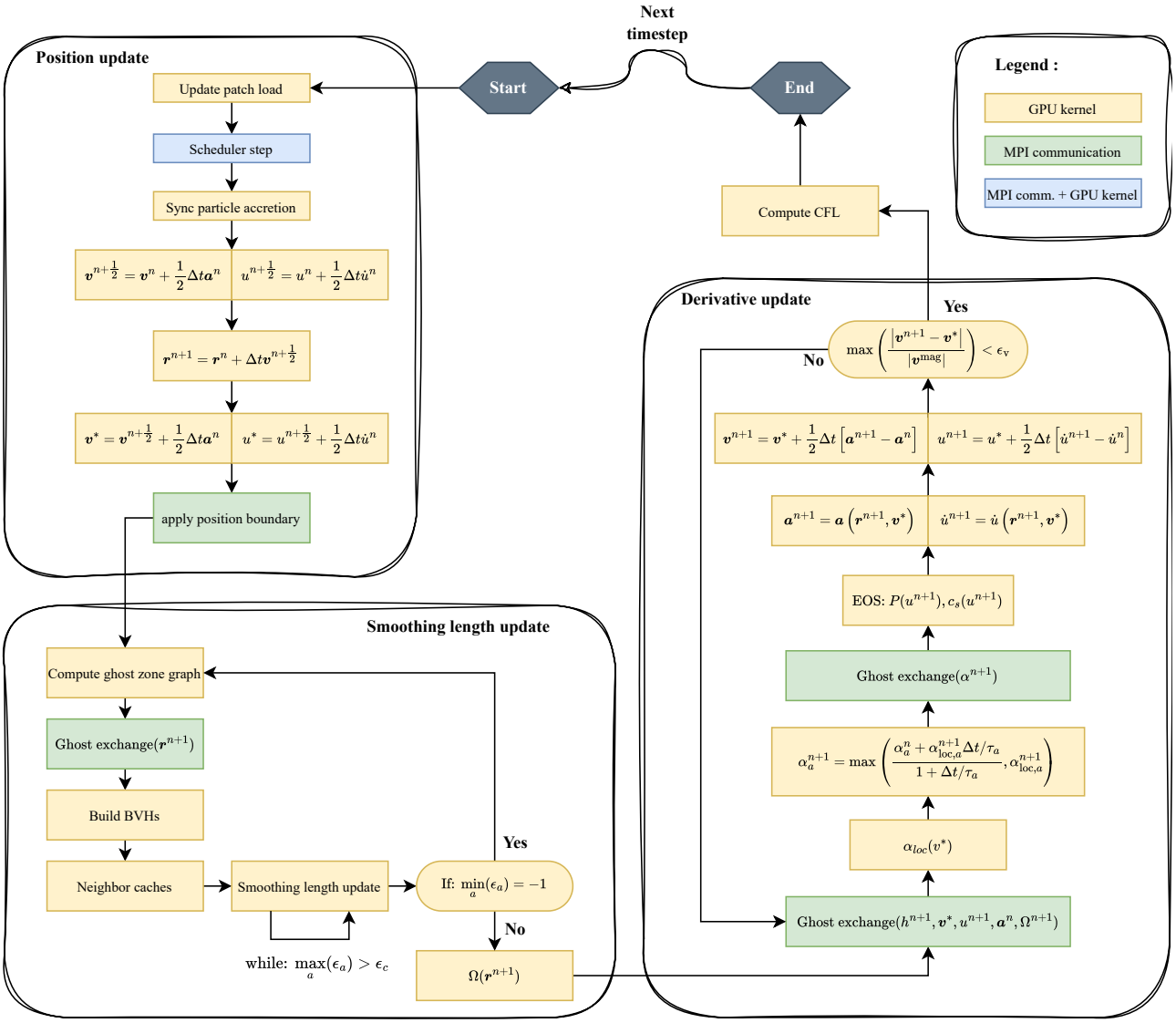

Figure 12. Illustration of an SPH time step through an organisational diagram representing one time step of the SPH scheme, the process being divided into three main sub-steps. Firstly, position updates (scheduling step for patch decomposition, leapfrog prediction, and application of position boundaries if necessary). Secondly, smoothing length updates (generation of ghost zone graph, construction of BVHs, creation of neighbour caches, smoothing length adjustment, computation of Ω). Thirdly, derivative updates (field exchange, viscosity and derivative updates, application of leapfrog corrector). The step concludes with updating the CFL condition. The corresponding equations showed on this flowchart corresponds for the position and derivative update to the equations showed Sect. 5.5.1 and 5.5.4. For the smoothing length the corresponding equations are detailed Sect. 5.4.

Price et al. (2018) from Lattanzio et al. (1986); Monaghan (1997b),

$$\Delta t \equiv \min(C_{\text{cour}} \frac{h_a}{v_{\text{sig},a}^{\text{dt}}}, C_{\text{force}} \sqrt{\frac{h_a}{|a_a|}}). \tag{54}$$

The first term allows to correctly capture the propagation of the hydrodynamic characteristic waves in the fluid at a given resolution. Similarly, the second term ensures correct treatment of the action of external forces on the fluid. The safety coefficients are set to the following values $C_{\text{cour}} = 0.3$ and $C_{\text{force}} = 0.25$.

5.5.3 *CFL multiplier*

To minimize the cost associated with executing the correction cycles of the leapfrog scheme, we reduce the time step for a few iterations when Eq. 53 is not satisfied, similar to the approach taken in Phantom. To do this in Shamrock, we introduce a so-called *CFL multiplier* λ_{CFL}, which consists of an additional variable factor applied to the CFL condition. Therefore, the effective C_{cour} and C_{force} used in Shamrock SPH solver are

$$C_{\text{cour}} = \lambda_{\text{CFL}} \tilde{C}_{\text{cour}}, \quad C_{\text{force}} = \lambda_{\text{CFL}} \tilde{C}_{\text{force}}, \tag{55}$$

where $\tilde{C}_{\text{cour}}$ and $\tilde{C}_{\text{force}}$ are the safety coefficients chosen by default by the user. If Eq. 53 is not satisfied, we divide λ_{CFL} by a factor of 2. Otherwise, at each time step,

$$\lambda_{\text{CFL}}^{n+1} = \frac{1 + \lambda_{\text{stiff}} \lambda_{\text{CFL}}^n}{1 + \lambda_{\text{stiff}}}, \tag{56}$$

where λ_{stiff} is a coefficient that parameters the stiffness of the evolution of the CFL multiplier. This numerical strategy allows to handle shocks in the simulation, automatically cycling leapfrog iterations over the CFL condition, thereby reducing the time step to enhance

energy conservation. This procedure is particularly effective during the first time steps of the Sedov-Taylor blast problem.

5.5.4 Shock detection

The shock viscosity parameter α is evolved according to Eq.27. After the leafprog prediction step, an implicit time step is used for this integration

$$\alpha^{n+1}_{\mathrm{loc},a} = \alpha_{\mathrm{loc},a}(\boldsymbol{v}^*, \nabla \boldsymbol{v}^*, \nabla \boldsymbol{a}^n), \tag{57}$$

$$\alpha^{n+1}_{a} = \max\left(\frac{\alpha^n_a + \alpha^{n+1}_{\mathrm{loc},a}\Delta t/\tau_a}{1+\Delta t/\tau_a}, \alpha^{n+1}_{\mathrm{loc},a}\right). \tag{58}$$

5.5.5 Summary

We have implemented in Shamrock an SPH hydrodynamical solver with self-consistent smoothing length that handles shock though the combined used of shock viscosity and conductivity with state-of-the-art shock detector. Fig. 12 shows a comprehensive overview of one SPH time step in Shamrock.

6 PHYSICAL TESTS

6.1 Generalities

Firstly, we validate the SPH solver by performing convergence tests against classical problems having analytical solutions, such as the Sod tube and the Sedov-Taylor blast test. The hydrodynamic tests presented in this Section are performed using the M_6 kernel with $h_{fact} = 1.0$, with an average number of neighbours of 113 neighbours for each SPH particle (almost no difference is expected in the results when using other spline kernels, e.g. Price et al. 2018). In all tests, momentum is conserved to machine precision. The choice of the CFL condition result in energy deviations that do not exceed 10^{-6} relative error with respect to the initial value.

Secondly, we examine the residuals obtained from comparing the results generated by Shamrock and Phantom. Since both codes use the same SPH algorithm, such an analysis is required for conducting further performance comparisons. L2 errors are estimated using the norm

$$\mathcal{L}_2(A_{\mathrm{sim}}, A_{\mathrm{ref}}) = \sqrt{\frac{1}{N_{\mathrm{part}}}\sum_i |A_{i,\mathrm{sim}} - A_{i,\mathrm{ref}}|^2}, \tag{59}$$

where $A_{i,\mathrm{ref}}$ represents the reference quantities, while $A_{i,\mathrm{sim}}$ denotes the quantities computed in the simulation.

6.2 Advection

We first perform an advection test in a periodic box of length unity to verify the correct treatment of the periodic boundaries by Shamrock. Three lattices of $(16 \times 12 \times 12)$, $(64 \times 24 \times 24)$ and $(16 \times 12 \times 12)$ particles having velocities $v_x = 1.0$ are initially juxtaposed, such that $\rho = 1.0$ if $0.25 \leq x \leq 0.75$, and $\rho = 0.1$ elsewhere. We let the simulation evolve until the step has crossed several times the boundaries of the box (we choose $t = 11$, although any other time, even very large, yields the same outcome since SPH is Galilean invariant). The result obtained at the end of the simulation is identical to the initial setup to machine precision.

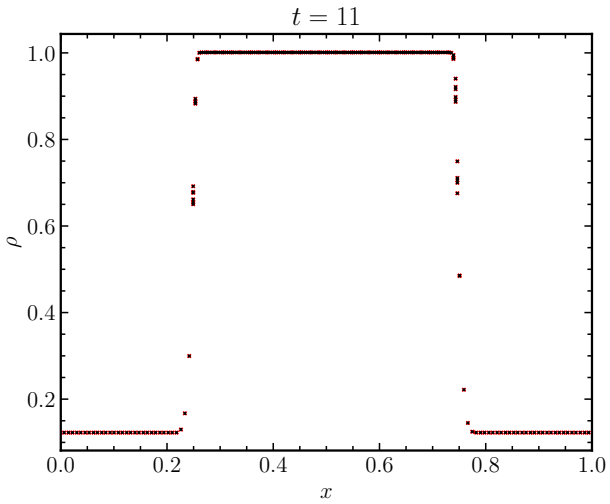

Figure 13. Advection of a density step across several traversal of a periodic box, in code units. SPH being Galilean invariant, the results (black dots) precisely match the initial setup (red crosses) down to machine precision, thus validating the boundary treatment in Shamrock.

6.3 Sod tube

We perform a Sod-tube test (Sod 1978) by setting up a box with a discontinuity between a left state and a right state initially positioned at $x = 0.5$. In the left state $x < 0.5$, the density and the pressure are set to $\rho_{\mathrm{l}} = 1$ and $P_{\mathrm{l}} = 1$, while in the right state $x > 0.5$, they are set to $\rho_{\mathrm{r}} = 0.125$ and $P_{\mathrm{r}} = 0.1$ respectively. To initialise the density profile, we use a periodic box in which we setup a $24 \times 24 \times 256$ hexagonal close packed lattice in $x \in [-0.5, 0.5]$ and $12\times12\times128$ in $x \in [0.5, 1.5]$. The initial velocity is uniformly set to zero throughout the simulation. The size of the simulation box size is adjusted such that we ensure periodicity across the y and z boundaries. We use $\gamma = 1.4$ to align our test with the Sod tube test commonly performed in grid codes. No particle relaxation step is used in this test, since the initial distribution of SPH particles closely resembles a relaxed distribution akin to a crystal lattice. We use periodic boundaries in the x direction. Shock viscosity is setup with the default parameters of Shamrock, namely $\sigma_d = 0.1$, $\beta = 2$, $\beta_u = 1$. The setup presented above is then evolved until $t = 0.245$. Fig. 14 shows results obtained for velocity, density, and pressure, displaying additionally the shock-capturing parameter α. For $N_x = 128$ particles $\mathcal{L}_2$ errors are $\sim 10^{-3}$ in v and $\sim 10^{-4}$ in ρ and P, similarly to what is obtained with other SPH codes. Similar setups are used to perform convergence analysis, except for the lattice, for which we use $24 \times 24 \times N_x$, and $12\times12\times(N_x/2)$ particles instead respectively. Results obtained when varying the value of N_x are reported in Fig. 15. We observe second-order convergence on the pressure, first-order convergence in density, and in-between convergence in velocity. The scattering observed in the velocity field behind the shock corresponds to particle having to reorganise the crystal lattice, a typical feature of SPH (e.g. Price et al. 2018). Letting the shock evolve further and interact with the periodic boundary, we verify that we obtain a second symmetric shock, as expected.

6.4 Sedov-Taylor blast

We perform a Sedov-Taylor blast wave test (Sedov 1959; Taylor 1950a,b) by first setting up a medium of uniform density $\rho = 1$ with $u = 0$ and $\gamma = 5/3$, in a 3D box of dimensions $[-0.6, 0.6]^3$.

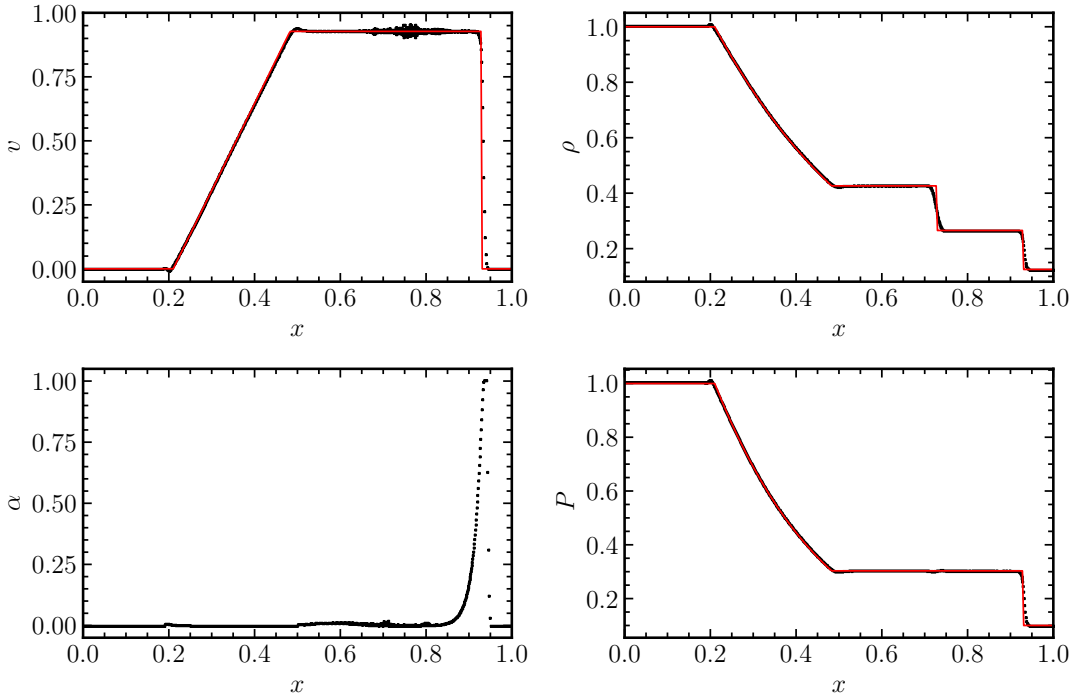

Figure 14. Result obtained for the Sod-tube test by juxtaposing two tubes of $24 \times 24 \times 512$ particles in $x \in [-0.5, 0.5]$ and $12 \times 12 \times 256$ particles in $x \in [0.5, 1.5]$ organised in hexagonal compact packing lattices. The density is set to $\rho = 1$ in $x \in [-0.5, 0.5]$ and $\rho = 0.125$ in $x \in [0.5, 1.5]$. Initial pressure is $P = 1$ for $x \in [-0.5, 0.5]$ and $P = 0.1$ for $x \in [0.5, 1.5]$, with zero initial velocities. An adiabatic equation of states with $\gamma = 1.4$ is used. Boundaries are periodic, and only half of the simulation is displayed. The simulation is performed until $t = 0.245$, and numerical results are compared against the analytic solution. We additionally show the values of the shock viscosity parameter α.

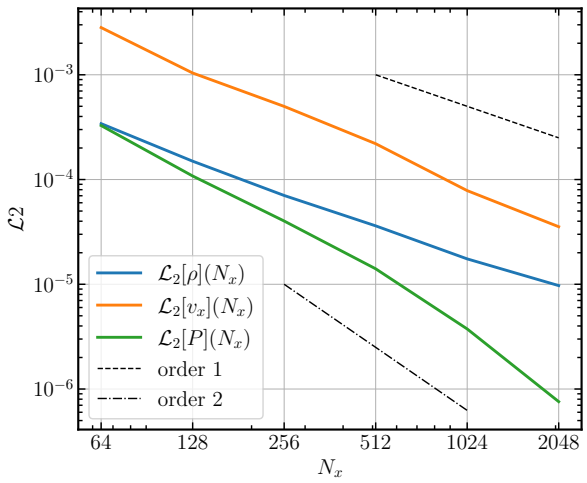

Figure 15. $\mathcal{L}_2$ errors obtained for the Sod shock tube test presented on Fig.14 as a function of the number N_x of particles used on the x axis for $x \in [-0.5, 0.5]$. We observe second-order convergence on the pressure, first-order convergence in density, and in-between convergence in velocity, as found in other SPH codes.

The particles are arranged locally on a compact hexagonal lattice. The smoothing length is initially converged by iterating a white time step. Internal energy is then injected in the centre of the box. This energy peak is smoothed by the SPH kernel according to $u_a = W(\mathbf{r}, 2h_0) \times E_0$, where the total amount of energy injected is fixed at $E_0 = 1$ and h_0 is the smoothing length of the particles after relaxation. For this test, the CFL condition is lowered to $\tilde{C}_{\text{cour}} = \tilde{C}_{\text{force}} = 0.1$ to prevent leapfrog corrector sub-cycling caused by the strong shock. This result in an enhanced energy conservation, with a maximum relative error of 10^{-6} observed across all tests. The simulation is then evolved up to $t = 0.1$. Simulations with $N = 26, 52, 105, 210$ have been performed on a single A100 GPU of an NVidia DGX workstation. Simulations with $N = 420$ and $N = 840$ were executed on the Adastra supercomputer (see Sect. 7) on 4 and 32 nodes respectively. The highest resolution blast test involves 1.255 Gpart, including ghost particles. The simulation consists in 14979 iterations performed in 14 hours, including setup and dumps, on 32 nodes corresponding to 128 Mi250X or equivalently 256 GCDs (see Sect. 7.1.1 for details). The total energy consumed for this test is 1.94 GJ, as reported by Slurm. The power consumption per node is 1195 W, which equates to slightly over half of the peak consumption of a single node (2240 W). Numerical results are compared against analytical solutions. Fig.16 shows results obtained for the density for N^3 particles, with $N = 105, 210, 420, 840$. For the latter case, $\mathcal{L}_2$ errors are of order $\sim 10^{-1}$, which is similar to what is obtained with other SPH codes with this particular setup. Figure 17 shows that order one convergence is achieved for v, ρ and P, similarly to what is obtained with other SPH codes.

6.5 Kelvin-Helmholtz instability

We test the ability of Shamrock to capture instabilities related to discontinuities on internal energy by performing a Kelvin-Helmholtz instability test (Price 2008). We adopt a setup close to the one proposed by Schaal et al. (2015), that gives rise to secondary instabilities fostering additional turbulence mixing. The initial pressure, density

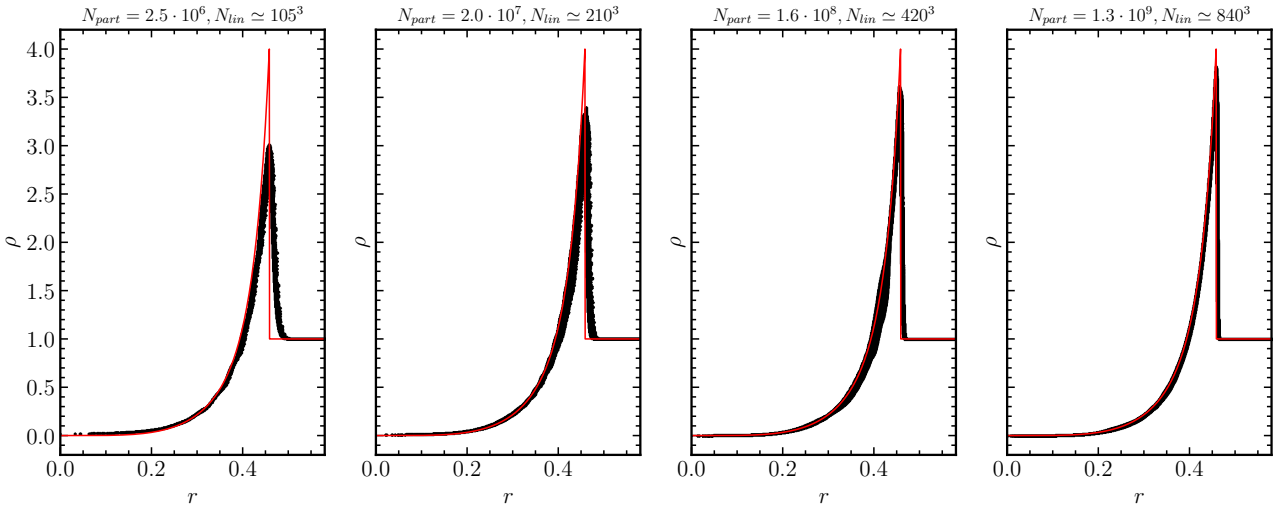

Figure 16. Result of the densities (black dots) obtained for the Sedov-Taylor blast test described in Sect. 6.4 at $t = 0.1$ for $N_{\rm part}$ particles, with $N_{\rm part} = 2.5 \cdot 10^6, 2.0 \cdot 10^7, 1.6 \cdot 10^8, 1.3 \cdot 10^9$ SPH particles (global time-stepping), corresponding to an inter particle spacing on the HCP lattice of respectively $10^{-2}/4$, $10^{-2}/8$, $10^{-2}/16$, $10^{-2}/32$. Results are represented against the analytical solution (solid red line). The legend provides the effective linear resolution $N_{\rm lin}$ corresponding the cubic root of the number of particle displayed on each graphs which are truncated at $r = 0.58$.

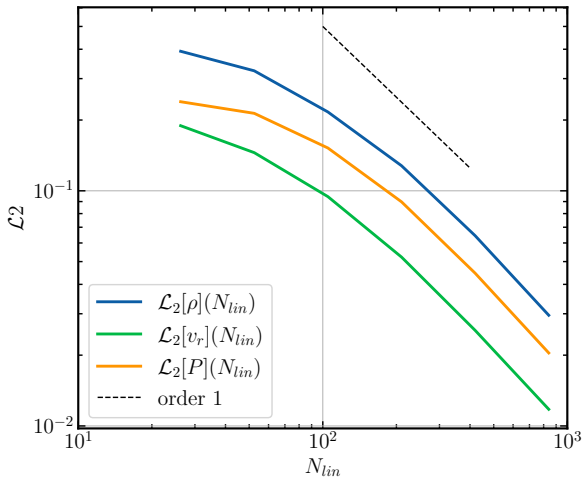

Figure 17. Convergence study of the Sedov-Taylor blast test presented in Fig. 16. Order one convergence is achieved for v, ρ and P, similarly to what is obtained with other SPH codes.

and velocity profiles are initialised according to

$$P = 3.5 \tag{60}$$

$$\rho = \begin{cases} 1, & \text{if } |y| > y_s/4, \\ (3/2)^3, & \text{if } |y| \leq y_s/4, \end{cases} \tag{61}$$

$$v_x = \begin{cases} \xi/2, & \text{if } |y| > y_s/4, \\ -\xi/2, & \text{if } |y| \leq y_s/4, \end{cases} \tag{62}$$

$$v_y = \varepsilon \sin(4\pi x) \left\{ \exp\left(-\frac{(y - y_s/4)^2}{2\sigma^2} \right) + \exp\left(-\frac{(y + y_s/4)^2}{2\sigma^2} \right) \right\}. \tag{63}$$

The test is performed in 2.5D, restricting the z axis to a thin layer comprising only 6 SPH particles in the low density region, and 9 particles in the high density region. We opt for a density ratio of $(3/2)^3$ between the two regions to simplify the particle setup process and circumvent unnecessary complexities associated with arranging particles on closed-packed lattices. We use $\gamma = 1.4$. The slip velocity and the perturbation parameters are $\xi = 1$, $\epsilon = 10^{-2}$, $\sigma = 0.05/\sqrt{2}$, similarly to the values used in Schaal et al. (2015). Simulations are performed on a single A100-40GB GPU. This GPU can accommodate a maximum of approximately $\sim 40 \cdot 10^6$ particles for the M_4 kernel and $\sim 20 \cdot 10^6$ particles for the M_6 kernel. Fig. 18 shows results obtained at increasing resolutions for the M_4 kernel (top panel) and the M_6 kernel (bottom panel). Similarly to the findings of Tricco (2019), we first observe that the M_4 fails to accurately capture the instability, even at high resolutions, as vortices appear flattened and overly diffused. Conversely, we observe that all these features are effectively captured when employing the M_6 kernel. The further Sect. 6.6 shows that our results align almost perfectly with those obtained with Phantom. The growth rate observed for the instability matches therefore the findings reported in Tricco (2019).

6.6 Conformance with Phantom

We aim to benchmark the performance of Shamrock against a state-of-the-art, robust, optimised and extensively tested SPH code running on CPUs. Several SPH codes are in use in the community (e.g. Bonsai-SPH Bedorf & Portegies Zwart 2020,Gadget Springel et al. 2021, Gasoline Wadsley et al. 2004, Gizmo Hopkins 2014, Seren Hubber et al. 2011, Swift Schaller et al. 2018). We choose Phantom, since it is optimised for hydrodynamics, well-used by the astrophysical community and extensively tested and documented (Price et al. 2018). Before conducting comparisons, one has to ensure that the two solvers are rigorously identical, up to identified insignificant errors. This is the purpose of the next two tests, that are uncommonly designed to reveal discrepancies by amplifying errors using lower resolution or less regular kernels than achievable. For this, we generate the initial conditions with Phantom, then start an identical simulation from the same dump using a fixed time step.

6.6.1 *Residuals: Low res Sedov-Taylor blast wave test*

We first measure the residual discrepancies between Shamrock and Phantom by comparing results obtained on two identical Sedov-

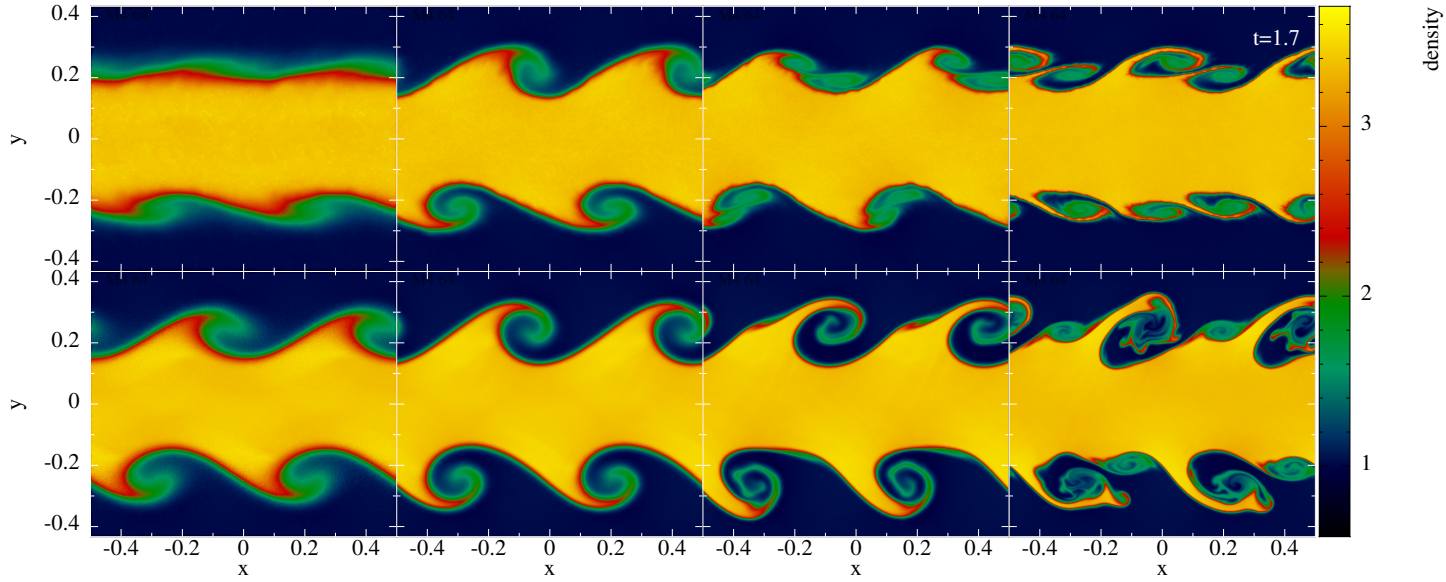

Figure 18. Density profiles obtained in the low-resolution 3D Kelvin-Helmholtz test described in Sect. 6.5 at $t = 1.7$ with the M_4 kernel (top panel) and the M_6 quintic kernel (bottom panel) respectively in code units. The instability is correctly captured with the M_6 kernel, similarly to the findings of Tricco (2019). From left to right, the numbers of particles N_l and N_r used along the x axis for the low-density and the high-density regions are: $N_\mathrm{l} = 128$ and $N_\mathrm{r} = 192$, $N_\mathrm{l} = 256$ and $N_\mathrm{r} = 384$, $N_\mathrm{l} = 512$ and $N_\mathrm{r} = 758$, $N_\mathrm{l} = 1024$ and $N_\mathrm{r} = 1536$, which corresponds to 215×10^3, 860×10^3, 3.4×10^6 and 13.7×10^6 particles respectively.

Taylor blast wave tests described in Sect. 6.4, fixing the time step to $dt = 10^{-5}$. This three-dimensional test is highly sensitive to rounding errors, primarily due to the presence of a low-density, zero-energy region surrounding the blast wave. In particular, aligning the behaviours of the shock viscosity parameter α^{AV} proves being particularly challenging. Finally, Fig. 19 shows that discrepancies between Shamrock and Phantom are imperceptible to the naked eye. Quantitatively, the $\mathcal{L}_2$ errors are

- relative $\mathcal{L}_2$ distance r : 2.0869658802024003e − 07,
- relative $\mathcal{L}_2$ distance h : 3.952645327403623e − 05,
- relative $\mathcal{L}_2$ distance v_r : 0.0005418229957181854,
- relative $\mathcal{L}_2$ distance u : 3.6622341394801246e − 05.

Following an in-depth examination, the sole identified distinctions between the two solvers are as follows: in Phantom, the shock parameter α^{AV} and the estimate of $\nabla \cdot \mathbf{v}$ are stored as single-precision floating-point numbers, while in Shamrock, they are double-precision.

6.6.2 Residuals: Low res Kelvin-Helmholtz instability test

We measure the residuals between Shamrock and Phantom by performing the Kelvin-Helmholtz instability test implemented in Phantom at commit number `e01f76c3`, at low resolution. Simulations are evolved to $t = 2$, while dumps are produced every $\Delta t = 0.1$ to sample the development of the instability. We choose the M4 kernel and a low number of particles to reveal the differences between the codes. Fig. 20 and Fig. 21 show the compared evolutions of the density and of the shock parameter respectively. At $t = 0.2$, no difference is observed in the density field. For shock viscosity, we observe for Phantom a small noise of relative amplitude $\lesssim 0.1\%$ along the line $y = 0.5$, which we attribute to $\alpha, h, \nabla \cdot \mathbf{v}$ being stored as single precision fields in Phantom. These fluctuations are not present in Shamrock, since these quantities are calculated in double precision. At $t = 0.5$ we can distinguish at this low resolution a small line of higher shock viscosity and density in Shamrock, which is not present in Phantom. We attribute these residuals to the fact that the Phantom simulation may have increased numerical viscosity due to single precision errors, while the SPH lattice in the Shamrock simulation is still reorganising. At $t = 1$, no significant difference is observed between the two simulations. Finally, at $t = 2$, tiny differences can be observed at the edges of the instability, for the same reasons as at $t = 1$.

6.7 Summary

The hydrodynamic SPH solver implemented in Shamrock passes successfully the standard tests (advection, Sod tube, Sedov-Taylor blast, Kelvin-Helmholtz instability). The implementation of the SPH solver in Shamrock is identical to that of Phantom. Results obtained with the two codes are almost indistinguishable, the residuals being attributed to the choice of floating-point precision for the quantities $h, \alpha, \nabla \cdot \mathbf{v}$. This sets the basis for rigorous performance comparison.

7 PERFORMANCE

7.1 Characteristics of the benchmarks

7.1.1 Hardware specificities

The tests performed to estimate performance with Shamrock were conducted on two systems. Single GPU and CPU tests were performed on an Nvidia A100-SXM4-40GB GPU of an Nvidia DGX workstation. This workstation is equipped with 4 Nvidia A100 40GB GPUs paired with an Epyc7742 64-core CPU, and are exploited via

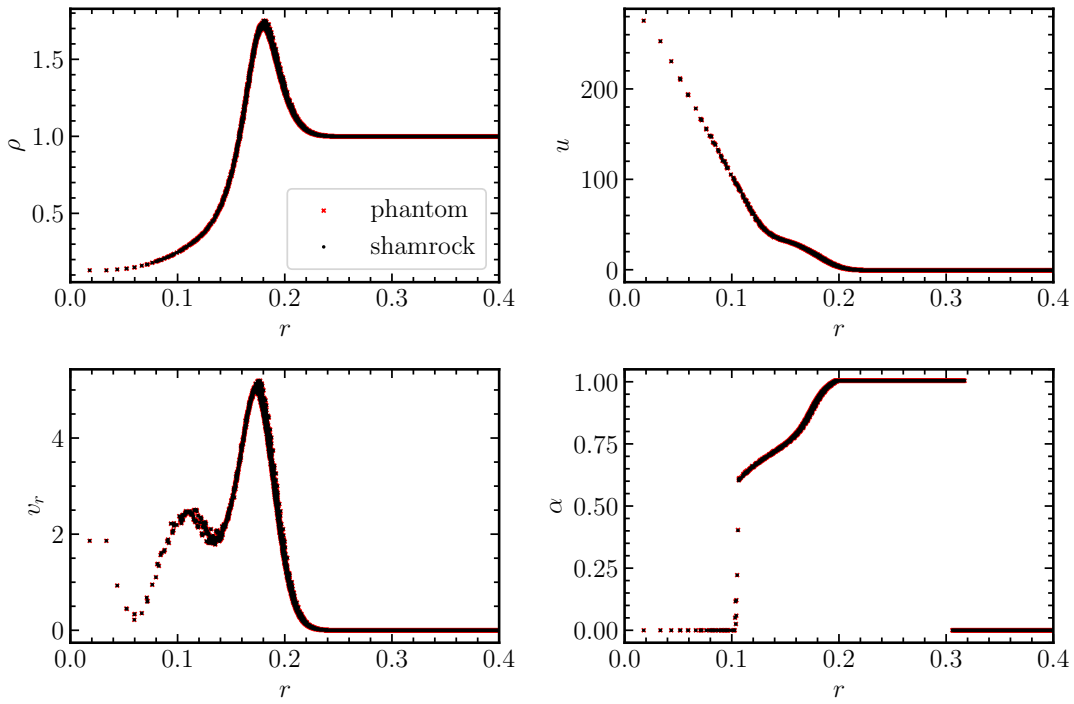

Figure 19. A comparison is made between the densities ρ, internal energies u, velocities v_r, and shock detection parameters α obtained at $t = 1$ from two identical low-resolution Sedov-Taylor blast wave tests conducted by Phantom (red dots) and Shamrock (black dots). Initially, Phantom is used to generate the same setup file for the two simulations. Runs are then conducted using a fixed time-step of $dt = 10^{-5}$. The dots are indistinguishable by eye (e.g. the $\mathcal{L}_2$ error on the velocity field is of order $5 \cdot 10^{-4}$): the implementations of the SPH solver are identical in the two codes.

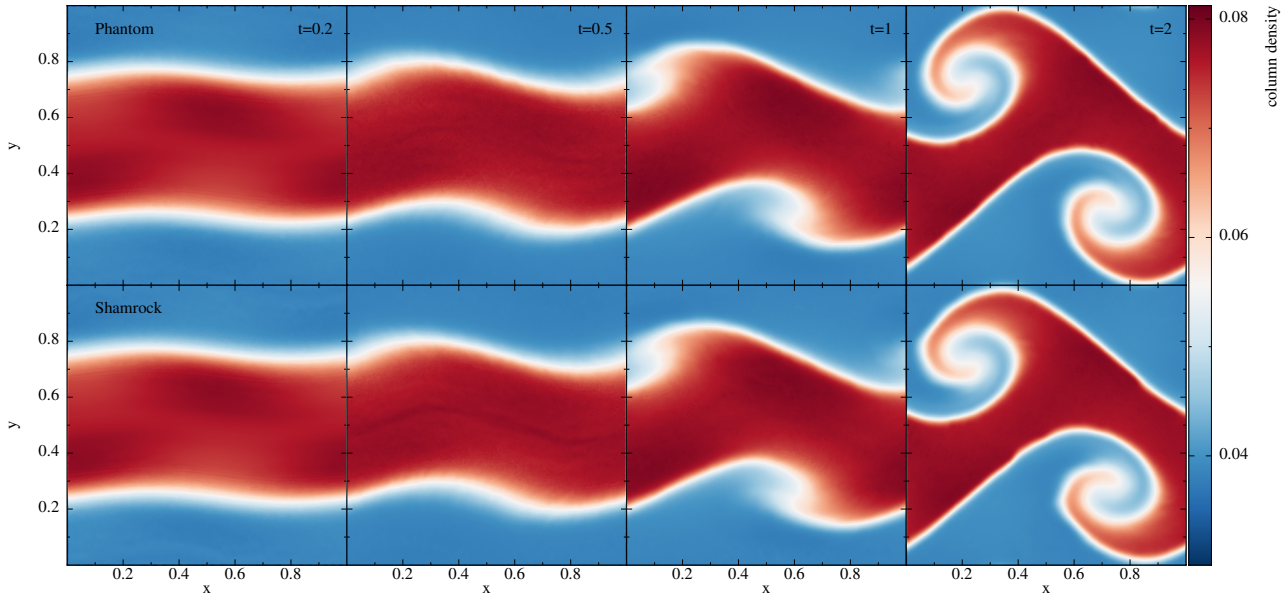

Figure 20. Density profiles obtained at $t = 0.2, 0.5, 1, 2$ on a low resolution Kelvin-Helmholtz test by Phantom (top panel) and Shamrock (bottom panel). Results are almost identical. The test is voluntarily performed with an unsuited M4 kernel at low-resolution to accentuate residual discrepancies between the two solvers. Those stem from the use of single precision in one and double precision in the other for shock detection variables.

SIDUS (Quemener & Corvellec 2013) by the Centre Blaise Pascal at ENS de Lyon . CPU tests were carried out on the CPU of the same DGX workstation using AdaptiveCPP OpenMP backend. For those Shamrock was compiled using `-O3 -march=native`. For single GPU tests of Shamrock compilation was performed using the Intel fork of the llvm/clang-19 compiler, also referenced as ONEAPI/DPC++ with optimizations `-O3 -march=native`. We voluntarily didn't used fast math optimisations as they would not be used in production. We use the CUDA/PTX backend of Intel llvm targetting the CUDA architecture `sm_s70` corresponding to the compute

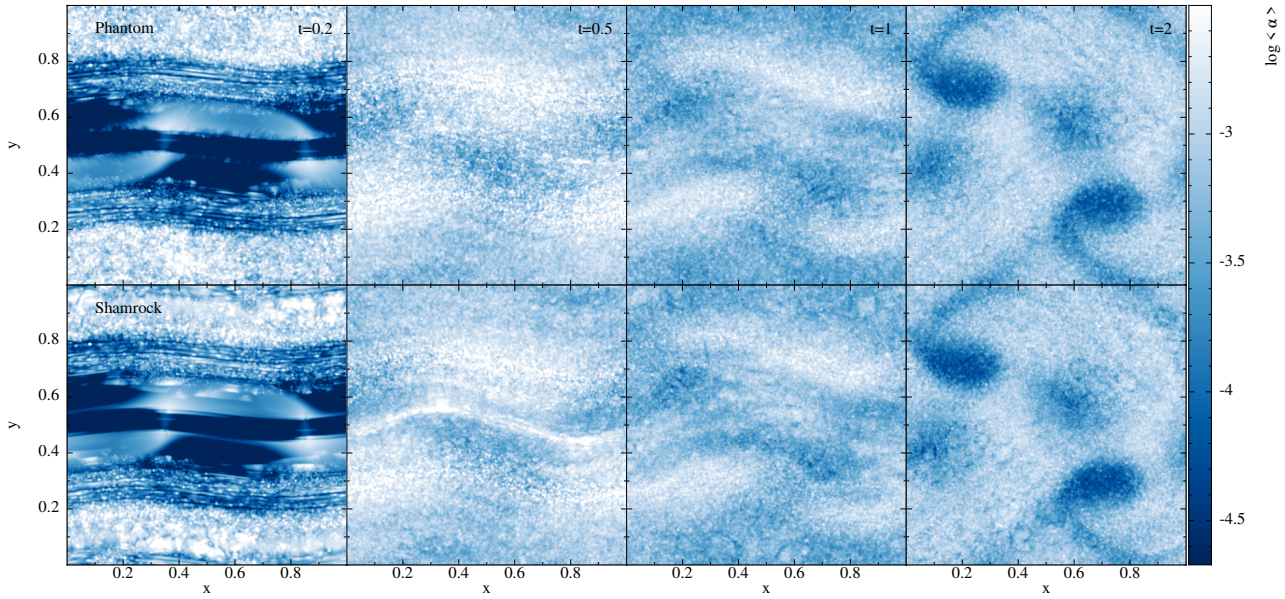

Figure 21. Same plot as in Fig.20, revealing the amplitude of truncature errors in the shock viscosity parameter. To our understanding, these errors represent the sole source of discrepancies between the implementations of SPH in Phantom and Shamrock.

capability of A100 GPUs. The Intel LLVM CUDA/PTX backend generates code using PTX ISA, the assembly language used to represent CUDA kernels. This result in a program that actually that is first lowered from C++ to PTX, then compiled using Nvidia `ptxas` tool, which enable the code to be profiled using Nvidia's CUDA tools. The CUDA version used is 12.0.

Multi-GPU and multi node tests were performed on Adastra Supercomputer at CINES in France, using up to 256 compute nodes, each compute node is a HPE Cray EX235a each equipped with 4 Mi250X GPUs paired with a 64 Cores AMD Epyc Trento CPU. On this platform we used ROCm/HIP backend of Intel llvm targetting the AMD GPU architecture `gfx90a` corresponding to the compute capability of Mi250X GPUs. The Intel llvm compiler was compiled using in the same module environment as Shamrock. We used Cray CPE 23.12 with acceleration on `gfx90a` and Trento on the host, in conjunction with the provided `PrgEnv-intel`. MPI with GPU aware support support was provided by the cray MPIch 8.1.26 module. ROCm support was provided by both `amd-mixed` 5.7.1 and `rocm` 5.7.1. Although the Mi250X GPU is a single chip, it is made up of two GCDs, which appear as separate instances on the compute node, where one MPI rank is assigned per GCDs.

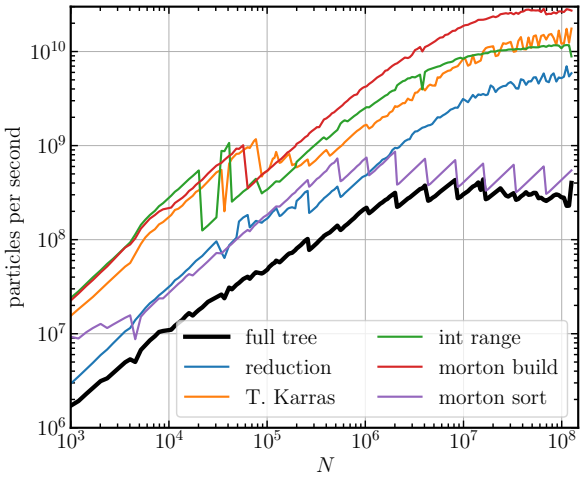

Figure 22. Benchmark of the performance of the tree building in Shamrock. Each curve represents the number of particles processed per second for various segments of the algorithm. The thick solid black curve shows the total time to build the tree. The other curves show the performance of the main algorithms involved in the tree building procedure. Those correspond to benchmarks of the isolated algorithms, which break the asynchronous nature of SYCL. As such, the sum of the individual times do not rigorously add up to the exact time of the entire algorithm. This benchmark used a dataset of input positions generated from an hexagonal closed packing lattice, with variations in lattice spacing. Varying the initial distribution of particles will not affect total performance of the tree, since overall, the building time is dominated by the bitonic sort. In this test, we used single precision Morton codes.

7.1.2 Setups

We present the performances of the SPH hydrodynamical solver of Shamrock on the Sedov Taylor blast wave, since it involves contributions of all the different terms in the hydrodynamical solver, and it is neither specific to astrophysics nor SPH. We compare the results with the one obtained with the hydro dynamical solver of Phantom with an almost identical implementation (see Sect. 6.6), on a computing units having similar power consumption.

We use the M_4 kernel and set $h_{fact} = 1.2$. To setup the lattice, we first consider a box of size $[-0.6, 0.6]^3$. For a desired number of particle N, the volume per particle is cV/N, where c is compacity of a close-packing of equal spheres. As such, the spacing dr between particles is $dr = (3cV/4\pi N)^{1/3}$. We then adapt the boundaries of the simulation volume to ensure periodicity in all directions for the initial close-packed lattice of particles.

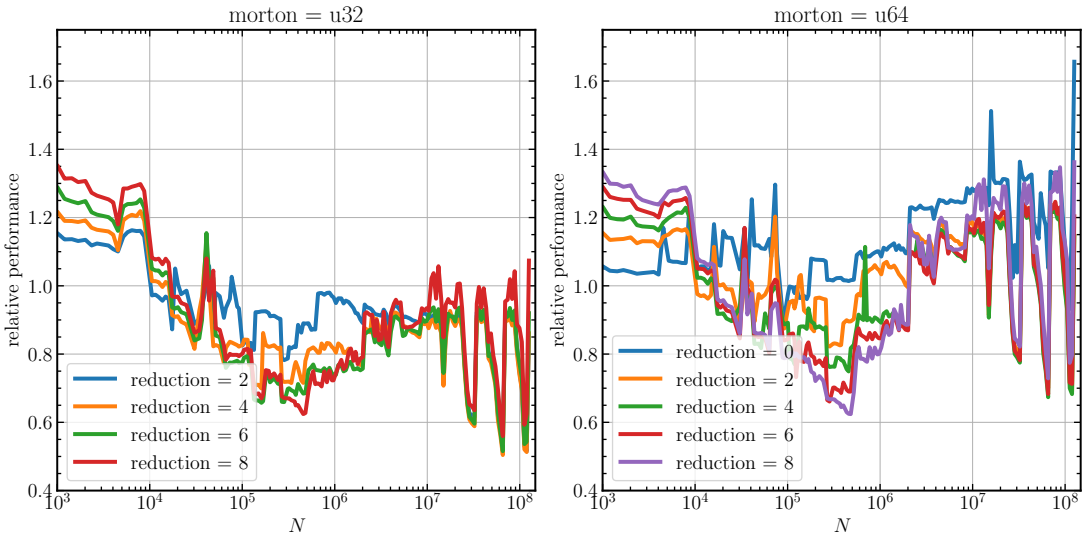

Figure 23. Relative performance of the complete tree building procedure for two different types of Morton codes (left: u32, right: u64), for different levels of reduction. The setup for this test is identical to the one presented in Fig. 22.

7.2 Performance of tree building

Fig. 22 shows the performance of the Shamrock tree building algorithm described in Sect. 4.11, by presenting results of tests carried out over 10^3 to 10^8 objects distributed on a regular cubic lattice. The results are presented in figures showing the number of object integrated to the tree per second, as a function of the total number of objects. This metric highlights the efficiency threshold of the GPU, where the computation time is shorter than the actual GPU programming overhead. It also highlights any deviation from a linear computation time as a function of the size of the input.

For a small to moderate number of objects $N \lesssim N_c$ where $N_c \sim 10^6$, the overhead of launching a GPU kernel leaves a significant inprint compared to the computational charge. A few million objects per GPU is the typical number of objects above which the algorithm can be used efficiently. For any $N \geq N_c$ tested, the tree is built at a typical constant rate of 5×10^{-9} s per object. Equivalently, 200 millions of objects per second are processed for Morton codes and the associated positions in double-precision.

For $N \geq N_c$, the algorithm achieves an almost constant performance, as long as it could be tested on current hardware. Fluctuations of up to 30% are observed for specific values of N. These peaks are consistent across several executions, and therefore probably due to the hardware scheduler on the GPU. Tree construction is dominated by the bitonic sorting algorithm (see Fig. 22). Since this algorithm does not depend on the values stored in the buffer, its performance is not affected by the spatial distribution of objects, and regular or randomly arranged points deliver the same performance. The performance of tree building of Shamrock is therefore independent of the distribution of objects considered.

Fig. 23 shows that performance is almost unaffected by the type (single, double, float or integer) used for the positions ($\sim 5 - 10\%$, spikes being probably due to the hardware scheduler). Performance is increased by a factor $\sim 30\%$ when the Morton code representation is reduced to single precision.

7.3 Performance of neighbour cache building

To measure performance of cache build and SPH time stepping, we first setup the particles as discussed in Sect. 7.2. Additionally, the smoothing length has been converged, resulting in 60 neighbours for the M4 kernel. After this setup, we perform a single time step. Fig. 24 reports the time spent during this time step to build the cache and perform the iteration. We compare the results obtained for the two strategies presented in Sect. 4.13 –4.14, along with different levels of reduction. We find that enhancing the reduction level results in better overall performance, particularly up to reduction level 6. For higher levels of reduction, performance drops as a consequence of the too large number of particles per leaf. Optimal configuration corresponds to $\sim$ 10 particles per leaf (reduction level of 4), which is similar to the number of particles per leaf in Phantom (Price et al. 2018). Additionally, integrating a two-stage neighbour cache alongside the reduction algorithm can double performance. To sum up, the combined use of reduction and a two-stage cache enhances cache building performance by tenfold, while doubling time-stepping performance.

7.4 Performance of time stepping

7.4.1 One GPU

We evolve setup a Sedov blast using Phantom git pulled at commit number `e01f76c3`, with compile flags `IND_TIMESTEPS=no MAXP=50000000` and evolve it with Phantom for a five timesteps and lower both CFL to 10^{-3} to avoid leapfrog corrector sub-cycling in both codes and produce a restart file. We then start both Shamrock and Phantom on the same restart for 5 iterations, to avoid result being affected by cache warm up. IO has carefully been subtracted from the Phantom measured time. The performance of Shamrock is first tested on a single A100-SXM4-40GB GPU, of total power 275 W. Fig. 25 shows the number of particles per second iterated as a function as the total number of particles N_{part} in the simulation. As expected, performance increase as the computational pressure on the GPU increases, up to the point where the solver becomes memory-bound ($\sim 10^6$ particles). Beyond this threshold, a typical speed of 12×10^6 particles per second is achieved. For a comparison, we perform a similar test with Phantom on an AMD Epyc7742 CPU. On this architecture, Phantom fully exploit its OpenMP parallelisation across

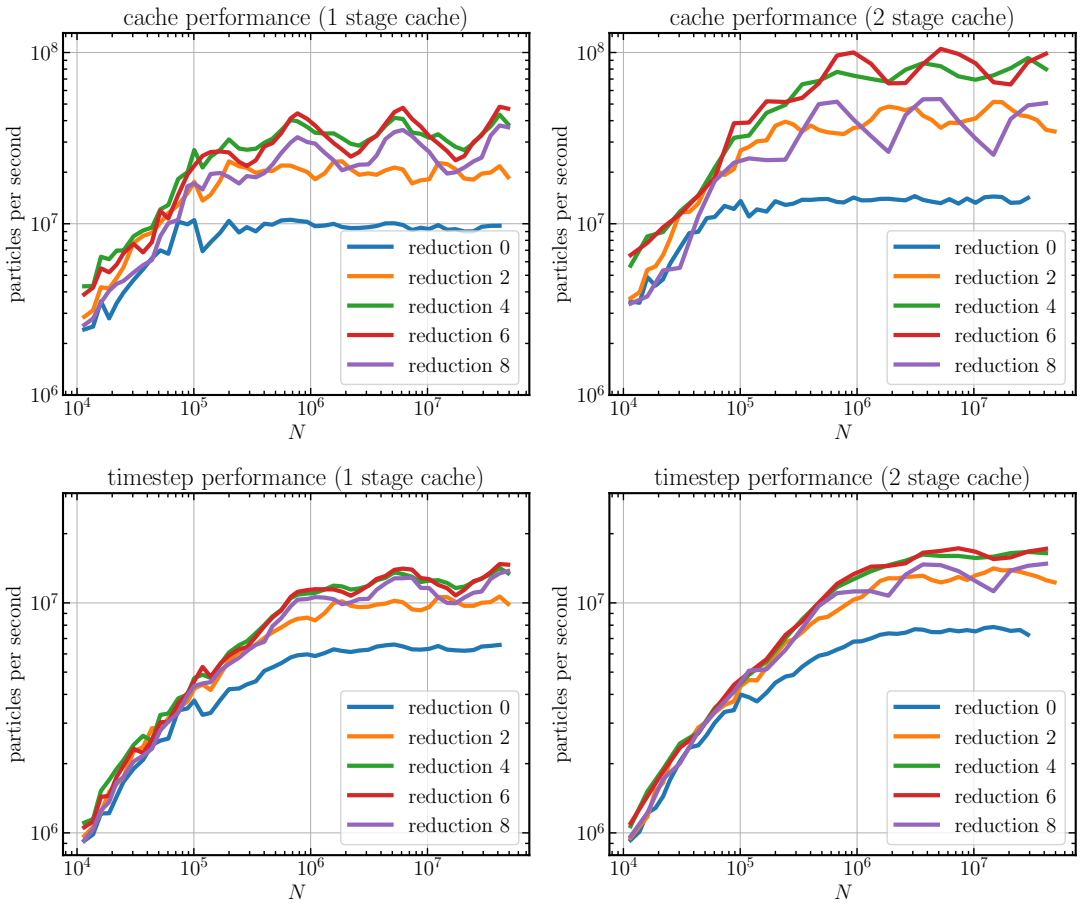

Figure 24. Performances of cache building and time stepping measured on one time step of the Sedov-Taylor blast wave test presented in Fig. 16. Increasing the level of reduction yields improved performance overall up to reduction level 6. Additionally, using two-stages neighbour caching improves performance up to a factor of two when used in conjunction with the reduction algorithm. In total, employing both reduction and a two-stage cache enhances cache building performance by a factor of ten, while doubling time-stepping performance.

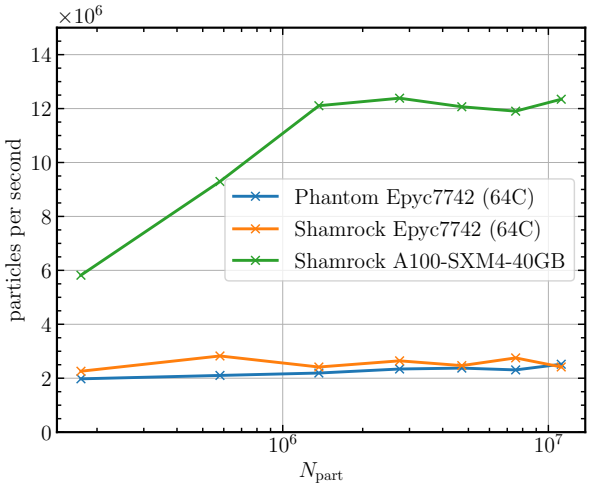

Figure 25. Comparative benchmark of Shamrock and Phantom ran on a same restart file of a Sedov-Taylor blast at multiple resolutions produced by Phantom. Shamrock does a achieve a slightly higher performance on CPU compared to Phantom, when run a on a single NVIDIA A100-SXM4-40GB GPU the performance is around 5 times higher for large datasets. Small datasets are not large enough to saturate the GPU explaining the lowered performance on GPU below 10^6 particles.

the 64 cores (128 threads). The power consumption is also similar to the one of the A100-SXM4-40GB used for Shamrock ($\sim$ 275W). For the test described above, one obtains $\gtrsim 2 \times 10^6$ particles per second in most cases. Note that on this Epyc7742 CPU architecture, Shamrock (compiled using AdaptiveCPP OpenMP backend) achieves slightly higher performance. Despite the limitations inherent in such a comparison, we estimate that Shamrock attains approximately a $\sim$ 5 factor gain in performance when executed on a single GPU compared to a state-of-the-art SPH CPU code with equivalent power consumption.

7.4.2 Multiple GPUs

We perform the multi-GPU test of Shamrock on the Adastra supercomputer of the French CINES, in its early February 2024 configuration. In this multi-GPU test, the split criterion of patches is set at one-sixth of the number of particles per GPU, guaranteeing a minimum of 8 patches per MPI process. We evolve over 5 time steps of the Sedov test and report the performance obtained on the last time step. Unlike in the case of a single GPU, we do not evolve the simulation prior to measurement to limit computational expenses. The scale at which internal energy is injected remains consistent across all tests.

Fig. 26 shows that for 65×10^9 particles distributed over 1000 GPUs (64×10^6 particles per GPU), Shamrock achieves 9×10^9 particles

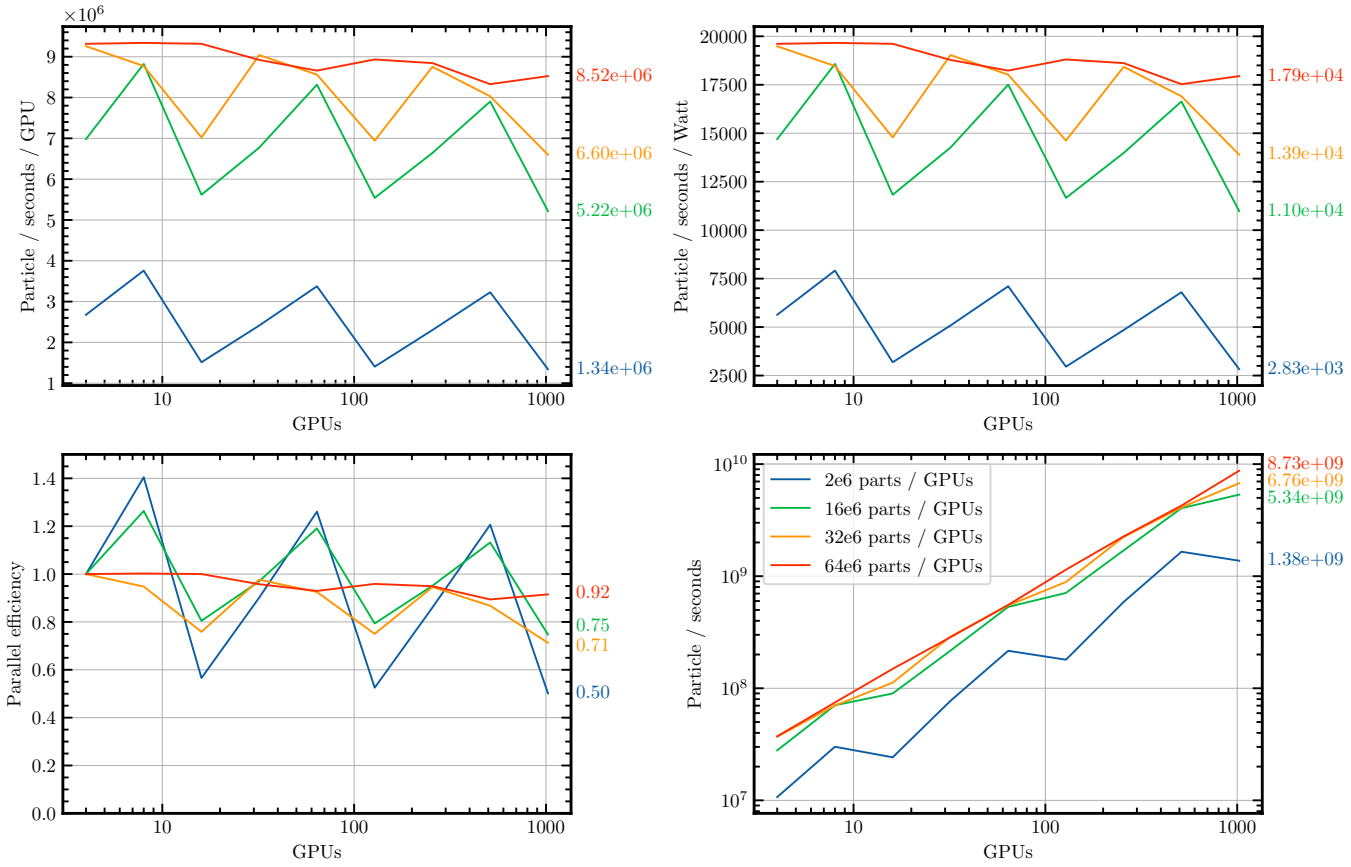

Figure 26. Weak scaling tests conducted on the CINES Adastra supercomputer. These tests are performed for multiple resolutions, from 1 node to 256 nodes, corresponding to using 4 GPUs with 8 MPI ranks to 1024 GPUs with 2048 MPI ranks. In these tests, we use the setup of a Sedov-Taylor blast and report the number of particles per GPU. The patch decomposition is set to have at least 8 patches per MPI ranks. We observe that for large simulations, the scaling test results in 92% parallel efficiency on 65 billion particles at 9 billion particle per seconds. Lowering the base resolution reduce the per-GPU performance since GPUs start to be under-utilised. Additionally the variation of the number of particle per patch result in variation in per-GPU performance, resulting in a saw-tooth pattern. We also report the energy efficiency of the tests, were the power consumption used was measured in the single node case and extrapolated as being the product of the number of nodes times the single node consumption.

iterated per second. Consequently, iterating one time step over the entire 65×10^9 particles requires 7 seconds on this cluster. These results correspond to around 1.5 times the performance achieved by a single A100 on the same test at the same commit, a value close to what is expected given the hardware specifications. This demonstrates no significant deviation in behaviour attributable to the choice of GPUs. To achieve good load balancing, we find that we need around 10 patches per MPI process. On Adastra, this translates to 20 patches per GPU, amounting to a bit over 1 million particles per Mi250x Graphics Compute Die. Fig. 25 shows that for small number of particles, the GPU execution units are not loaded efficiently. A correct load corresponds to a typical 2 millions of particles per GPU. Fig. 26 also reveals saw tooth shape as the number of particles increases. This feature can be interpreted by noting that every multiple of 8 GPUs, the patches are divided to avoid becoming too large, causing performance to drop below the efficiency threshold. Efficiency then increases again with the number of particles, until another factor of 8 in the resolution is reached, necessitating further patch refinement. Additionally on Fig. 26 we also report the energy efficiency of the weak scaling tests. We measure on every single nodes tests the power consumption related to an iteration of the solver using the hardware counters of the HPE Cray EX235a node. The reported value for multiple nodes is extrapolated from the single node case assuming a total power consumption being the product of the number of nodes times the single node power consumption time the parallel efficiency. Finally we report the power efficiency measured in particles per second per Watt which is also the number of particles processes with a single Joule. The total power consumption of a node in those tests is not very sensible to the number of particles per GPUs. However the GPU performance can be significantly reduced when GPUs do not have enough particle to process, and having a larger number of particle per GPUs result in the highest efficiency. Maximising the number of particles per GPU maximises efficiency in most cases.

7.5 Summary

The larger the simulation, the higher the performance per GPU. Multi-GPU architectures therefore require large simulations to scale and be energy-efficient. With Shamrock, this effect can be mitigated by reducing the number of patches, albeit with the potential drawback of rising load balancing issues. Similar benchmarks would be required for further implementations in Shamrock of additional physical processes or setups, possibly involving different particle distributions.

8 PERSPECTIVES

8.1 Multi-physics

Multi-scale astrophysical problems are often multi-physical. To be consistent, very high-resolution simulations must include realistic physics. The version of Shamrock discussed in this article focuses on a purely hydrodynamic SPH solver. Next steps of development consist of implementing local algorithms to address the radiative dynamics of magnetised and dusty fluids. In principle, the modular format of Shamrock facilitates the assembly of a new set of known numerical equations into a solver. The biggest challenge is the implementation of gravity, a non-local interaction that requires a new algorithmic layer based on group-group interactions. Two main algorithms are used to handle gravity by the various astrophysical codes. SPH codes mainly use the method of Fast Multipole Moments (FMM), which takes advantage of the tree structure inherent in particle methods. The technical hurdle lies in achieving numerical efficiency, even for high opening angles involving summation over hundreds or even thousands of neighbours. An alternative approach is to employ a multigrid method, but to our knowledge this has not yet been implemented in an SPH code. These additional physical elements require the implementation of an individual time step to maximise performance. It will consequently be also possible to take advantage of Shamrock's efficiency to benchmark individual time stepping against fixed time stepping in simulations that were previously not tractable.

8.2 Multi-methods

The Shamrock framework relies on the efficient construction and traversal of its tree, irrespective of the numerical object considered. In this study, we have considered particles to develop an SPH solver, but these can be replaced by Eulerian cells in an agnostic way. Since the tree algorithm of Shamrock scales almost perfectly to any disordered particle distribution, we can expect similar performance, even on an AMR (Adaptive Mesh Refinement) grid. In principle, various algorithms can be implemented on this grid (such as finite differences or finite volumes), with the moderate cost of incorporating a few specific modules tailored to these solvers (like the accumulation of flows on faces for finite volumes).

The advantages of such a unified framework are twofold. Firstly, to validate an astrophysical model by achieving consistent results with inherently different methods, while the physical model used is rigorously identical (e.g. opacities, cooling rates, resistivities, chemical networks, equations of state). Secondly, to enable rigorous evaluation of numerical methods in terms of accuracy and computational efficiency for specific numerical problems. To our knowledge, no such framework currently exists.

8.3 Data analysis

The efficiency of the Shamrock tree means that data analyses can be carried out very efficiently, whether on particles, cells or both. We plan to develop a native library that will enable these analyses to be performed efficiently on multi-GPU parallel architectures.

8.4 Optimisation of latencies

Finally, the Shamrock framework has been designed to optimise performance on multi-node architectures. As discussed in the previous section, specificities of modern hardware imply that performance increases with the computational load demanded of the GPUs. Maybe counter-intuitively, Shamrock is therefore not designed by default to run small simulations with a large number of iterations. Since efficient execution of such simulations remains a complementary challenge to that of Exascale, a layer of optimisations regarding remnant latencies remains to be implemented. This would enable users to employ Shamrock for both small and large-scale simulations. We will benefit from the knowledge of the work done by the Gromacs team towards optimising latencies in SYCL (Alekseenko et al. 2024).

9 CONCLUSION

We introduced Shamrock, a modular and versatile framework designed to run efficiently on multi-GPUs architectures, towards Exascale simulations. The efficiency of Shamrock is due to its tree, based on a fully parallel binary logic (Karras algorithm). On a single GPU of an A100, the algorithm builds a tree for 200 million particles in one second. The tree traversal speed, while summing over approximately 60 neighbours, reaches 12 million particles per second per GPU. This property makes it possible to build a Smoothed Particle Hydrodynamics (SPH) solver where neighbours are not stored but recalculated on the fly, reconstructing a tree almost instantaneously.

To exploit the efficiency of this framework, we have implemented and tested an hydrodynamic SPH solver in Shamrock. For a Sedov test performed with 10^6 particles on a single A100 GPU, a Shamrock simulation is around ~6 times faster than an identical simulation performed with Phantom on an Epic 7742 multicore CPU architecture of equivalent power. The parallelisation of Shamrock on several nodes relies on an MPI protocol with hollow communications between the interfaces of a patch system that groups calculations performed on different GPUs. Shamrock 's scaling has been tested on the Adastra supercomputer (2000 mi250x GPUs). As expected, the higher the computational load on the GPU, the better the efficiency of the code. For 32×10^6 particles per GPU and 65 billions of particles in total, we achieve 92% efficiency at low scaling, in a simulation where 9×10^9 particles are iterated per second. Iterating one time step over the 65×10^9 particles takes therefore 7 seconds on this architecture. Shamrock is therefore a promising framework that will soon be extended to other grid-based algorithms with adaptive refinement.

ACKNOWLEDGEMENTS

We warmly thank G. Lesur for guidance on the use of multiple GPU systems, and advices on Grid-5000 for prototyping, as well as on the Adastra cluster. We also thank A. Alpay (AdaptiveCPP) and A. Alekseenko (Gromacs) for guidance on the use of SYCL and related optimizations, F. Lovascio, B. Commerçon, L. Sewanou, J. Fensch, A. Durocher and L. Marchal for useful comments and discussions, E. Quemener and the Centre Blaise Pascal de Simulation et de Modélisation Numérique for support on the benchmarks, the ENS de Lyon for partly funding the DGX Nvidia Workstation and the electric costs associated to local simulations, the CINES for having granted access and support for the Adastra machine during the duration of the Adastra GPU hackaton in Feb. 2024, and in particular E. Malaboeuf, J.-Y. Vet for technical discussions. A. Charlet, E. Lynch, R. Lenoble for comments on the manuscript. The Shamrock code follows the developments of the WP 5.1 of the Programme et équipements prioritaires de recherche (PEPR) *Origins* (PI: A.

Morbidelli). We acknowledge funding from the ERC CoG project PODCAST No 864965.

DATA AVAILABILITY

The relevant sources can be found in the Github repository of the code (https://github.com/Shamrock-code/Shamrock)

REFERENCES

Abraham M. J., Murtola T., Schulz R., Páll S., Smith J. C., Hess B., Lindahl E., 2015, SoftwareX, 1, 19

Adinets A., Merrill D., 2022, arXiv preprint arXiv:2206.01784

Alekseenko A., Páll S., 2023, in Proceedings of the 2023 International Workshop on OpenCL. IWOCL '23. Association for Computing Machinery, New York, NY, USA, doi:10.1145/3585341.3585350, https://doi.org/10.1145/3585341.3585350

Alekseenko A., Páll S., Lindahl E., 2024, arXiv preprint arXiv:2405.01420

Alpay A., Heuveline V., 2020, in Proceedings of the International Workshop on OpenCL. IWOCL '20. Association for Computing Machinery, New York, NY, USA, doi:10.1145/3388333.3388658, https://doi.org/10.1145/3388333.3388658

Alpay A., Soproni B., Wünsche H., Heuveline V., 2022, in International Workshop on OpenCL. IWOCL'22. Association for Computing Machinery, New York, NY, USA, doi:10.1145/3529538.3530005, https://doi.org/10.1145/3529538.3530005

Arkhipov D. I., Wu D., Li K., Regan A. C., 2017, arXiv e-prints, p. arXiv:1709.02520

Balsara D. S., 1995, Journal of Computational Physics, 121, 357

Batcher K. E., 1968, in Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference. AFIPS '68 (Spring). Association for Computing Machinery, New York, NY, USA, p. 307–314, doi:10.1145/1468075.1468121, https://doi.org/10.1145/1468075.1468121

Bedorf J., Portegies Zwart S., 2020, SciPost Astronomy, 1, 001

Blelloch G. E., 1990, School of Computer Science, Carnegie Mellon University Pittsburgh, PA, USA

Chow E., Monaghan J. J., 1997, Journal of Computational Physics, 134, 296

Courant R., Friedrichs K., Lewy H., 1928, Mathematische Annalen, 100, 32

Cullen L., Dehnen W., 2010, MNRAS, 408, 669

Deakin T., McIntosh-Smith S., 2020, in Proceedings of the International Workshop on OpenCL. IWOCL '20. Association for Computing Machinery, New York, NY, USA, doi:10.1145/3388333.3388643, https://doi.org/10.1145/3388333.3388643

Dehnen W., Aly H., 2012, MNRAS, 425, 1068

Gafton E., Rosswog S., 2011, MNRAS, 418, 770

Gingold R. A., Monaghan J. J., 1977, MNRAS, 181, 375

Grete P., et al., 2022, arXiv e-prints, p. arXiv:2202.12309

Hairer E., Lubich C., Wanner G., 2003, Acta Numerica, 12, 399

Hopkins P. F., 2014, GIZMO: Multi-method magneto-hydrodynamics+gravity code, Astrophysics Source Code Library, record ascl:1410.003

Hopkins P. F., 2015, MNRAS, 450, 53

Horn D., 2005, Gpu gems, 2, 573

Hubber D. A., Batty C. P., McLeod A., Whitworth A. P., 2011, A&A, 529, A27

Jakob W., Rhinelander J., Moldovan D., 2024, URL: https://github.com/pybind/pybind11

Jin Z., Vetter J. S., 2022, in Proceedings of the 13th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics. BCB '22. Association for Computing Machinery, New York, NY, USA, doi:10.1145/3535508.3545591, https://doi.org/10.1145/3535508.3545591

Karras T., 2012, in Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics. pp 33–37

Landshoff R., 1955, Technical report, A numerical method for treating fluid flow in the presence of shocks. Los Alamos National Lab.(LANL), Los Alamos, NM (United States)

Lattanzio J., Monaghan J., Pongracic H., Schwarz M., 1986, SIAM Journal on Scientific and Statistical Computing, 7, 591

Lattner C., Adve V., 2004, in International Symposium on Code Generation and Optimization, 2004. CGO 2004.. pp 75–86, doi:10.1109/CGO.2004.1281665

Lauterbach C., Garland M., Sengupta S., Luebke D. P., Manocha D., 2009, Computer Graphics Forum, 28

Lesur G. R. J., Baghdadi S., Wafflard-Fernandez G., Mauxion J., Robert C. M. T., Van den Bossche M., 2023, arXiv e-prints, p. arXiv:2304.13746

Lodato G., Price D. J., 2010, MNRAS, 405, 1212

Lucy L. B., 1977, AJ, 82, 1013

Margolin L. G., Lloyd-Ronning N. M., 2022, arXiv e-prints, p. arXiv:2202.11084

Markomanolis G. S., et al., 2022, in Supercomputing Frontiers: 7th Asian Conference, SCFA 2022, Singapore, March 1–3, 2022, Proceedings. Springer-Verlag, Berlin, Heidelberg, p. 79–101, doi:10.1007/978-3-031-10419-0_6, https://doi.org/10.1007/978-3-031-10419-0_6

Merrill D., Garland M., 2016, NVIDIA, Tech. Rep. NVR-2016-002

Monaghan J. J., 1997a, Journal of Computational Physics, 136, 298

Monaghan J. J., 1997b, Journal of Computational Physics, 136, 298

Monaghan J. J., 2002, MNRAS, 335, 843

Monaghan J. J., Price D. J., 2001, MNRAS, 328, 381

Morris J. P., 1996, PhD thesis, -

Morris J. P., Monaghan J. J., 1997, Journal of Computational Physics, 136, 41

Morton G. M., 1966, International Business Machines Company New York

Nassimi Sahni 1979, IEEE Transactions on Computers, C-28, 2

Noh W. F., 1987, Journal of Computational Physics, 72, 78

Price D. J., 2008, Journal of Computational Physics, 227, 10040

Price D. J., 2012, Journal of Computational Physics, 231, 759

Price D. J., Federrath C., 2010, MNRAS, 406, 1659

Price D. J., et al., 2018, Publ. Astron. Soc. Australia, 35, e031

Quemener E., Corvellec M., 2013, Linux Journal, 2013, 3

Samet H., 2006, Foundations of multidimensional and metric data structures. Morgan Kaufmann

Schaal K., Bauer A., Chandrashekar P., Pakmor R., Klingenberg C., Springel V., 2015, MNRAS, 453, 4278

Schaller M., Gonnet P., Draper P. W., Chalk A. B. G., Bower R. G., Willis J., Hausammann L., 2018, SWIFT: SPH With Inter-dependent Fine-grained Tasking, Astrophysics Source Code Library, record ascl:1805.020

Schoenberg I. J., 1946, Quarterly of Applied Mathematics, 4, 45

Sedov L. I., 1959, Similarity and Dimensional Methods in Mechanics

Seliger R. L., Whitham G. B., 1968, Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences, 305, 1

Sod G. A., 1978, Journal of Computational Physics, 27, 1

Springel V., Hernquist L., 2002, MNRAS, 333, 649

Springel V., Pakmor R., Zier O., Reinecke M., 2021, MNRAS, 506, 2871

Taylor G., 1950a, Proceedings of the Royal Society of London Series A, 201, 159

Taylor G., 1950b, Proceedings of the Royal Society of London Series A, 201, 175

Tricco T. S., 2019, MNRAS, 488, 5210

Trott C., et al., 2021, Computing in Science and Engineering, 23, 10

Verlet L., 1967, Physical Review, 159, 98

Von Neumann J., Richtmyer R. D., 1950, Journal of Applied Physics, 21, 232

Wadsley J. W., Stadel J., Quinn T., 2004, New Astron., 9, 137

Wendland H., 1995, Advances in computational Mathematics, 4, 389

Wibking B. D., Krumholz M. R., 2022, MNRAS, 512, 1430

APPENDIX A: SOFTWARE DESIGN

A1 Development

A1.1 Codebase organisation

The Shamrock project aims to be fully modular, in the sense that it is made up of several cmake projects which are connected using standardised interfaces. For example, the algorithmic library of Shamrock is a cmake sub-project that depends on the backend library. This allows the shamrock sub-projects to be as independent as possible, avoiding merge conflicts and enabling development efforts to be better focused. To date, the project comprises 12 sub-projects. This number is very likely to change in the future, with future additions and refactoring.

A1.2 Git

The Shamrock project is hosted on GitHub. We adopt a methodology akin to the one employed by the LLVM project (Lattner & Adve 2004), where the main branch is protected and can only be modified by pull requests from the feature/fix branches from contributors forks of the project. Releases are performed by branching from the main branch, facilitating the implementation of fixes to existing versions of the code. The CI test pipeline is routinely executed on GitHub, assessing both the main branch and all incoming pull requests. Successful completion of all tests is mandatory for changes to be merged into the main branch.

A2 Testing

Numerous unit testing and validation options are available for C++. However, none of the standard solutions available match our specific requirements, the main one being that tests are integrated with MPI . Because of this constraint, we have developed our own in-house test library, designed to provide the main features of gtest, while retaining the ability to specify the number of MPI ranks for a particular test. The current test library is capable of performing unittest, validation tests, and benchmarks. On GitHub, we use self-hosted runners to perform the tests with multiple configurations of compilers, targets and versions.

A3 Environment scripts

Compiling Shamrock on different machines entails dealing with a wide range of diversity. Typical technical aspects involve setting up LLVM, MPI and SYCL, which may involve numerous steps on a machine with missing libraries or having complex configuration. To ensure consistency in Shamrock configuration across machines, we have designed environment scripts. These scripts aim to produce a build directory with all the requirements for building the code, as well as to provide an 'activate' script in this folder, which configures the environment variable and loads the correct modules by sourcing them. In addition, these scripts offer utility functions such as

- setupcompiler: Setup the SYCL compiler
- updatecompiler: Update the environment
- shamconfigure: Configure Shamrock
- shammake: Build Shamrock

This functionality is provided by a 'new-env' script that configures the build directory with all requirements, including the compiler SYCL, automatically. In summary, only 5 commands are needed to build a working version of Shamrock, an example would be

```
# Setup the environment
./env/new-env \
    --builddir build \
    --machine debian-generic.acpp \
    -- \
    --backend cuda \
    --arch sm_70

# Now move in the build directory
cd build
# Activate the workspace, which will
# define some utility functions
source activate
# Configure Shamrock
shamconfigure
# Build Shamrock
shammake
```

A4 Runscripts

In Shamrock, our aim is to handle setup files and configuration files that would allow great versatility in the use of the code, as well as on-the-fly analysis. Handling such a complexity through configuration files alone is both difficult and non-standard. Moreover, a user should not be required to know C++ to be able to use the code. Using a Python frontend offers a suitable solution to ensure both code versatility and ease of use. To do this, we use pybind11 (Jakob et al. 2024), which allows to map C++ functions or classes from the C++ source code to a '`shamrock`' python library. In the current version

```
import shamrock

# Create a Shamrock context
ctx = shamrock.Context()
ctx.pdata_layout_new()

# Get the SPH model
model = shamrock.get_SPHModel(
    context = ctx,
    vector_type = "f64_3",
    sph_kernel = "M6")

# configure the solver
cfg = model.gen_default_config()
cfg.set_artif_viscosity_VaryingCD10(
    alpha_min = 0.0,
    alpha_max = 1,
    sigma_decay = 0.1,
    alpha_u = 1,
    beta_AV = 2)
cfg.set_boundary_periodic()
cfg.set_eos_adiabatic(gamma = 5./3.)
cfg.print_status()

model.set_solver_config(cfg)
model.set_cfl_cour(0.3)
model.set_cfl_force(0.25)

# Initialise the patch scheduler
model.init_scheduler(
    split_crit = 1e6,
    merge_crit = 1e4)

#  .... Do setup ....

# Run the simulation until t=1 and dump
t_end = 1.0
model.evolve_until(t_end)
dump = model.make_phantom_dump()
dump.save_dump("output")
```

Figure A1. Example of a simplified Shamrock runscript

of Shamrock, two uses are possible. The first is to use Shamrock as a python interpreter that will go through and execute the content of a runscript (the script of a Shamrock run), which can include, if desired, configuration, simulation and post-processing in a single run and script (see Fig. A1 for an example of a runscript).

The other use is to compile Shamrock as a Python library and install it through pip, enabling the code to be used in Jupyter notebooks. Using Shamrock as a Python library is ideal for local machine prototyping, while on a cluster, employing Shamrock as a Python interpreter is highly recommended.

A5 Units

In Shamrock we have chosen to use code units which are a rescaling of base SI units, where the factor is chosen at runtime in the runscript.

APPENDIX B: AABB EXTENSION/INTERSECTION PERMUTATION

We prove the following theorem:

$$AABB_1 \oplus h \cap AABB_2 \neq \emptyset \Leftrightarrow AABB_1 \cap AABB_2 \oplus h \neq \emptyset,$$

where $AABB \oplus l$ is the operation that extends the AABB in every direction by a distance l. One initial observation is that an AABB is equivalent to a ball defined using the infinity norm $||\cdot||_\infty$. Consequently, the intersection of two AABBs is the result of intersecting along each axis independently. Formally, define a first AABB 1 as the Cartesian product of three intervals $AABB_1 = I_{1,x} \times I_{1,y} \times I_{1,z}$, and a second AABB as $AABB_2 = I_{2,x} \times I_{2,y} \times I_{2,z}$. Their intersection is $AABB_1 \cap AABB_2 = (I_{1,x} \cap I_{2,x}) \times (I_{1,y} \cap I_{2,y}) \times (I_{1,z} \cap I_{2,z})$. Hence, proving the theorem in one dimensions directly extends to three dimensions. Consider now two one-dimensional intervals $I_1 = [\alpha_1, A_1], I_2 = [\beta_1, B_1]$. With $d(a, b)$ the distance in one dimension between a point a and b, and $B(r, h)$ being a ball in one dimension of position r and radius h, we have

$$\begin{aligned}
&\emptyset \neq I_1 \oplus h \cap I_2 \\
\Leftrightarrow\ &\emptyset \neq [\alpha_1 - h, A_1 + h] \cap [\beta_1, B_1] \\
\Leftrightarrow\ &\emptyset \neq B\left(\frac{A_1+\alpha_1}{2}, \frac{A_1-\alpha_1}{2} + h\right) \cap B\left(\frac{B_1+\beta_1}{2}, \frac{B_1-\beta_1}{2}\right) \\
\Leftrightarrow\ &d\left(\frac{A_1+\alpha_1}{2}, \frac{B_1+\beta_1}{2}\right) \leq \frac{A_1-\alpha_1}{2} + h + \frac{B_1-\beta_1}{2} \\
\Leftrightarrow\ &\emptyset \neq B\left(\frac{A_1+\alpha_1}{2}, \frac{A_1-\alpha_1}{2}\right) \cap B\left(\frac{B_1+\beta_1}{2}, \frac{B_1-\beta_1}{2} + h\right) \\
\Leftrightarrow\ &\emptyset \neq [\alpha_1, A_1] \cap [\beta_1 - h, B_1 + h] \\
\Leftrightarrow\ &\emptyset \neq I_1 \cap I_2 \oplus h,
\end{aligned}$$

which completes the proof.

This paper has been typeset from a TEX/LATEX file prepared by the author.