

















SWIFT : a modern highly parallel gravity and smoothed particle hydrodynamics solver for astrophysical and cosmological applications

Matthieu Schaller ^{1,2}★ Josh Borrow ^{3,4,5} Peter W. Draper,⁵ Mladen Ivkovic,^{6,7,8} Stuart McAlpine ^{9,10}
 Bert Vandembroucke ^{2,11} Yannick Bahé ^{2,5,6} Evgenii Chaikin ² Aidan B. G. Chalk,¹²
 Tsang Keung Chan ^{5,13,14} Camila Correa ^{15,16} Marcel van Daalen ² Willem Elbers ⁵
 Pedro Gonnet,¹⁷ Loïc Hausammann,^{6,18} John Helly,⁵ Filip Huško ⁵ Jacob A. Kegerreis ^{5,19}
 Folkert S. J. Nobels ² Sylvia Ploeckinger,^{1,20} Yves Revaz ⁶ William J. Roper ²¹
 Sergio Ruiz-Bonilla ⁵ Thomas D. Sandnes,⁵ Yolán Uyttenhove,¹¹ James S. Willis²² and Zhen Xiang^{1,23,24}

Affiliations are listed at the end of the paper

Accepted 2024 March 28. Received 2024 March 27; in original form 2023 May 22

ABSTRACT

Numerical simulations have become one of the key tools used by theorists in all the fields of astrophysics and cosmology. The development of modern tools that target the largest existing computing systems and exploit state-of-the-art numerical methods and algorithms is thus crucial. In this paper, we introduce the fully open-source highly-parallel, versatile, and modular coupled hydrodynamics, gravity, cosmology, and galaxy-formation code SWIFT. The software package exploits hybrid shared- and distributed-memory task-based parallelism, asynchronous communications, and domain-decomposition algorithms based on balancing the workload, rather than the data, to efficiently exploit modern high-performance computing cluster architectures. Gravity is solved for using a fast-multipole-method, optionally coupled to a particle mesh solver in Fourier space to handle periodic volumes. For gas evolution, multiple modern flavours of Smoothed Particle Hydrodynamics are implemented. SWIFT also evolves neutrinos using a state-of-the-art particle-based method. Two complementary networks of sub-grid models for galaxy formation as well as extensions to simulate planetary physics are also released as part of the code. An extensive set of output options, including snapshots, light-cones, power spectra, and a coupling to structure finders are also included. We describe the overall code architecture, summarize the consistency and accuracy tests that were performed, and demonstrate the excellent weak-scaling performance of the code using a representative cosmological hydrodynamical problem with ≈ 300 billion particles. The code is released to the community alongside extensive documentation for both users and developers, a large selection of example test problems, and a suite of tools to aid in the analysis of large simulations run with SWIFT.

Key words: methods: numerical – software: public release – software: simulations.

1 INTRODUCTION

Over the last four decades, numerical simulations have imposed themselves as the key tool of theoretical astrophysics. By allowing the study of the highly non-linear regime of a model, or by allowing *in-silico* experiments of objects inaccessible to laboratories, simulations are essential to the interpretation of data in the era of precision astrophysics and cosmology. This is particularly true in the field of galaxy evolution and non-linear structure formation, where the requirements of modern surveys are such that only large dedicated campaigns of numerical simulations can reach the necessary precision and accuracy targets. Hence, it is no surprise that this field has seen a recent explosion in numerical tools, models, analysis methods and predictions (for reviews, see Somerville & Davé 2015;

Naab & Ostriker 2017; Vogelsberger et al. 2020; Angulo & Hahn 2022; Crain & van de Voort 2023).

Meeting this growing demand and complexity of numerical simulations requires increasingly efficient and robust tools to perform such calculations. For instance, these software involve more and more coupled differential equations to approximate, themselves coupled to increasingly complex networks of sub-grid models. At the same time, the evolution of computer architectures towards massively parallel systems further complicates the software development task. The details of the machine used, as well as an intimate knowledge of parallelization libraries, are often required to achieve anywhere near optimal on these the systems. This, however, often puts an additional burden on scientists attempting to make small alterations to the models they run and is often a barrier to the wider adoption of software packages. Nevertheless, the significant ecological impact of large astrophysical simulations (Portegies Zwart 2020; Stevens et al. 2020) make it imperative to address these technical challenges.

* E-mail: schaller@strw.leidenuniv.nl

Jointly, all these needs and sometimes orthogonal requirements make constructing such numerical software packages a daunting task. For these reasons, developing numerical software packages that are both efficient and sufficiently flexible has now become a task undertaken by large teams of contributors with mixed expertise, such as our own. This, in turn, implies that better code development practices need to be adopted to allow for collaborative work on large code bases.

Despite all this, the community has seen the arrival of a number of simulation software packages that rise to these challenges, many of which have also been released publicly. This recent trend, guided by open-science principles, is an important development allowing more scientists to run their own simulations, adapt them to their needs, and modify the code base to solve new problems. The public release of software is also an important step towards the reproducibility of results. Whilst some packages only offer the core solver freely to the community, some other collaborations have made the choice to fully release all their developments; we follow this latter choice here. This is an essential step that allows for more comparisons between models (as well as between models and data) to be performed and to help understand the advantages and shortcomings of the various methods used. The characterization and inclusion of uncertainty on model predictions, especially in the field of non-linear structure formation, is now becoming common practice (for examples targeted to the needs of large cosmology surveys see Heitmann et al. 2008; Schneider et al. 2016; Grove et al. 2022).

In this paper, we introduce the fully open-source code SWIFT.¹ designed to solve the coupled equations of gravity and hydrodynamics together with multiple networks of extensions specific to various sub-fields of astrophysics. The primary applications of the code are the evolution of cosmic large-scale structure, cluster and galaxy formation, and planetary physics. A selection of results obtained with the code is displayed in Fig. 1.

SWIFT was designed to be able to run the largest numerical problems of interest to the large-scale structure, cosmology & galaxy formation communities by exploiting modern algorithms and parallelization techniques to make efficient use of both existing and the latest CPU architectures. The scalability of the code was the core goal, alongside the flexibility to easily alter the physics modules. Our effort is, of course, not unique and there is now a variety of codes exploiting many different numerical algorithms and targeted at different problems in the ever-growing field of structure formation and galaxy evolution. Examples in regular use by the community include ART (Kravtsov, Klypin & Khokhlov 1997), FALCON (Dehnen 2000), FLASH (Fryxell et al. 2000), RAMSES (Teyssier 2002), GADGET-2 (Springel 2005), AREPO (Springel 2010b), GREEM (Ishiyama, Nitadori & Makino 2012), PLUTO (Mignone et al. 2012), CUBEP³M (Harnois-Déraps et al. 2013), 2HOT (Warren 2013), ENZO (Bryan et al. 2014), NYX (Almgren et al. 2013), CHANGA (Menon et al. 2015), GEVOLUTION (Adamek et al. 2016), HACC (Habib et al. 2016), GASOLINE-2 (Wadsley, Keller & Quinn 2017), PKDGRAV-3 (Potter, Stadel & Teyssier 2017), PHANTOM (Price et al. 2018), ATHENA++ (Stone et al. 2020), ABACUS (Garrison et al. 2021), and GADGET-4 (Springel et al. 2021) as well as many extensions and variations based on these solvers. They exploit a wide variety of numerical methods and are designed to target a broad range of astrophysics, galaxy formation, and cosmology problems.

Besides exploiting modern parallelization concepts, SWIFT makes use of state-of-the-art implementations of the key numerical methods.

The gravity solver relies on the algorithmically ideal fast-multipole method (see e.g. Greengard & Rokhlin 1987; Cheng, Greengard & Rokhlin 1999; Dehnen 2014) and is optionally coupled to a particle-mesh method using the Fourier-space representation of the gravity equations to model periodic boundary conditions (See Springel et al. (2021) for a detailed discussion of the advantages of this coupling over a pure tree approach). The hydrodynamics solver is based on the Smoothed Particle Hydrodynamics (SPH) method (see e.g. Springel 2010a; Price 2012) with multiple flavours from the literature implemented as well as our own version (SPHENIX; Borrow et al. 2022). The code is also being extended towards other unstructured hydrodynamics methods (such as moving mesh (see e.g. Springel 2010b; Vandenbroucke & De Rijcke 2016), renormalized mesh-free techniques or SPH-ALE (see e.g. Hopkins 2015), which will be released in the future. For cosmological applications, SWIFT was extended to use the particle-based ‘delta-f’ method of Elbers et al. (2021) to evolve massive neutrinos, allowing us to explore variations of the Λ CDM model. On top of these core components, the software package was extended to provide models for galaxy formation. We make two such models available: one based on that used for the EAGLE project (Crain et al. 2015; Schaye et al. 2015) and a second one based on the GEAR code (Revaz & Jablonka 2018; Hausammann 2021). These were designed to target very different scales and resolution ranges—massive galaxies and their large-scale environment for EAGLE, and dwarf galaxies for GEAR and are hence highly complementary. The EAGLE model is additionally and optionally extended with the implementation of jet feedback from active galactic nuclei by Huško et al. (2022).

Although SWIFT was originally developed for large-scale structure cosmology and galaxy formation applications, it quickly became clear that the benefits of the improved parallelization of the coupled gravity–hydrodynamics solver could also be extended to other areas in astrophysics. In particular, the code has been extended to support planetary simulations by adding equations of state for the relevant materials. These extensions have been designed by expanding the existing SPH schemes to allow for multiple materials to interact, hence opening the window to simulate the collisions and interactions of planets and other bodies made of various layers of different materials.

Another, and to our knowledge unique, feature of SWIFT is the extent of the material distributed as part of the public release.² We do not only distribute the core gravity and hydrodynamics solver but also offer the multiple modules for galaxy formation mentioned and other applications above, as well as the different flavours of SPH, the full treatment of cosmological neutrinos, and more than 100 ready-to-run example problems. All these elements are documented in detail, including developer instructions for extending the code. We emphasize too that the code is in active development and we expect future releases to further extend the physics modules presented here.

This paper is arranged as follows. In Section 2, we present the overall SWIFT code design philosophy and core principles. The equations of SPH that the code solves are summarized in Section 3. In Sections 4 and 5, we introduce the equations for gravity, neutrinos, and the cosmology framework used by the code. Sections 6 and 7 are dedicated to the input & output strategy and cosmological structure finding respectively. In Section 8, we present some extensions including galaxy formation (sub-grid) models and planetary physics models. We complete the code presentation in Section 9 with some implementation details and performance results.

¹SPH With Inter-dependent Fine-grained Tasking

²See www.swiftsim.com.

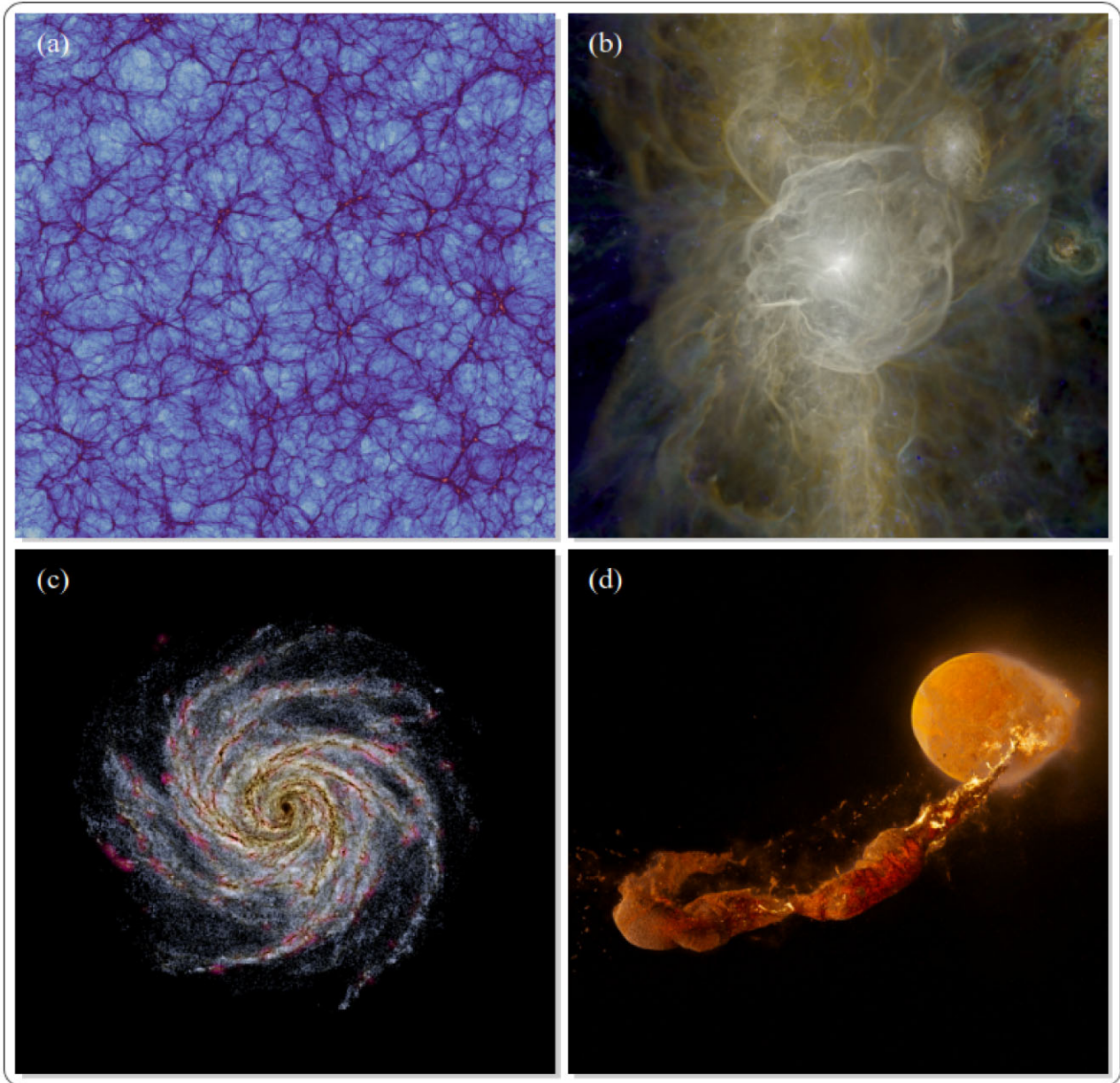


Figure 1. A selection of simulation results obtained with the SWIFT code, illustrating the huge range of problems that have already been targeted and the flexibility of the solver. The panels show: (a) a projection of the large-scale distribution of dark matter from a $10 \text{ Mpc } h^{-1}$ slice of the $(500 \text{ Mpc } h)^{-3}$ benchmark simulation of Schneider et al. (2016, Section 5.5); (b) the temperature of the gas weighted by its velocity dispersion in a zoom-in simulation of a galaxy cluster using the SWIFT-EAGLE galaxy formation model (Section 8.1) extracted from the runs of Altamura et al. (2023); (c) an idealized isolated galaxy from the Agora-suite (Kim et al. 2016) simulated using the GEAR model (Section 8.2) rendered using PNBODY (Revaz 2013); and (d) a snapshot extracted from a Moon-forming giant impact simulation of Kegerreis et al. (2022) using the planetary physics extension of the code (Section 8.5) and rendered using the HOUDINI software.

Finally, some conclusions are given and future plans are presented in Section 10.

2 CODE DESIGN AND IMPLEMENTATION CHOICES

We begin by laying out the core design principles of SWIFT, in particular its strategy for making efficient use of massively parallel (hybrid shared and distributed memory) high-performance computing systems.

2.1 The case for a hydrodynamics-first approach

Astrophysical codes solve complex networks of coupled differential equations, often acting on a large dynamic range of temporal and spatial scales. Over time, these pieces of software frequently evolve from their original baseline, through the addition of increasingly complex equations and physical processes, some of them treated as ‘sub-grid’ models. This process is often repeated multiple times with each new iteration of the code, leading to multiple layers of additions on top of one another. In many cases these layers do not use the most appropriate algorithms or parallelization strategies,

but rather rely on the decisions made for the previous layers' implementations.

A particularly relevant example of this issue is the generalized use of a tree-code infrastructure (e.g. Barnes & Hut 1986), originally designed to solve the equations of gravity, to also perform a neighbour-finding search for SPH (see e.g. Monaghan 1992; Price 2012, for a review). Similarly, this gas neighbour-finding code is then sometimes reused to find neighbours of star particles (for feedback or enrichment), although the two species are clustered very differently. These kinds of infrastructure re-use are ubiquitous in contemporary simulation codes (e.g. Hernquist & Katz 1989; Couchman, Thomas & Pearce 1995; Davé, Dubinski & Hernquist 1997; Springel, Yoshida & White 2001; Wadsley, Stadel & Quinn 2004; Springel 2005, 2010b; Hubber et al. 2011; Wadsley et al. 2017; Price et al. 2018; Springel et al. 2021). Although appealing for its reduced complexity, and successful in the past, this approach can in some cases result in noticeable sub-optimal computational efficiency, in particular for modern computing hardware. The data structure itself (a nested set of grids) are not the culprit here, the way it is traversed is the limitation. For example, tree walks typically involve frequent jumps in memory moving up and down the tree, a pattern that is not ideal for modern CPUs or GPUs. Such a pattern is particularly sub-optimal to make efficient use of the hierarchy of memory caches as most of the data read will be discarded. Instead, modern hardware prefers to access memory linearly and predictably, which also allows for a more efficient utilization of the memory bandwidth and caches, but also enables vector instructions (SIMD). To exploit vector instructions, we need all the elements of the vector (e.g. particles) to follow the same branching path. Thus, if an independent tree-walk has to be performed for each particle, and there is no obvious way to meaningfully group the particles into batches that will follow the same path in the tree, then it will seriously hinder our ability to use such vector instructions in our algorithms. Such an approach would hence, from the outset, forfeit $7/8^{th}$ ³ of the available computing performance of a modern system. The loss of performance due to a tree-walk's inability to make use of the various cache levels is more difficult to quantify. However, the recent trend in computing hardware to add more layers of caches is a clear sign that their use ought to be maximized in order to extract performance out of the computing units. To back up this intuition, we performed a detailed analysis of the large cosmological simulations from the EAGLE project (Schaye et al. 2015), based on a heavily modified version of the GADGET-3 code. It showed that the majority (> 65 per cent) of the computing time was spent in the neighbour-finding operations (both for gas and stars) performed via a tree walk.

All these considerations suggest that a simulation code designed with a hydrodynamics-first approach could achieve substantial performance gains. In SPH-like methods, the neighbourhood is defined entirely by demanding a certain number $N_{ngb} \sim 50-500$ of particles around the particle of interest from which to compute physical quantities and their derivatives. Similarly, many sub-grid implementations (see e.g. Sections 8.1, 8.2, and 8.3) rely on the same neighbourhoods for most of their calculations. Hence, grouping particles in cells that contain a number of particles $\gtrsim N_{ngb}$ will naturally construct neighbourhoods of the required size. This will lead to the construction of a Cartesian grid with cells whose size is similar to the size of the search radius of the particles. The neighbour-

³On a computer using AVX2 instructions (i.e. a SIMD vector size of 8), which is typical of current hardware. We note however that such peak performance is rarely achieved in actual production simulations.

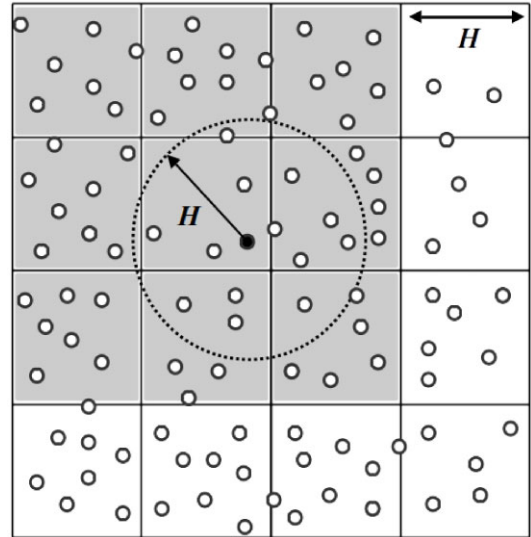


Figure 2. The Verlet-list method. By constructing a mesh structure with cell sizes matching the search radius H of particles, the neighbour-finding strategy is entirely set by the geometry of the cells and the list of potential candidates is thus exactly known. The particle in black only has potential neighbours in the cell where it resides or any of the 8 (26 in 3D) directly neighbouring cells (in grey). The smoothly varying nature of SPH leads to particles having similar H in nearby regions, with this scale only varying slowly over the whole simulated domain.

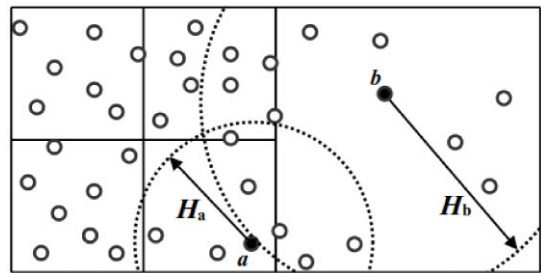


Figure 3. An example of interactions between regions of different densities, i.e. particles with different search radii. Particle a will interact with the particles on the left and above using the smaller cells. It will interact with the particles on the right using the larger cell. The particle b will only interact using the cells at the coarser level. Thanks to the nested grids, interactions happen at different levels in the hierarchy depending on the local search radius. Once the grid is constructed, all the possible interactions at the different levels are known without the need of a speculative tree-walk.

finding algorithm can then be greatly simplified. Each particle only needs to search for particles in the cell where it lies and any of the directly adjacent cells (Fig. 2). To ensure this property is always fulfilled, we force the cell sizes to not be smaller than the search radii of the particles in a given region. If the condition is violated, this triggers a reconstruction of the grid. This so-called *Verlet-list* method (Verlet 1967) is the standard way neighbour-finding is performed in molecular dynamics simulations. Once the cell structure has been constructed, all the required information is known. There is no need for any speculative tree-walk and the number of operations, as well as the iteration through memory, are easily predictable.

In the case of SPH for astrophysics, the picture is slightly more complex as the density of particles and hence the size of their neighbourhoods can vary by orders of magnitude. The method can nevertheless be adapted by employing a series of nested grids (Fig. 3).

Instead of constructing a single grid with a fixed cell size, we recursively divide them, which leads to a structure similar to the ones employed by adaptive-mesh-refinement codes (see Section 9.1). As we split the cells into eight children, this entire structure can also be interpreted as an oct-tree. We emphasize, however, that we do not walk up and down the tree to identify neighbours; this is a key difference with respect to other packages.

With the cells constructed, the entire SPH neighbour-related workload can then be decomposed into two sets of operations (or two sets of *tasks*): the interactions between all particles within a single cell and the interactions between all particles in directly adjacent cells. Each of these operations involves $\sim N_{\text{ngb}}^2$ particle operations. For typical scenarios, that is an amount of work that can easily be assigned to one single compute core with the required data fitting nicely in the associated memory cache. Furthermore, since the operations are straightforward (no tree-walk), one can make full use of vector instructions to parallelize the work at the lowest level.

This approach, borrowed from molecular dynamics, was adapted for multi-resolution SPH and evaluated by Gonnet (2015) and Schaller et al. (2016). It forms the basis of the SWIFT code described here. We emphasize that such an approach is not restricted to pure SPH methods; other mesh-free schemes, such as the arbitrary Lagrangian-Eulerian (ALE) renormalized mesh-free schemes (Vila 1999; Gaburov & Nitadori 2011; Hopkins 2015; Alonso Asensio et al. 2023), finite volume particle methods (e.g. Hietel et al. 2001, 2005; Ivanova et al. 2013), or moving mesh (Springel 2010b; Vandenbroucke & De Rijcke 2016) also naturally fit within this paradigm as they also rely on the concepts of neighbourhoods and localized interactions.

As it turns out, the series of nested grids constructed to accommodate the distribution of particles also forms the perfect structure on which to attach a gravity solver. We argued against such re-use at the start of our presentation; the situation here is, however, slightly different. Unlike what is done for the hydrodynamics, the gravity algorithm we use requires a tree-walk and some amount of pointer-chasing (jumps in memory) is thus unavoidable. We eliminated the tree-walk for the identification of SPH neighbourhoods, which was our original goal. We can now use a much more classic structure and algorithm for the gravity part of the SWIFT solver. Viewing the grid cells as tree nodes and leaves, we implement a *Fast-Multipole-Method* (FMM; see Greengard & Rokhlin 1987; Cheng et al. 1999; Dehnen 2002, 2014; Springel et al. 2021) algorithm to compute the gravitational interactions between particles. Here again, the work can be decomposed into interactions between particles in the same cell (tree-leaf), particles in neighbouring cells, or in distant cells. Once the tree is constructed, all the information is available and no new decision making is in principle necessary. The geometry of the tree and the choice of opening angle entirely characterizes all the operations that will need to be performed. All the arithmetic operations can then be streamlined with the particles treated in batches based on the tree-leaves they belong to.

2.2 Parallelization strategy: task-based parallelism

All modern computer architectures exploit multiple levels of parallelism. The trend over the last decade has been to increase the number of computing units (CPUs, GPUs, or other accelerators) in a single system rather than to speed up the calculations performed by each individual unit. Scientific codes that target modern high-performance computing systems must thus embrace and exploit this massive parallelism from the outset to get the most out of the underlying hardware.

As discussed in the previous section, the construction of a cell-based decomposition of the computational volume leads to natural units of work to be accomplished by the various compute cores. In principle, no ordering of these operations is required: as long as all the internal (*self* i.e. particle-particle interactions of particles within a single cell) and external (*pair* i.e. particle-particle interactions of particles residing in two different cells) interactions of these cells have been performed, all particles will have iterated over all their neighbours. One can therefore list all these cell-based units of work or *tasks* and use a piece of software that simply lets the different compute threads on a node fetch a task, execute it, and indicate its successful completion. Such tasks can e.g. take all the particles in a cell and compute the N_{cell}^2 SPH (or gravity) interactions between them; or take all the particles and drift them (i.e. integrate their positions) forward. This constitutes a very basic form of *task-based parallelism*. In astrophysics, the CHANGA code (Menon et al. 2015) uses a similar parallel framework.

Compared to the traditional ‘branch-and-bound’ approach in which all operations are carried out in a pre-specified order and where all compute units perform the same operation concurrently, as used by most other astrophysics simulation codes, this task-based approach has two major performance advantages. Firstly, it dynamically balances the work load over the available compute cores. In most simulations, the distribution of computational work over the simulation domain is highly inhomogeneous, with a small part of the volume typically dominating the total cost. Decomposing this work a priori (i.e. statically) is a very challenging problem, and practical solutions inevitably lead to substantial work imbalance. By not pre-assigning regions to a specific computing unit, the task scheduler can instead naturally and dynamically assign fewer cells to an individual computing unit if they turn out to have a high computational cost, and vice versa.

The second advantage of the task-based approach is that it naturally allows the gravity and hydrodynamics computations to be performed at the same time without the need for a global synchronization point between the two that typically leads to (sometimes substantial) idle time. The list of tasks simply contains both kinds of calculations and the threads can pick any of them; there is no need for the code to wait for all the gravity operations to be done before the SPH calculations can begin, or vice versa (Fig. 4).

This tasking approach forms the basis of SWIFT. In its form discussed above; however, it is too simple for the complex physics entering actual simulations. Most SPH implementations require multiple loops over the particles in their neighbourhoods. Sub-grid models often require that some hydrodynamic quantities be computed before they can themselves operate. One could first construct a list of all tasks related to the first loop and then distribute the threads on it. A second list could then be constructed of all the tasks related to the second loop and the process repeated. This would, however, re-introduce global synchronization points between the individual lists, leading to undesirable idle time. Instead, we construct a single list but introduce so-called *dependencies* between operations acting on a given cell (and hence its particles). For instance, all the first loop tasks have to be performed on a given cell before the tasks associated with the second loop can be performed. This transforms the list of tasks into an orientated *graph* with connections indicating the localized ordering of the physical operations to perform. This graph can now include all the operations, even the ones not requiring neighbour loops (e.g. time integration). Different cells can thus naturally progress in a given time step at different rates, leading to no global barriers between each loop (Fig. 5). When a task has completed, it reports this to all other tasks that depend on it. Once

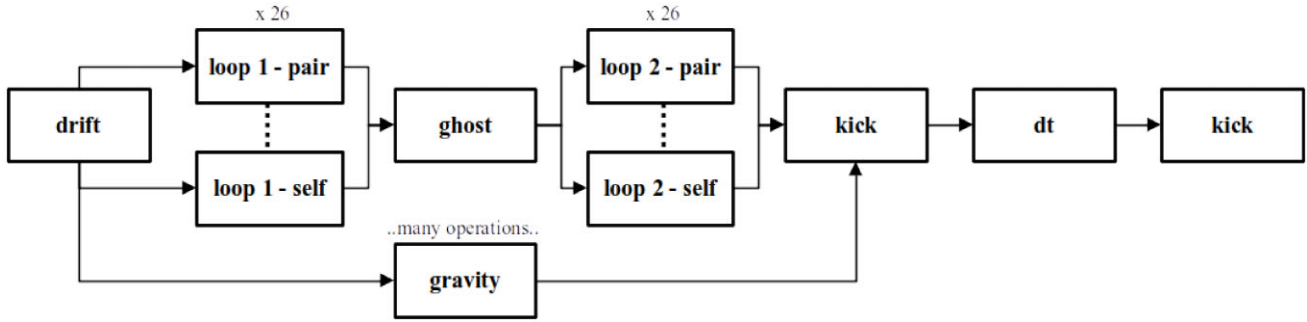


Figure 4. A simplified graph of the tasks acting on a given cell for SPH and gravity during one time step in SWIFT. Dependencies are depicted as arrows and conflicts by dotted lines. Once the particles have been drifted to the current point in time, the first loop over neighbours can be run. The so-called ‘ghost’ task serves mainly to reduce the number of dependencies between successive loops over the neighbours. Once the second loop has run, the time integration (Section 2.4) can be performed. In parallel to the SPH operations, the gravity tasks (condensed into a single one here for clarity) can be run as they act on different subsets of the data. To prevent different threads from overwriting each others’ data, the various SPH loop tasks (1 self and 26 pairs) are prevented from running concurrently via our conflict mechanism. Additional loops over neighbours, used for instance in more advanced SPH implementations, in sub-grid models or for radiative transfer, can be added by repeating the same pattern. They can also be placed after the time integration tasks if they correspond to terms entering the equations in an operator splitting way.

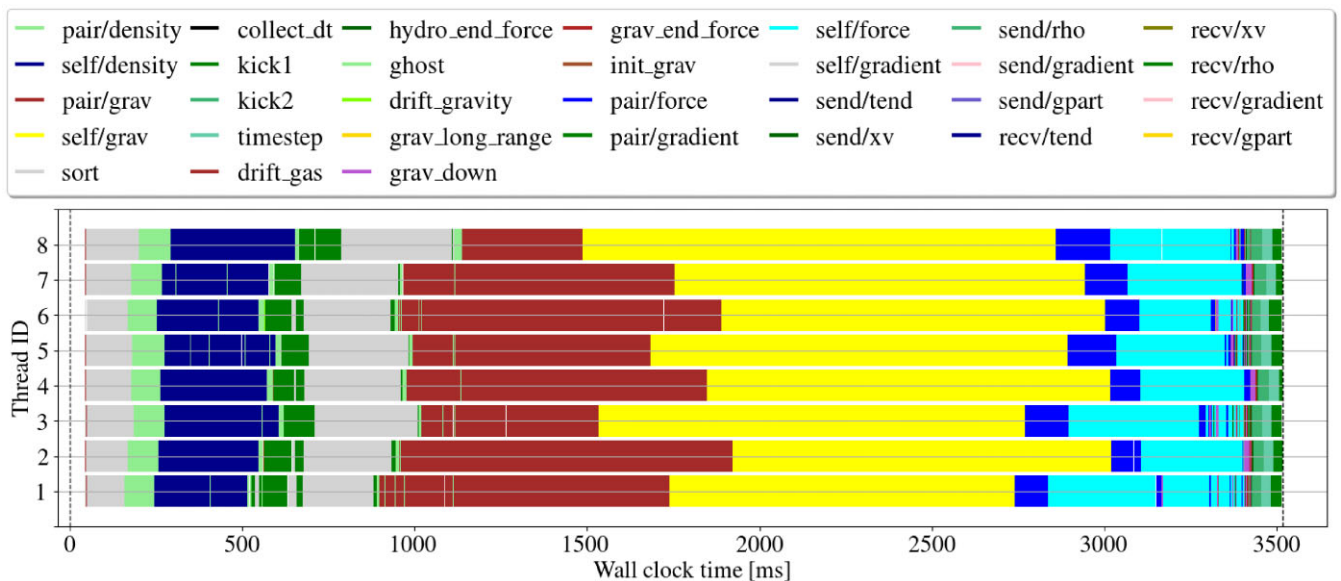


Figure 5. The execution of various tasks using 8 threads over the course of one time-step, extracted from a cosmological hydrodynamical simulation with 2×128^3 particles using only gravity and hydrodynamics on a shared-memory system. The different rows correspond to the different threads on the compute node. The work each thread performs is coloured to correspond to the task type it executes. Yellow, for instance, corresponds to a self-task performing gravity operation on a cell, whereas navy blue corresponds to a pair-task performing a 3rd SPH loop over two cells. Note that some tasks displayed in the legend do not actually run in this example. For instance, no MPI-related *send* or *recv* tasks are executed here. We show them in the legend for consistency with Fig. 9. The long bands are actually a series of the same task acting on different cells one after the others. There are for instance 512 yellow tasks. As desired, the threads display essentially no idle time (white gaps) between operations and all end their work at very nearly the same time. In other words, the load balancing is near-perfect with no parallel performance loss. The small gap at the start corresponds to cost of deciding what tasks to activate for this step. Bands of a given colour can have different lengths, indicating that tasks can correspond to very different workloads depending on how many particles are present in the cell(s) on which they act. At a given point in time, different threads often process different task types, and hence solve a different set of equations. This is different from the traditional branch-and-bound parallelism approach where all threads perform the same action and have to wait until they have all completed it before moving to the next piece of physics.

all dependencies for a task are satisfied (i.e. all the other tasks that must have run before it in the graph have completed), it is allowed to run; it is placed in a queue from where it can be fetched by available compute threads.

In addition to this mechanism, the task scheduling engine in the SWIFT code also uses the notion of *conflicts* (Fig. 4) to prevent two threads from working on the same cell at the same time. This eliminates the need to replicate data in different caches, which is

detrimental to performance. More crucially, it also ensures that all work performed inside a single task is intrinsically thread-safe without the need to use atomic operations. Because the code executed by a thread inside a task is guaranteed to run on a private piece of data, developers modifying the physics kernels need not worry about all the usual complexities related to parallel programming. This reduces the difficulty barrier inherent to programming on modern architectures and allows astrophysicists to easily modify and adapt

the physics model in SWIFT to their needs. To our knowledge, the combination of dependency and conflict management in the tasking engine is a unique feature of SWIFT.⁴ For a detailed description, we refer the reader to Gonnet, Chalk & Schaller (2016), where a stand-alone problem-agnostic version of this task scheduling engine is introduced.

One additional advantage of this conflict mechanism is the opportunity to symmetrize the operations. As no other compute thread is allowed to access the data within a cell, we can update *both* particles that take part in an interaction simultaneously, effectively halving the number of interactions to compute. This is typically not possible in a classic tree-walk scenario as each particle would need to independently search for its neighbours. The same optimization can be applied to the gravity interactions involving direct interactions of particles, usually between two tree leaves.

Last but not least, the thread-safe nature of the work performed by the tasks, combined with the small memory footprint of the data they act on, leads to them being naturally cache efficient but also prime candidates for SIMD optimization. The gravity calculations are simple enough that modern compilers are able to automatically generate vector instructions and thus parallelize the loops over pairs of particles. For instance, on the realistic gravity-only test problem of Section 4.6 we obtain speed-ups of 1.96x, 2.5x, and 3.14x on the entire calculation when switching on AVX, AVX2, and AVX512 auto-vectorization on top of regular optimization levels. This could also be the case for simple versions of the SPH loops (see discussion by Willis et al. 2018). The cut-off radius beyond which no interactions take place does, however, allow for additional optimizations. Borrowing, once more, from molecular dynamics, we implement sorted interactions and pseudo-Verlet lists (Gonnet 2013). Instead of considering all particles in neighbouring cells as potential candidates for interactions, we first sort them along the axis linking the cells' centres. By walking along this axis, we drastically reduce the number of checks on particles that are within neighbouring cells but outside each other's interaction range, especially in the cases where the cells only share an edge or a corner (Fig. 6). This way of iterating through the particle pairs is much more complex and compilers are currently unable to recognize the pattern and generate appropriate vector instructions. We therefore implemented SIMD code directly in SWIFT, for some of the flavours of SPH, following the method of Willis et al. (2018). This approach does, however, break down when more complex physics (such as galaxy formation models, see Section 8) are solved, as too many variables enter the equations.

Despite the advantages outlined above, one possible drawback to the task-based approach, as implemented in SWIFT, is the lack of determinism. The ordering in which the tasks are run will be different between different runs, even on the same hardware and with the exact same executable. This can (and does) lead to small differences in the rounding and truncation of floating point numbers throughout the code, which, in turn will lead to slightly different results each time. This is, of course, not an issue on its own as every single one of these results was obtained using the same combination of operations and within the same set of floating point rules. As an example, the study by Borrow et al. (2023) shows that the level of randomness created by the code is consistent with other studies varying random seeds to generate different galaxy populations. The same differences

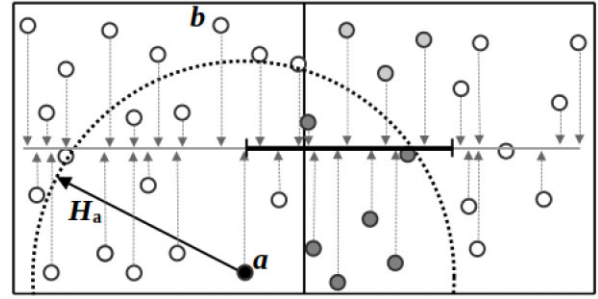


Figure 6. Pseudo-Verlet list optimization for the interactions between all particles within a pair of neighbouring cells. Here the particles in the left cell receive contributions from the particles in the right cell. In the first phase, all particles are projected onto the axis linking the two cells (grey line) and sorted based on their projected coordinates. In the interaction phase, the particles iterate along this axis to identify candidates. For instance, the particle *a* (in black) will identify plausible neighbours (in light and dark grey) on this axis up to a distance H_a (indicated by the black ruler). These candidates are then tested for 3D distance to verify whether they are genuine neighbours (i.e. within the dotted circle and highlighted in dark grey here) or not. With this technique, the number of false-positives (light grey) is greatly reduced compared to the total number of possible candidates in the right-hand cell (here, 3 vs. 11). The advantage is even greater when considering the next particle (from right to left) on the axis. Particle *b* knows that it will at most have to iterate on the axis up to the end of the ruler set by particle *a*, i.e. its list of candidates is at most as large as *a*'s for the same value of H . Moving from particle to particle in the left-hand cell, we can also stop the whole operation as soon as the distance on the axis does not reach at least the first particle in the right-hand cell. Because particles move only by small amounts between steps, the sorted list can be re-used multiple times provided a sufficient buffer is added to the length of the black ruler. Finally, the process is reversed to update the particles on the RHS with contributions from particles in the left cell. In 3D, even larger gains are achieved when the two cells share only an edge or just a corner.

between runs can also arise in pure MPI codes or when using other threading approaches such as OpenMP as neither of these guarantee the order of operations (at least in their default operating modes). Our approach merely exacerbates these differences. In practice, we find that the main drawback is the difficulty this intrinsic randomness can generate when debugging specific math-operation related problems. We note that nothing prevents us from altering the task scheduling engine to force a specific order. This would come at a performance cost, but could be implemented in a future iteration of the code to help with the aforementioned debugging scenario.

2.3 Beyond single-node systems

So far, we have described the parallelization strategy within single shared-memory compute nodes. To tackle actual high-performance computing (HPC) systems and run state-of-the-art calculations, mechanisms must be added to extend the computational domain to more than one node. The classic way to achieve this is to decompose the physical volume simulated into a set of discrete chunks and assign one to each compute node or even each compute thread. Communications, typically using an MPI implementation, must then be added to exchange information between these domains, or to perform reduction operations over all domains.

SWIFT exploits a variation of this approach, with two key guiding principles: first, MPI communication is only used between different compute nodes, rather than between individual cores of the same node (who use the previously described tasking mechanism to share

⁴The classical alternative to conflict management is to introduce explicit dependencies between tasks acting on the same data. This is less desirable as it introduces an ordering of the cells where no natural one exists.

work and data between each other). Secondly, we base the MPI domain decomposition on the same top-level grid structure as used for the neighbour finding, and aim to achieve a balanced distribution of work, rather than data, between nodes.

The base grid constructed for neighbour finding (Section 2.1) is split into regions that get assigned to individual compute nodes. The algorithm used to decide how to split the domain will be described in Section 9.3; we focus here on how the exchange of data is integrated into the task-based framework of SWIFT.

As the domain decomposition assigns entire cells to compute nodes, none of the tasks acting on a single cell require any changes; all their work is, by definition, purely local. We only need to consider operations involving pairs of particles, and hence pairs of cells, such as SPH loops, gravitational force calculation by direct summation (see Section 4.3), or sub-grid physics calculations (see Section 8).

Consider a particle needing information from a neighbour residing on another node to update its own fields. There are generally two possible approaches here. The first one is to send the particle over the network to the other node, perform a neighbour finding operation there, update the particle, and send the particle back to its original node. This may need to be repeated multiple times if the particle has neighbours on many different nodes. The second approach instead consists of importing all foreign neighbours to the node and then only updating the particles of interest local to the node once the foreign neighbour particle data is present. We use this second approach in SWIFT and construct a set of *proxy* cells to temporarily host the foreign particles needed for the interactions. The advantage of this approach is that it requires only a single communication, since no results have to be reported back to the node hosting the neighbour particle. Also, since we constructed the grid cells in such a way that we know a priori which particles can potentially be neighbours, and since we attach the communications to the cells directly, we also know which particles to communicate. We do not need to add any walk through a tree to identify which cells to communicate.

As SWIFT exploits threads within nodes and only uses MPI domains and communications between nodes, we actually construct relatively large domains when compared to other MPI-only software packages that must treat each core as a separate domain. This implies that each node's own particle (or cell) volume is typically much larger than any layer of proxy cells surrounding it. In typical applications, the memory overhead for import buffers of foreign particles is therefore relatively small. Furthermore, the trend of the last decade in computing hardware is to have an ever larger number of cores and memory on each node, which will increase the volume-to-surface ratio of each domain yet further. Note, however, that some of these trends are not followed by a proportional raise in memory bandwidth and some architectures also display complex NUMA designs. On such systems it may be beneficial to use a few MPI domains per node rather than a single one.

Once the proxy cells have been constructed, we create communication tasks to import their particles (see Fig. 7). When the import is done, the work within the pair task itself is identical to a purely local pair. Once again, users developing physics modules need therefore not be concerned with the complexities of parallel computing when writing their code.

The particles need to be communicated prior to the start of the pair interactions. After all, the correct up-to-date particle data needs to be present before the computation of the interactions for them to be correct. The commonly adopted strategy is to communicate all particles from each boundary region on all nodes to their corresponding proxy regions before the start of the calculations. This can be somewhat inefficient, for two reasons. First, it typically saturates the

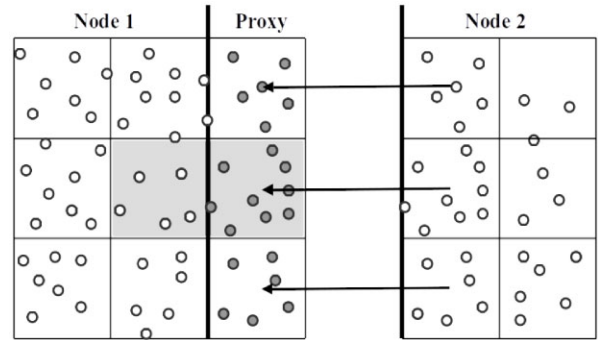


Figure 7. A pair interaction taking place over a domain boundary. The cell pair interaction in grey involves cells residing on either side of the domain boundary (thick black line), on two separate nodes. To allow for the interaction to happen, we create a set of proxy cells on the first node and create communication tasks (arrows) that import the relevant particles (in grey) from the second node. We also create a dependency between the communication and the pair task to ensure the data have arrived before the pair interaction can start. The pair task can then update the particles entirely locally, i.e. by exploiting exactly the same piece of code as for pairs that do not cross domain boundaries. A similar proxy exists on the other node to import particles in the opposite direction in order to process the pair also on that node and update its local particles.

communication network and the memory bandwidth of the system, leading to poor performance, especially on smaller, mid-range, computing facilities where the communication hardware is less powerful than in big national centres. Secondly, no other operations are performed by the code during this phase, even though particles far from any domain boundaries require no foreign neighbours at all and could therefore, in principle, have their interactions computed in the meantime. The traditional branch-and-bound approach prevents this, but SWIFT treats the communications themselves as tasks that can naturally be executed concurrently with other types of calculation (see above).

At a technical level, we achieve this concurrency by exploiting the concept of non-blocking communications offered by the MPI standard.⁵ This allows one compute node to mark some data to be sent and then return to process other work. The data are silently transferred in the background. On the receiving end, the same can be done and a receive operation can be posted before the execution returns to the main code. One can then probe the status of the communication itself, i.e. use the facilities offered by the MPI standard to know whether the data have arrived or are still in transit. By using such a probe, we can construct *send* and *receive* communication tasks that can then be inserted in the task graph where needed and behaving like any of the other (computing) tasks. Once the data have arrived on the receiving side, the receive task can simply unlock its dependencies and the work (pair tasks) that required the foreign data can now be executed (Fig. 8). By adding the communications in the tasking system, we essentially allow computational work to take place *at the same time* as communications. Note that the communication operations can be performed by any of the running threads. We do not reserve one thread for communications. The tasks not requiring foreign data can run as normal while the data for other pairs is being exchanged, eliminating the performance loss incurred from waiting for all exchanges to complete in the traditional approach. The large volume-to-surface ratio of our domains (see above) implies that there

⁵See section 3.7 of Message Passing Interface Forum (2021).

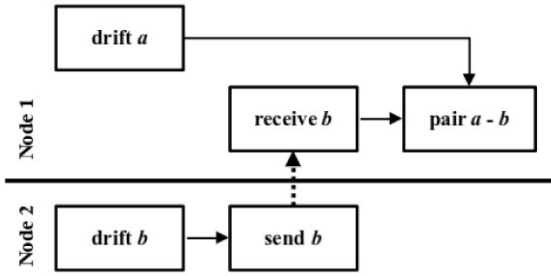


Figure 8. Extra communication tasks. The pair $a-b$ task (SPH or gravity) corresponds to the grey pair in Fig. 7. Each compute node has a task to drift its own local cell. The foreign node (here below the thick black line) then executes a *send* operation. On the local node, a *receive* task is run to get the data before unlocking the dependency (solid arrow) and letting the scheduler eventually run the pair $a-b$ interaction task. The communication itself (dotted arrow) implicitly acts as a dependency between the nodes. The converse set of tasks exists on the other compute node to allow the pair $b-a$ to also be run on that node.

are typically many more tasks that require no foreign data than ones that do. There is, hence, almost always enough work to perform during the communication time and overheads.

An example of task execution over multiple nodes is displayed on Fig. 9. This is running the same simulation as was shown on Fig. 5 but exploiting 4 nodes each using 8 threads. We show here the full hybrid distributed and shared memory capability of SWIFT. Here again, tasks of different kind are executed simultaneously by different threads. No large data exchange operation is performed at

the start of the step; the threads immediately start working on tasks involving purely local data whilst the data are being transferred. The work and communication are thus effectively overlapping. The four nodes complete their work at almost the same time and so do the threads within each node, hence showing near perfect utilization of the system and thus the ability to scale well.

The ability of SWIFT to perform computations concurrently with MPI communications reduces idle time, but the actual situation is somewhat more complex. In reality, the MPI library as well as the lower software layers interacting with the communication hardware also need to use CPU cycles to process the messages and perform the required copies in memory, so that a complete overlap of communications and computations is not feasible. This is often referred to as the MPI progression problem. Such wasted time can for instance be seen as blank gaps between tasks on Fig. 9. The extra cost incurred can vary dramatically between different implementations of the MPI protocol and depending on the exact hardware used. A similar bottleneck can occur when certain sub-grid models requiring many neighbour loops are used (e.g. Chaikin et al. 2023). These may generate many back-and-forth communications with only little work to be done concurrently.

We remark, however, that whilst the communications taking place during a time-step are all formally asynchronous, we still have a synchronization point at the end of a step where all the compute nodes have to wait. This is necessary as we need all nodes to agree what the next time-step size is for instance. This can be detrimental in the cases where the time-step hierarchies become very deep (see below) and when only a handful of particles require updates every step. A strategy akin to the one used by the DISPATCH code

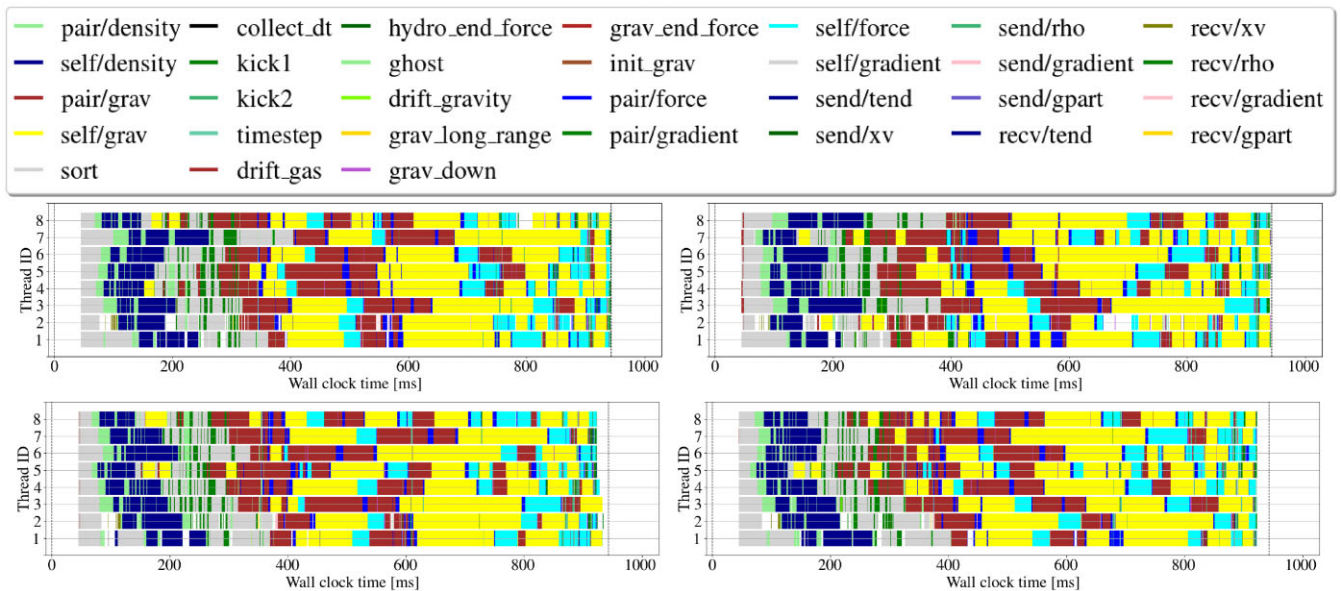


Figure 9. The same physics problem (2×128^3 particles cosmological simulation) as displayed on Fig. 5 but now split across 4 nodes, each using 8 threads, i.e. a combination of distributed and shared parallelism. This is the hybrid mode in which SWIFT is run for large calculations that do not fit on a single node. Each panel corresponds to a different compute node. Within each panel the different rows correspond to the different threads on the compute node. The work each thread performs is coloured to correspond to the task type it executes using the same scheme as on Fig. 5. The vertical dashed line on the right of each panel indicates the end of the time-step, which is determined by the point where the last compute node finishes. As can be seen, the node-to-node balance is not perfect; some nodes complete their work slightly earlier. This is due to the MPI library requiring some time to process messages in an unpredictable way, which the domain decomposition algorithm (Section 9.3) can thus not compensate for. This leads to small gaps in the execution (white gaps in the coloured bands). All required communication for the tasks occurs within this same figure, and overlaps (asynchronously) with work that only has local or already satisfied dependencies. All the exchanges happen whilst other tasks are running. The communications are overlapping with actual work. Note also that with less work per node overall compared to the shared-memory case, shown in Fig. 5, it is easier to see here that a given point in time different threads often process different task types, and hence solve a different set of equations.

(Ramsey, Haugbølle & Nordlund 2018), where regions can evolve at independent rates, would remove this last barrier. In practice, thanks to our domain decomposition aiming to balance the work not the data (see Section 9.3), this barrier is typically not a bottleneck for steps with a lot of work as the nodes all take a similar amount of time to reach this end-of-step barrier.

2.4 Local time-step optimizations

In most astrophysical simulations, not only do the length-scales of interest span several orders of magnitude, but so too do the time-scales. It would therefore typically, be prohibitively expensive to update all particles at every step; localized time-step sizes or even per-particle time-steps are essential. For a system governed by a Hamiltonian, it is possible to rewrite the classic leapfrog algorithm and consider *sub-cycles* where only a fraction of the particles receive acceleration updates (a kick operation) whilst all other particles are only moved (drifted) to the current point in time (Duncan, Levin & Lee 1998; Springel 2005). SWIFT exploits this mechanism by first creating long time-steps for the long-range gravity interaction (Section 4.5), where all the particles are updated, and then creating a hierarchy of smaller steps using powers-of-two subdivisions, where only the short-range gravity and hydrodynamic forces are updated (Hernquist & Katz 1989). This hierarchy is implemented by mapping the physical time from start to end of a simulation to the range of values representable by an integer. A jump of one thus represents the minimum time-step size reachable by a particle (e.g. $(t_{\text{end}} - t_{\text{begin}})/2^{32}$ for a 32-bit integer.). Each actual time-step size is then a power-of-two multiple of this base quantum of time, hence ensuring exactly the hierarchy of time-steps we expected. Using a 64-bit integer, we get a maximal possible number of steps in a run of $2^{64} \approx 10^{19}$, much more than will be necessary.

In real applications, this hierarchy can be more than 10 levels deep, meaning that the longest time-step sizes can be $> 1000 \times$ larger than the base time-step length (see e.g. Borrow et al. 2018).

The speed gains obtained by updating only a small fraction of the particles are immense. However, at the level of code implementation and parallelization, this concept creates complicated challenges. First, it requires added logic everywhere to decide which particle and hence which cell needs updating. This can be detrimental on some architectures (e.g. GPUs or SIMD vector units) where more streamlined operations are required. Secondly, and most importantly, it leads to global simulation steps where less computing time is spent moving the system forward than is spent in overheads. This challenge cannot simply be overcome by making the software more parallel; there will be steps where there are fewer particles to update than there are CPU threads running. As small steps (i.e. steps with a low number of particles to update) are orders of magnitude more frequent than the base step, they can actually dominate the overall simulation run time. It is hence of paramount importance to minimize all possible overheads.

One of the key overheads is the time spent communicating data across the network. The domain decomposition algorithm used in SWIFT (see Section 9.3) attempts to minimize this by not placing frequently active particles (or their cells) close to domain boundaries. If this is achieved, then entire steps can be performed without a single message being exchanged. The other main overhead is the drift operation. In the classic sub-cycling leapfrog (e.g. Quinn et al. 1997; Springel 2005), only the active particles are kicked, but *all* particles are drifted, since they could potentially be neighbours of the active ones. Whilst the drift is easily scalable, as it is a pure per-particle operation, it would nevertheless be wasteful to move

all particles for only the handful of them that are eventually found in the neighbourhood of the few active particles. In SWIFT, as is also done in some other modern codes, we alleviate this by first identifying the regions of the domain that contain active particles and all their neighbours. We then activate the drift task for these cells and only them. We thus do not drift all the particles just the required ones, which is, to our knowledge, not an approach that is discussed in the literature by other authors. This additional bit of logic to determine the regions of interest is similar to a single shallow tree-walk from the root of the tree down to the level where particles will be active. The benefit of this reduced drift operation is demonstrated by Borrow et al. (2018). We note that SWIFT can nevertheless be run in a more standard ‘drift-everything’ mode to allow for comparisons.

2.5 Language, implementation choices, and statistics

The design described above is, in principle, agnostic of the programming language used and of the precise libraries exploited⁶ to implement the physics or parallelism approach. It was decided early on to write the code in the C language (specifically using the GNU99 dialect) for its ease of use, wide range of available libraries, speed of compilation, and access to the low level threads, vector units, and memory management of the systems.

The task engine exploited by SWIFT is available as a stand-alone tool, QuickSched (Gonnet et al. 2016), and makes use of the standard POSIX threads available in all UNIX-based systems. The advantage of using our own library over other existing alternative (e.g. Cilk (Blumofe et al. 1995), TBB (Reinders 2007), SMPSS (Perez, Badia & Labarta 2008), StarPU (Augonnet et al. 2011), or the now standard OpenMP tasks) is that it is tailored to our specific needs and can be adapted to precisely match the code’s structure. We also require the use of task conflicts (see Section 2.2) and the ability to interface with MPI calls (see Section 2.3), two requirements not fulfilled by other alternatives when the project was started.

By relying on simple and widely available tools, SWIFT can be (and has been) run on a large variety of systems ranging from standard x86 CPUs, ARM-based computers, BlueGene architecture, and IBM Power microprocessors.

The entirety of the source code release here comprises more than 150 000 lines of code and 90 000 lines of comments. These large numbers are on the one hand due to the high verbosity of the C language and on the other hand due to the extent of the material released and the modular nature of the code. The majority of these lines are contained in the code extensions and i/o routines. Additionally, about 30 000 lines of python scripts are provided to generate and analyse examples. The basic COCOMO model (Boehm 2000) applied to our code base returns an estimate of 61 person-years for the development of the package.

SWIFT was also designed, from the beginning, with a focus on an open and well-documented architecture both for ease of use within the development team but also for the community at large. For that reason, we include fifteen thousand lines of narrative and theory documentation,⁷ a user onboarding guide, and large open-source, well-documented, and well-tested analysis tools.⁸

⁶With the exception of MPI, as its programming model drove many of the design decisions.

⁷Documentation is available at <http://www.swiftsim.com/docs>

⁸These tools are all available on the SWIFT project GitHub page <http://www.github.com/swiftsim>

3 SMOOTHED PARTICLE HYDRODYNAMICS SOLVER

Having discussed the mechanism used by SWIFT to perform loops over neighbouring particles, we now turn to the specific forms of the equations for hydrodynamics evolved in the code.

SPHs (Gingold & Monaghan 1977; Lucy 1977) has been prized for its adaptivity, simplicity, and Lagrangian nature. This makes it a natural fit for simulations of galaxy formation, with these simulations needing to capture huge dynamic ranges in density (over four orders of magnitude even for previous-generation simulations), and where the coupling to gravity solvers is crucial. Future releases of SWIFT will also offer more modern hydrodynamics solver options (see Section 10.2).

SWIFT implements a number of SPH solvers, all within the same neighbour-finding and time-stepping framework. These solvers range from a basic re-implementation of equations from Monaghan (1992) in Sections 3.1 and 3.2, to newer models including complex switches for artificial conductivity and viscosity. We introduce our default scheme SPHENIX in Section 3.3 and present our implementation of a time-step limiter and of particle splitting in Sections 3.4 and 3.5, respectively. For completeness, we give the equations for the additional flavours of SPH available in SWIFT in Appendix A. Note also that in this section, we limit ourselves to the equations of hydrodynamics in a non-expanding frame. Information on comoving time integration is presented later in Section 5.4.

As comparing hydrodynamic models is complex, and often a significant level of investigation is required even for a single test problem (e.g. Agertz et al. 2007; Braspenning et al. 2023), we do not directly compare the implemented models in SWIFT here. We limit our presentation to the classic ‘nIFTy cluster’ problem (Sembolini et al. 2016, Section 3.6), which is directly relevant to galaxy formation and cosmology applications. For our fiducial scheme, SPHENIX, the results of many of the standard hydrodynamics tests were presented by Borrow et al. (2022). The initial conditions and parameters for these tests, and many others, are distributed as part of SWIFT and can be run with all the schemes introduced below.

3.1 A brief introduction to SPH

SPH is frequently presented from two lenses: the first, a series of equations of motion derived from a Lagrangian with the constraint that the particles must obey the laws of thermodynamics (see e.g. Nelson & Papaloizou 1994; Monaghan & Price 2001; Springel & Hernquist 2002; Price 2012; Hopkins 2013); or a coarse-grained, interpolated, version of the Euler equations (as in Monaghan 1992).

As the implemented methods in SWIFT originate from numerous sources, there are SPH models originally derived from, and interpreted through, both of these lenses. Here, we place all of the equations of motion into a unified framework for easy comparison.

SPH, fundamentally, begins with the kernel.⁹ This kernel, which must be normalized, must have a central gradient of zero, and must be isotropic, is usually truncated at a compact support radius H . We describe the kernel as a function of radius r and smoothing length h , though all kernels implemented in SWIFT are primarily functions of the ratio between radius and smoothing length r/h to ensure that the function remains scale-free. The kernel function

$$W(r, h) = \frac{1}{h^{n_d}} w(r/h) \quad (1)$$

⁹An expanded discussion of the following is available in both Price (2012) and Borrow, Schaller & Bower (2021).

where here n_d is the number of spatial dimensions and $w(r/h)$ is a dimensionless function that describes the form of the kernel.

Throughout, SWIFT uses the Dehnen & Aly (2012) formalism, where the smoothing length of a particle is independent of the kernel used, with the smoothing length given by $h = \sqrt{2 \ln 2} a$, with a the full-width half maximum of a Gaussian. The cut-off radius $H = \gamma_K h$ is given through a kernel-dependent γ_K . We implement the kernels from that same paper, notably the Wendland (1995) C2, C4, and C6 kernels, as well as the Cubic, Quartic, and Quintic splines (Monaghan & Lattanzio 1985) using their normalization coefficients. Generally, we recommend that production simulations are performed with the Wendland-C2 or Quartic spline kernels for efficiency and accuracy reasons.

3.1.1 Constructing the number density and smoothing length

The kernel can allow us to construct smoothed, volume-dependent quantities from particle-carried quantities. Particle-carried quantities are intrinsic to individual mass elements (e.g. mass, thermal energy, and so on), whereas smoothed quantities (here denoted with a hat) are created from particle-carried quantities convolved with the kernel across the smoothing scale (e.g. mass density, thermal energy density, and so on).

The most basic smoothed quantity is referred to as the particle number density,

$$\hat{n}(\mathbf{r}, h) = \sum_j W(|\mathbf{r} - \mathbf{r}_j|, h), \quad (2)$$

for a sum runs over neighbouring particles j . This is effectively a partition of unity across the particle position domain when re-scaled such that

$$\hat{n}(h) \left(\frac{h}{\eta}\right)^{n_d} = 1, \quad (3)$$

for all positions \mathbf{r} and constant smoothing scale η ¹⁰, assuming that the smoothing length h is chosen to be large enough compared to the inter-particle separation.

Given a disordered particle arrangement (i.e. any arrangement with non-uniform particle spacing in all dimensions), it is possible to invert equation (3) with a fixed value of η to calculate the expected smoothing length given a measured number density from the current particle arrangement. In principle, this is possible for all values of η , but in practice there is a (kernel dependent, see Dehnen & Aly 2012) lower limit on η which gives acceptable sampling of the particle distribution (typically $\eta > 1.2$). Higher values of η give a smoother field, and can provide more accurate gradient estimates, but lead to an increase in computational cost. For some kernels, high values of η can also lead to occurrences of the pairing instability (Dehnen & Aly 2012; Price 2012).

Given a computation of \hat{n}_i at the position of a particle \mathbf{r}_i , for a given smoothing length h_i , an expected particle number density can be computed from equation (3). In addition, we compute the derivative

$$\frac{d\hat{n}_i}{dh} = - \sum_j \left(\frac{n_d}{h_i} W_{ij} + \frac{r_{ij}}{h_i} \nabla_i W_{ij} \right), \quad (4)$$

where here $r_{ij} \equiv |\mathbf{r}_i - \mathbf{r}_j|$, and $W_{ij} \equiv W(r_{ij}, h_i)$, with ∇_i implying a spatial derivative with respect to \mathbf{r}_i . This gradient is used, along with

¹⁰Relationships between the classic ‘number of neighbours’ definition and the smoothing scale η are described in Price (2012).

the difference between the expected density and measured density, within a Newton–Raphson scheme to ensure that the smoothing length h_i corresponds to equation (3) to within a relative factor of 10^{-4} by default.

We calculate the mass density of the system in a similar fashion, with this forming our fundamental interpolant:

$$\hat{\rho}_i = \sum_j m_j W_{ij}, \quad (5)$$

where here m_j is the particle mass. We choose to use the particle number density in the smoothing length calculation, rather than mass, to ensure adequate sampling for cases where particle masses may be very different, which was common in prior galaxy formation models due to stellar enrichment sub-grid implementations.

SWIFT calculates (for most implemented flavours of SPH) the pressure of particles based upon their smoothed density and their internal energy per unit mass u , or adiabat A , with

$$P_i = (\gamma - 1)u_i \hat{\rho}_i = A_i \hat{\rho}_i^\gamma, \quad (6)$$

where γ is the ratio of specific heats.

3.1.2 Creating general smoothed quantities

Beyond calculating the density, any quantity can be convolved with the kernel to calculate a smoothed quantity. For a general particle-carried quantity Q ,

$$\hat{Q}_i = \frac{1}{\hat{\rho}_i} \sum_j m_j \mathbf{Q}_j W_{ij}, \quad (7)$$

with spatial derivatives

$$\nabla \cdot \hat{Q}_i = \frac{1}{\hat{\rho}_i} \sum_j m_j \mathbf{Q}_j \cdot \nabla W_{ij}, \quad (8)$$

$$\nabla \times \hat{Q}_i = \frac{1}{\hat{\rho}_i} \sum_j m_j \mathbf{Q}_j \times \nabla W_{ij}, \quad (9)$$

provide basic estimates of smoothed quantities. Better estimators exist, and are used in specialized cases (see e.g. Price 2012), but in all other cases when we refer to a smoothed quantity these are the interpolants we rely on.

3.1.3 SPH equations of motion

Following Hopkins (2013), we write equations of motion for SPH in terms of two variables describing a volume element for conserving neighbour number (\bar{x} in their formalism, here we use a) and a volume element for the thermodynamical system (x in their formalism, here we use b). We then can write the conservative equations of motion for SPH as derived from a Lagrangian as follows:

$$\frac{d\mathbf{v}_i}{dt} = - \sum_j b_i b_j \left[\frac{f_{ij} P_i}{\hat{b}_i^2} \nabla_i W_{ij} + \frac{f_{ji} P_j}{\hat{b}_j^2} \nabla_j W_{ji} \right], \quad (10)$$

where here the factors f_{ij} are given by

$$f_{ij} = 1 - \frac{a_j}{b_j} \left(\frac{h_i}{n_d \hat{b}_i} \frac{\partial \hat{b}_i}{\partial h_i} \right) \left(1 + \frac{h_i}{n_d \hat{a}_i} \frac{\partial \hat{a}_i}{\partial h_i} \right)^{-1}. \quad (11)$$

The second equation of motion, i.e. the one evolving the thermodynamic variable (u or A) depends on the exact flavour of SPH, as described below.

3.2 Basic SPH flavours

SWIFT includes two so-called traditional SPH solvers, named MINIMAL (based on Price (2012)) and GADGET2 (based on Springel (2005)), which are Density–Energy and Density–Entropy-based solvers, respectively. This means that they use the particle mass as the variable b in equation (10) and evolve the internal energy u or, respectively the adiabat A (equation (6)), as thermodynamic variable. These two solvers use a basic prescription for artificial viscosity that is not explicitly time-varying. They are included in the code mainly for comparison to existing literature and to serve as basis for new developments.

These two solvers share the same equation of motion for velocity and internal energy,

$$\frac{d\mathbf{v}_i}{dt} = - \sum_j m_j \left[\frac{f_i P_i}{\hat{\rho}_i^2} \nabla_i W_{ij} + \frac{f_j P_j}{\hat{\rho}_j^2} \nabla_j W_{ij} \right], \quad (12)$$

$$\frac{du_i}{dt} = \sum_j m_j \frac{f_i P_i}{\hat{\rho}_i^2} \mathbf{v}_{ij} \cdot \nabla_i W_{ij} \quad (13)$$

but as they each track different thermodynamic variables (u , internal energy per unit mass for MINIMAL, and entropy/adiabat A for GADGET2). In this latter flavour, the equation for the adiabat is absent as $dA/dt = 0$ in the absence of additional source terms. In the equations above we also defined, $\mathbf{v}_{ij} \equiv \mathbf{v}_i - \mathbf{v}_j$, and

$$f_i = \left(1 + \frac{h_i}{n_d \hat{\rho}_i} \frac{\partial \hat{\rho}_i}{\partial h} \right), \quad (14)$$

which is known as the ‘f-factor’ or ‘h-factor’ to account for non-uniform smoothing lengths.

In addition to these conservative equations, the two basic SPH solvers include a simple viscosity prescription, implemented as an additional equation of motion for velocity and internal energy (entropy). The artificial viscosity implementation corresponds to the equations 101, 103, and 104 of Price (2012), with $\alpha_u = 0$ and $\beta = 3$. We solve the following equations of motion

$$\frac{d\mathbf{v}_i}{dt} \Big|_{\text{visc}} = - \sum_j m_j \frac{v_{ij}}{2} (f_i \nabla W_{ij} + f_j \nabla W_{ji}), \quad (15)$$

$$\frac{du_i}{dt} \Big|_{\text{visc}} = \sum_j m_j \frac{v_{ij}}{4} f_i \mathbf{v}_{ij} \cdot \nabla W_{ij}, \quad (16)$$

where the interaction-dependent factor

$$v_{ij} = - \frac{\alpha_{\mathbf{v},ij} \mu_{ij} v_{\text{sig},ij}}{\hat{\rho}_i \hat{\rho}_j}, \quad (17)$$

$$\mu_{ij} = \begin{cases} \frac{v_{ij} \cdot \mathbf{x}_{ij}}{|\mathbf{x}_{ij}|} & \text{if } \mathbf{v}_{ij} \cdot \mathbf{x}_{ij} < 0, \\ 0 & \text{otherwise.} \end{cases} \quad (18)$$

These rely on the signal velocity between all particles, which is also used in the time-step calculation, and is defined in these models as

$$v_{\text{sig},ij} = c_{s,i} + c_{s,j} - \beta \mu_{ij}, \quad (19)$$

where the constant $\beta = 3$.

Finally, the viscosity is modulated using the Balsara (1989) switch, which removes viscosity in shear flows. The switch is applied to the viscosity constants $\alpha_{\mathbf{v},ij}$ as follows:

$$\alpha_{\mathbf{v},ij} = \alpha_{\mathbf{v},i} = \alpha_{\mathbf{v}} B_i, \quad (20)$$

$$B_i = \frac{|\nabla \cdot \mathbf{v}_i|}{|\nabla \cdot \mathbf{v}_i| + |\nabla \times \mathbf{v}_i| + \epsilon c_{s,i}/h_i}, \quad (21)$$

where here $\alpha_{\mathbf{v}} = 0.8$ is a fixed constant, $c_{s,i}$ is the gas sound speed, and $\epsilon = 0.0001$ is a small dimensionless constant preventing divisions by zero.

3.3 The SPHENIX flavour of SPH

The SPHENIX flavour of SPH is the default flavour in SWIFT, and was described in detail by Borrow et al. (2022). SPHENIX inherits from the Density–Energy formulation of SPH, uses similar discontinuity treatments and limiters as the ANARCHY scheme use in the EAGLE cosmological simulations (see Schaller et al. 2015; Schaye et al. 2015, and Appendix A4), and uses a novel limiter for feedback events. SPHENIX was designed with galaxy formation applications in mind. As the scheme uses the Density–Energy equation of motion and not a pressure-smoothed implementation (Section A3), it must use a comparatively higher amount of conduction at contact discontinuities to avoid spurious pressure forces (e.g. Agertz et al. 2007; Price 2008, 2012). As such, removing the additional conduction in scenarios where it is not warranted (in particular strong shocks) becomes crucial for accurate modelling and to not dissipate energy where not desired.

As such, the major equations of motion are the same as described above in the tradition SPH case, with the dissipationless component being identical to equation (13). The artificial viscosity term, however, is more complex. We no longer use a constant α_V in equation (17). We follow the framework of Morris & Monaghan (1997) and turn it into a time-evolving particle-carried quantity. This scalar parameter is integrated forward in time using

$$\alpha_{V,i}(t + \Delta t) = \alpha_{V,i}(t) - \alpha_{V,loc,i} \exp\left(-\frac{\ell \cdot c_{s,i}}{H_i} \Delta t\right), \quad (22)$$

with $H_i = \gamma_K h_i$ the kernel cut-off radius, and where

$$\alpha_{V,loc,i} = \alpha_{V,max} \frac{S_i}{v_{sig,i}^2 + S_i}, \quad (23)$$

$$S_i = H_i^2 \cdot \max(0, -\nabla \cdot \mathbf{v}_i), \quad (24)$$

which ensures that $\alpha_{V,i}$ decays away from shocks. In these expressions, $\ell = 0.05$ is the viscosity decay length, and $\alpha_{V,max} = 2.0$ is the maximal value of the artificial viscosity parameter. The S_i term is a shock indicator (see Cullen & Dehnen 2010) which we use here to rapidly increase the viscosity in their vicinity. For this detector, we calculate the time differential of the velocity divergence using the value from the previous time-step,

$$\dot{\nabla} \cdot \mathbf{v}_i(t + \Delta t) = \frac{\nabla \cdot \mathbf{v}_i(t + \Delta t) - \nabla \cdot \mathbf{v}_i(t)}{\Delta t}. \quad (25)$$

Additionally, If $\alpha_{V,loc,i} > \alpha_{V,i}(t)$, then $\alpha_{V,i}(t + \Delta t)$ is set to $\alpha_{V,loc,i}$ to ensure a rapid increase in viscosity when a shock front approaches. The value of the parameter entering the usual viscosity term (equation (17)) is then

$$\alpha_{V,ij} = \frac{\alpha_{V,i} + \alpha_{V,j}}{2} \cdot \frac{B_i + B_j}{2}, \quad (26)$$

which exploits the Balsara (1989) switch so that we can rapidly shut down viscosity in shear flows. Note that, by construction, these terms ensure that the interaction remains fully symmetric.

In SPHENIX, we also implement a thermal conduction (also known as artificial diffusion) model following Price (2008), by adding an additional equation of motion for internal energy

$$\left. \frac{du_i}{dt} \right|_{diff} = \sum_j \alpha_{c,ij} v_{c,ij} m_j (u_i - u_j) \frac{f_{ij} \nabla_i W_{ij} + f_{ij} \nabla_j W_{ji}}{\rho_i + \rho_j}, \quad (27)$$

where here the new dimensionless parameter for the artificial conduction strength is constructed using a pressure weighting of the contribution of both interacting particles:

$$\alpha_{c,ij} = \frac{P_i \alpha_{c,i} + P_j \alpha_{c,j}}{P_i + P_j}. \quad (28)$$

with the $\alpha_{c,i}$ evolved on a particle-by-particle basis with a similar time dependency to the artificial viscosity parameter. The artificial conduction uses the Laplacian of internal energy as a source term, in an effort to remove nonlinear gradients of internal energy over the kernel width, with

$$\frac{d\alpha_{c,i}}{dt} = \beta_c H_i \frac{\nabla^2 u_i}{\sqrt{u_i}} - (\alpha_{c,i} - \alpha_{c,min}) \frac{v_{c,i}}{H_i}, \quad (29)$$

where here $\beta_c = 1$ is a dimensionless parameter, and $\alpha_{c,i,min} = 0$ is the minimal value of the artificial conduction coefficient. The artificial conduction parameter is bounded by a maximal value of $\alpha_{c,i,min} = 2$ in all cases. The value of β_c is high compared to other schemes to ensure the conduction parameter can vary on short timescales. Note that the velocity entering the last term of equation (29) is not the signal velocity but we instead follow Price et al. (2018) and write

$$v_{c,ij} = \frac{|\mathbf{v}_{ij} \cdot \mathbf{x}_{ij}|}{|\mathbf{x}_{ij}|} + \sqrt{2 \frac{|P_i - P_j|}{\hat{\rho}_j + \hat{\rho}_j}}. \quad (30)$$

This is a combination of the signal velocities used by Price et al. (2018) for the cases with and without gravity. As the thermal conduction term (equation (27)) is manifestly symmetric, no equation of motion for velocity is required to ensure energy conservation.

Finally, we ensure that the conduction is limited in regions undergoing strong shocks, limiting α_c by applying

$$\alpha_{c,max,i} = \alpha_{c,max} \left(1 - \frac{\alpha_{V,max,i}}{\alpha_{V,max}}\right), \quad (31)$$

with $\alpha_{c,max} = 1$ a constant, and

$$\alpha_{c,i} = \begin{cases} \alpha_{c,i} & \alpha_{c,i} < \alpha_{c,max} \\ \alpha_{c,max} & \alpha_{c,i} > \alpha_{c,max} \end{cases}. \quad (32)$$

Note the explicit appearance of the viscosity parameters $\alpha_{V,i}$ in these expressions. More information on the motivation behind the limiter, and its implementation, are presented by Borrow et al. (2022).

3.4 Time-step limiter

For all these schemes, a necessary condition to ensure energy conservation, especially when additional source terms such as stellar feedback are in use, is to impose some form of limit between the time-step size of neighbouring particles. This allows for information to be correctly propagated between particles (see Durier & Dalla Vecchia 2012). In SWIFT, we use three different mechanisms to achieve the desired outcome; these are all called ‘time-step limiters’ in different parts of the literature. We describe them here briefly.

The first limit we impose is to limit the time-step of *active* particles. When a particle computes the size of its next time-step, typically using the CFL condition, it also additionally considers the time-step size of all the particles it interacted within the loop computing accelerations. We then demand that the particle of interest’s time-step size is not larger than a factor Δ of the minimum of all the neighbours’ values. We typically use $\Delta = 4$ which fits naturally within the binary structure of the time-steps in the code. This first mechanism is always activated in SWIFT and does not require any additional loops or tasks; it is, however, not sufficient to ensure energy conservation in all cases.

The time-step limiter proposed by Saitoh & Makino (2009) is also implemented in SWIFT and is a recommended option for all simulations not using a fixed time-step size for all particles. This extends the simple mechanism described above by also considering inactive particles and waking them up if one of their active neighbours

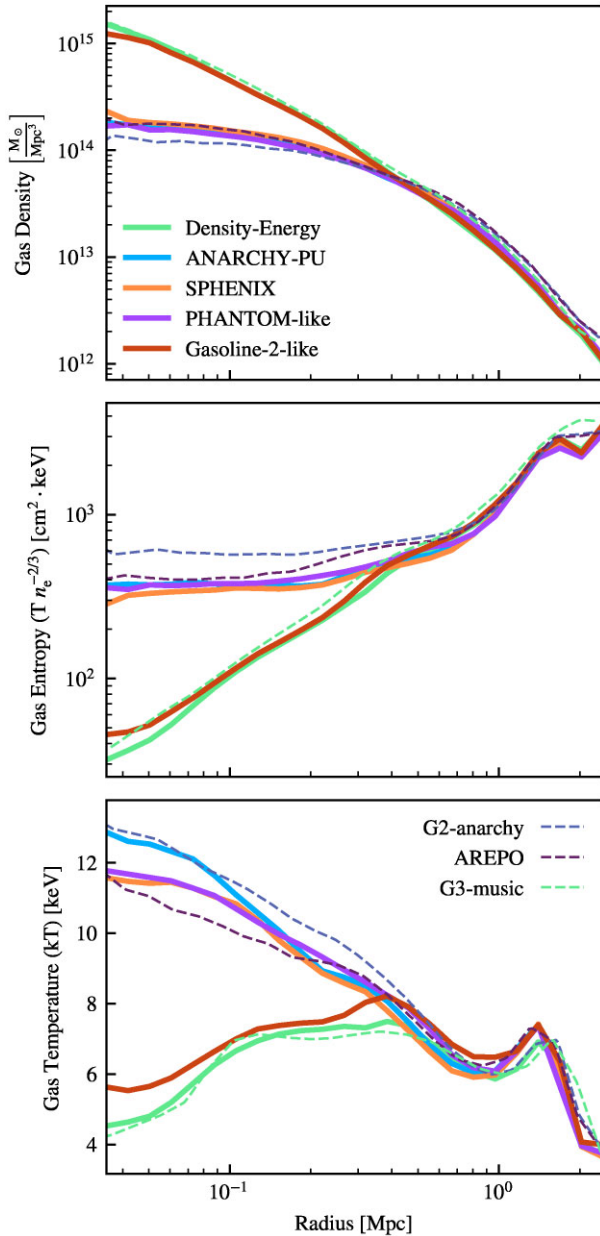


Figure 10. *Top panel:* The gas density profile of the nIFTy cluster when simulated with five models within SWIFT (thick solid lines of various colours), and three external codes (dashed thin lines), shown at redshift $z = 0$. *Middle panel:* Gas entropy profile of the cluster (as extracted from the temperature and electron density profiles). *Bottom panel:* Gas temperature profile of the cluster with the same models.

uses a much smaller time-step size. This is implemented by means of an additional loop over the neighbours at the end of the regular sequence (Fig. 4). Once an active particle has computed its time-step length for the next step, we perform an additional loop over its neighbours and activate any particles whose time-step length differs by more than a factor Δ (usually also set to 4). As shown by Saitoh & Makino (2009), this is necessary to conserve energy and hence yield the correct solution even in purely hydrodynamics problems such as a Sedov–Taylor blast wave. The additional loop over the neighbours is implemented by duplicating the already existing tasks and changing the content of the particle interactions to activate the requested neighbours.

The third mechanism we implement is a synchronization step to change the time-step of particles that have been directly affected by external source terms, typically feedback events. Durier & Dalla Vecchia (2012) showed that the Saitoh & Makino (2009) mechanism was not sufficient in scenarios where particles receive energy in the middle of their regular time-step. When particles are affected by feedback (see Sections 8.1, 8.2, and 8.3), we flag them for *synchronization*. A final pass over the particles, implemented as a task acting on any cell which was drifted to the current time, takes these flagged particles, interrupts their current step to terminate it at the current time and forces them back onto the timeline (Section 2.4) at the current step. They then recompute their time-step and get integrated forward in time as if they were on a short time-step all along. This guarantees a correct propagation of energy and hence an efficient implementation of feedback. The use of this mechanism is always recommended in simulations with external source terms.

3.5 Particle splitting

In some scenarios, particles can see their mass increase by large amounts. This is particularly the case in galaxy formation simulations, where some processes such as enrichment from stellar evolution (see Section 8.1.3) can increase some particle masses by large, sometimes unwanted, factors. To mitigate this problem, the SWIFT code can optionally be run with a mechanism to split particles that reach a specific mass. We note that this is a mere mitigation tool and should not be confused for a more comprehensive multiresolution algorithm where particle would adapt their masses dynamically in different regions of the simulation volume and/or based on refinement criteria.

When a particle reaches a user-defined mass m_{thresh} , we split the particle into two equal mass particles. The two particles are exact copies of each other but they are displaced in a random direction by a distance $0.2h$. All the relevant particle-carried properties are also halved in this process. One of the two particles then receives a new unique identifier.¹¹ To keep track of the particles’ history, we record the number of splits a particle has undergone over its lifetime and the ID of the original progenitor of the particle present in the initial conditions. Combined with a binary tree of all the splits, also stored in the particle, this leads to fully traceable, unique, identifier for every particle in the simulation volume.

3.6 The nIFTy cluster

In Fig. 10, we demonstrate the performance of a selection of the hydrodynamics solvers within SWIFT on the (non-radiative) nIFTy cluster (Sembolini et al. 2016) benchmark. The initial conditions used to perform this test are available for download as part of the SWIFT package in hdf5 format. All necessary data, like the parameter file required to run the test, is also provided in the repository as a ready-to-go example.

In the figure, we demonstrate the performance of five models from SWIFT [Density–Energy (Section 3.2) in green, ANARCHY-PU (Section A4) in blue, SPHENIX (Section 3.3) in orange, PHANTOM (Section A3) in purple, and GASOLINE-2 (Appendix A4) in red].¹²

¹¹ Depending on how the IDs are distributed in the initial conditions, we either generate a new random ID or append one to the maximal ID already present in the simulation.

¹² We remind the reader that all solvers are independent re-implementations within SWIFT rather than using their original codes, and all use the same neighbour-finding and time-step limiting procedures.

All simulations use the same Wendland-C2 kernel and $\eta = 1.2$. For comparison purposes, we display the results on this problem from the GADGET-2 flavour of ANARCHY (based upon Pressure-Entropy; G2-ANARCHY in dashed blue), the AREPO code and moving mesh-based solver (dashed purple), and a more standard SPH flavour implemented in GADGET-3 (G3-MUSIC). These additional curves were extracted from the original Sembolini et al. (2016) work.

Outside of radius $R > 0.5$ Mpc, all models show very similar behaviour. Internally to this radius, however, two classes of hydrodynamics model are revealed: those that form a flat entropy profile (i.e. the entropy tends towards very low values within the centre, driven by high densities and low temperatures), or a declining entropy profile (entropy flattens to a level of $k_B T n_e^{-2/3} \approx 10^{2.5} \text{ cm}^2 \text{ keV}$, driven by a low central density and high temperature). There has been much debate over the specific reasons for this difference between solvers. Here, we see that we form a flat profile with the GASOLINE-2-like (GDF) and Density-Energy models within SWIFT, and the G3-MUSIC code. These models have relatively low levels of diffusion or conduction (or none at all, in the case of Density-Energy and G3-MUSIC). For instance, within our GASOLINE-2-like implementation, we choose the standard value of the conduction parameter $C = 0.03$, consistent with the original implementation. Using a similar model Wadsley, Veeravalli & Couchman (2008) demonstrated that the formation of flat or declining entropy profiles was sensitive to the exact choice of this parameter (only forming flat profiles for $0.1 < C < 1.0$), and it is likely that this is the case within our SWIFT implementation too, though any such tuning and parameter exploration is out of the scope of this technical paper.

4 GRAVITY SOLVER

We now turn our attention towards the equations solved in SWIFT to account for self-gravity (see Dehnen & Read 2011; Angulo & Hahn 2022, for reviews). We start by introducing the gravity softening kernels (Section 4.1), then move on to summarize the Fast-Multipole-Method at the core of the algorithm (Section 4.2), and describe how it is implemented in our task-based framework (Section 4.3). We then present our choice of opening angle (Section 4.4) and the coupling of the method to a traditional Particle-Mesh algorithm (Section 4.5). We finish by showing a selection of test results (Section 4.6) before discussing how massive neutrinos are treated (Section 4.7).

4.1 Gravitational softening

To avoid artificial two-body relaxation and avoid singularities when particles get too close, the Dirac δ -distribution of the density field corresponding to each particle is convolved with a softening kernel of a given fixed, but possibly time-varying, scale-length H . Beyond H , a purely Newtonian regime is recovered.

Instead of the commonly used spline kernel of Monaghan & Lattanzio (1985) we use a C2 kernel (Wendland 1995), which leads to an expression for the force that is cheaper to compute whilst yielding a very similar overall shape. We modify the density field generated by a point-like particle $\delta(\mathbf{r}) = \rho(|\mathbf{r}|) = W(|\mathbf{r}|, 3\epsilon_{\text{Plummer}})$, where

$$W(r, H) = \frac{21}{2\pi H^3} \begin{cases} 4u^5 - 15u^4 + 20u^3 - 10u^2 + 1 & \text{if } u < 1, \\ 0 & \text{if } u \geq 1, \end{cases} \quad (33)$$

with $u = r/H$, and $\epsilon_{\text{Plummer}}$ is a free parameter linked to the resolution of the simulation (e.g. Power et al. 2003; Ludlow, Schaye & Bower 2019). The potential $\varphi(r, H)$ corresponding to this density distribution

reads

$$\varphi(r, H) = \begin{cases} f\left(\frac{r}{H}\right) \times H^{-1} & \text{if } r < H, \\ r^{-1} & \text{if } r \geq H, \end{cases} \quad (34)$$

with $f(u) \equiv -3u^7 + 15u^6 - 28u^5 + 21u^4 - 7u^2 + 3$. These choices lead to a potential at $|\mathbf{x}| = 0$ that is equal to the central potential of a Plummer (1911) sphere (i.e. $\varphi(r=0) = 1/\epsilon_{\text{Plummer}}$).¹³ From this expression the softened gravitational force can be easily obtained:

$$\nabla\varphi(r, H) = \mathbf{r} \cdot \begin{cases} g\left(\frac{r}{H}\right) \times H^{-3} & \text{if } r < H, \\ r^{-3} & \text{if } r \geq H, \end{cases} \quad (35)$$

with $g(u) \equiv f'(u)/u = -21u^5 + 90u^4 - 140u^3 + 84u^2 - 14$. This last expression has the advantage of not containing any divisions or branching (besides the always necessary check for $r < H$), making it faster to evaluate than the softened force derived from the Monaghan & Lattanzio (1985) spline kernel.¹⁴ It is hence well suited to target modern hardware, for instance to exploit SIMD instructions. In particular, the use of a C2 kernel here allows most of the commonly used compilers to automatically generate vectorized code, which is not the case when using a spline-based kernel with branches. On the realistic scenario used as a convergence test of Section 4.6, we get a speed-up of 2.5x when using AVX2 vectorization over the regularly optimized code.¹⁵ The same code using a spline kernel forfeits that speed-up and is even slightly slower due to the extra operations even in the non-vectorized case.

The softened density profile, with its corresponding potential and resulting forces¹⁶ are shown in Fig. 11. For comparison purposes, we also implemented the more traditional spline-kernel softening in SWIFT. For a recent discussion of the impact of different softening kernel shapes see section 8 of Hopkins et al. (2023).

4.2 Evaluating the forces using the Fast Multipole Method

The algorithmically challenging aspect of the N -body problem is to generate the potential and associated forces received by *each* particle in the system from *every other* particle in the system. Mathematically, this means evaluating

$$\phi(\mathbf{x}_a) = \sum_{b \neq a} G_N m_b \varphi(\mathbf{x}_a - \mathbf{x}_b) \quad \forall a \in N \quad (36)$$

efficiently for large numbers of particles N (with G_N the gravitational constant). In the case of collisionless dynamics, the particles are a mere Monte-Carlo sampling of the underlying coarse-grained phase-space distribution (e.g. Dehnen & Read 2011), which justifies the use of approximate methods to evaluate equation (36). The *Fast Multipole Method* (FMM Greengard & Rokhlin 1987; Cheng et al. 1999) is an $\mathcal{O}(N)$ approximation of equation (36), popularized in astronomy and adapted specifically for gravity solvers by Dehnen (2000, 2002) (see also Warren & Salmon (1995) for related ideas). The FMM works by expanding the potential in a Taylor series around *both* \mathbf{x}_a and \mathbf{x}_b and grouping similar terms arising from

¹³Note the factor of 3 in the definition of $\rho(|\mathbf{x}|)$ differs from the factor 2.8 used for the cubic spline kernel, as a consequence of the change of the functional form of W .

¹⁴A Plummer softening would also be branch-free but would have undesirable consequences on the dynamics (see e.g. Dehnen 2001).

¹⁵Note that switching off all optimization levels slows down the code by a factor 3.6x compared to the non-vectorized baseline.

¹⁶For more details about how these are constructed see Section 2 of Price & Monaghan (2007).

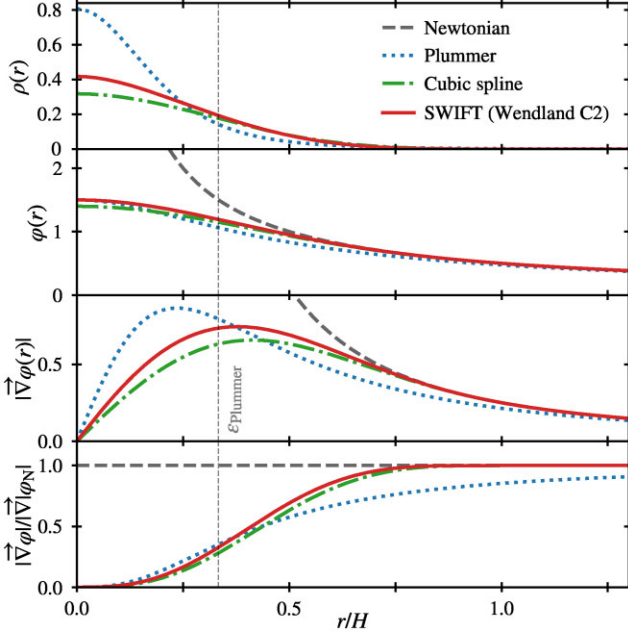


Figure 11. The density, potential, force, and force ratio to the Newtonian case generated by a point unit mass in our softened gravitational scheme. We use distances in units of the kernel cut-off H to normalize the figures. A Plummer-equivalent sphere is shown for comparison. The spline kernel of Monaghan & Lattanzio (1985) is depicted for comparison but note that it has not been normalized to match the Plummer-sphere potential at $r = 0$ (as is done in simulations) but rather normalized to the Newtonian potential at $r = H$ to better highlight the differences in shapes.

nearby particles to compute long-distance interactions between well-separated groups only once. In other words, we consider groups of particles with a large enough separation that the forces between them can be approximated well enough by just the forces between their centres of mass. Higher-order expressions, as used in SWIFT and other FMM codes, then not only approximate these groups as interacting point masses, but also take into account their shape, i.e. use the next order terms such as inertia tensors and beyond. A more rigorous derivation is given below.

The convergence of FMM and its applicability to a large range of gravity problems have been explored extensively (see e.g. Dehnen 2002, 2014; Potter et al. 2017; Garrison et al. 2021; Springel et al. 2021). For comparison, a Barnes & Hut (1986) tree-code, used in other modern codes such as 2HOT (Warren 2013) and GADGET-4 (Springel et al. 2021, in its default operating mode), only expands the potential around the sources \mathbf{x}_b . The formal complexity of such a method is $\mathcal{O}(N \log N)$.

4.2.1 Double expansion of the potential

In this section, we use the compact multi-index notation of Dehnen (2014) (repeated in appendix B for completeness) to simplify expressions and ease comparisons with other published work. In what follows \mathbf{k} , \mathbf{m} , and \mathbf{n} denote the multi-indices and \mathbf{r} , \mathbf{R} , \mathbf{x} , \mathbf{y} , and \mathbf{z} are vectors, whilst a and b denote particle indices. Note that no assumptions are made on the specific functional form of the potential φ .

For a single pair of particles a and b located in respective cells A and B with centres of mass \mathbf{z}_A and \mathbf{z}_B , as shown in Fig. 12, the

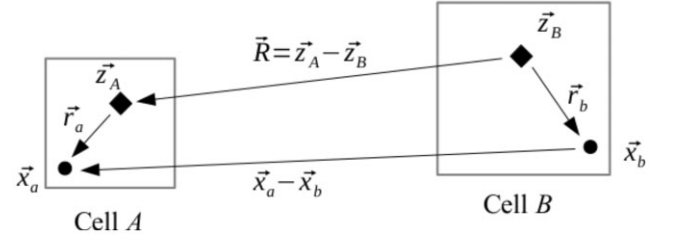


Figure 12. The basics of the Fast Multipole Method: The potential generated by a particle at position \mathbf{x}_b on a particle at location \mathbf{x}_a is replaced by a double Taylor expansion of the potential around the distance vector \mathbf{R} linking the two centres of mass (\mathbf{z}_A and \mathbf{z}_B) of cell A and B . The expansion converges towards the exact expression provided $|\mathbf{R}| > |\mathbf{r}_a + \mathbf{r}_b|$. In contrast, in a traditional Barnes & Hut (1986) tree-code, all the particles in the cell A receive direct contributions from \mathbf{z}_B without involving the centre of expansion \mathbf{z}_A in A .

potential generated by b at the location of a can be written as

$$\begin{aligned} \varphi(\mathbf{x}_a - \mathbf{x}_b) &= \varphi(\mathbf{x}_a - \mathbf{z}_A - \mathbf{x}_b + \mathbf{z}_B + \mathbf{z}_A - \mathbf{z}_B) \\ &= \varphi(\mathbf{r}_a - \mathbf{r}_b + \mathbf{R}) \\ &= \sum_{\mathbf{k}} \frac{1}{\mathbf{k}!} (\mathbf{r}_a - \mathbf{r}_b)^{\mathbf{k}} \nabla^{\mathbf{k}} \varphi(\mathbf{R}) \\ &= \sum_{\mathbf{k}} \frac{1}{\mathbf{k}!} \sum_{\mathbf{n} < \mathbf{k}} \binom{\mathbf{k}}{\mathbf{n}} \mathbf{r}_a^{\mathbf{n}} (-\mathbf{r}_b)^{\mathbf{k}-\mathbf{n}} \nabla^{\mathbf{k}} \varphi(\mathbf{R}) \\ &= \sum_{\mathbf{n}} \frac{1}{\mathbf{n}!} \mathbf{r}_a^{\mathbf{n}} \sum_{\mathbf{m}} \frac{1}{\mathbf{m}!} (-\mathbf{r}_b)^{\mathbf{m}} \nabla^{\mathbf{n}+\mathbf{m}} \varphi(\mathbf{R}), \end{aligned} \quad (37)$$

where the Taylor expansion of φ around $\mathbf{R} \equiv \mathbf{z}_A - \mathbf{z}_B$ was used on the third line, $\mathbf{r}_a \equiv \mathbf{x}_a - \mathbf{z}_A$, $\mathbf{r}_b \equiv \mathbf{x}_b - \mathbf{z}_B$ is defined throughout, and $\mathbf{m} \equiv \mathbf{k} - \mathbf{n}$ is defined for the last line. Expanding the series only up to order p , we get

$$\varphi(\mathbf{x}_a - \mathbf{x}_b) \approx \sum_{\mathbf{n}} \frac{1}{\mathbf{n}!} \mathbf{r}_a^{\mathbf{n}} \sum_{\mathbf{m}} \frac{1}{\mathbf{m}!} (-\mathbf{r}_b)^{\mathbf{m}} \nabla^{\mathbf{n}+\mathbf{m}} \varphi(\mathbf{R}), \quad (38)$$

with the approximation converging towards the correct value provided $|\mathbf{R}| > |\mathbf{r}_a + \mathbf{r}_b|$ as $p \rightarrow \infty$. If we now consider all the particles within B and combine their contributions to the potential at location \mathbf{x}_a in cell A , we get

$$\begin{aligned} \phi_{BA}(\mathbf{x}_a) &= \sum_{b \in B} G_N m_b \varphi(\mathbf{x}_a - \mathbf{x}_b) \\ &\approx G_N \sum_{\mathbf{n}} \frac{1}{\mathbf{n}!} \mathbf{r}_a^{\mathbf{n}} \sum_{\mathbf{m}} \frac{1}{\mathbf{m}!} \sum_{b \in B} m_b (-\mathbf{r}_b)^{\mathbf{m}} \nabla^{\mathbf{n}+\mathbf{m}} \varphi(\mathbf{R}). \end{aligned} \quad (39)$$

This last equation forms the basis of the FMM. The algorithm decomposes equation (36) into three separated sums, evaluated at different stages.

4.2.2 The FMM algorithm

As a first step, multipoles are constructed from the innermost sum. For each cell, we compute up to order p all the necessary multipoles (i.e. all terms M whose norm of the multi-index $\mathbf{m} \leq p$)

$$M_{\mathbf{m}}(\mathbf{z}_B) = \frac{1}{\mathbf{m}!} \sum_{b \in B} m_b (-\mathbf{r}_b)^{\mathbf{m}} = \sum_{b \in B} m_b X_{\mathbf{m}}(-\mathbf{r}_b), \quad (40)$$

where we re-used the tensors $X_{\mathbf{m}}(\mathbf{r}_b) \equiv \frac{1}{\mathbf{m}!} \mathbf{r}_b^{\mathbf{m}}$ to simplify the notation. This is the first kernel of the method, commonly labelled as P2M (particle to multipole). In a second step, we compute the second

kernel, M2L (multipole to local expansion), which corresponds to the interaction of a cell with another one:

$$F_n(\mathbf{z}_A) = G_N \sum_{\mathbf{m}}^{p-|\mathbf{m}|} M_{\mathbf{m}}(\mathbf{z}_B) D_{\mathbf{n}+\mathbf{m}}(\mathbf{R}), \quad (41)$$

where $D_{\mathbf{n}+\mathbf{m}}(\mathbf{R}) \equiv \nabla^{\mathbf{n}+\mathbf{m}}\varphi(\mathbf{R})$ is an order $n + m$ derivative of the potential. This is the computationally expensive step of the FMM algorithm, as the number of operations in a naive implementation using Cartesian coordinates scales as $\mathcal{O}(p^6)$. More advanced techniques (e.g. Dehnen 2014) can bring the cost down to $\mathcal{O}(p^3)$, albeit at a considerable algebraic cost. In the case of collisionless dynamics, accuracy down to machine precision for the forces is not required, and low values of p are thus sufficient, which maintains a reasonable computational cost for the M2L kernel (even in the Cartesian form).

Finally, the potential is propagated from the local expansion centre back to the particles (L2P kernel) using

$$\phi_{BA}(\mathbf{x}_a) = \sum_{\mathbf{n}}^p \frac{1}{\mathbf{n}!} \mathbf{r}_a^{\mathbf{n}} F_{\mathbf{n}}(\mathbf{z}_A) = \sum_{\mathbf{n}}^p X_{\mathbf{n}}(\mathbf{r}_a) F_{\mathbf{n}}(\mathbf{z}_A). \quad (42)$$

This expression is purely local, and can be efficiently implemented in a loop that updates all the particles in cell A .

In summary, the potential generated by a cell B on the particles in cell A is obtained by the successive application of the P2M, M2L, and L2P kernels. The P2M and L2P kernels need only be applied once per particle, whilst one M2L calculation must be performed for each pair of cells.

The forces applied to the particles are obtained by the same procedure, now using an extra order in the Taylor expansion. For instance, for the acceleration along the x -axis, we have:

$$a_x(\mathbf{x}_a) = \sum_{\mathbf{n}}^{p-1} X_{\mathbf{n}}(\mathbf{r}_a) F_{\mathbf{n}+(1,0,0)}(\mathbf{z}_A). \quad (43)$$

Higher-order terms, such as tidal tensors, can be constructed using the same logic. Note that only the last step in the process, the L2P kernel, needs to be modified for the accelerations or tidal tensors. The first two steps of the FMM, and in particular the expensive M2L phase, remain identical.

In practice, the multipoles can be constructed recursively from the leaves of the tree to the root, and the local expansions from the root to the leaves by shifting the M and F tensors and adding their contributions to their parent or child cell's tensors respectively. This can be done during the tree construction phase, for instance. Similarly, the local expansion tensors (F) can be propagated downwards using the opposite expressions.

While constructing the multipoles M , we also collect the centre of mass velocity of the particles in the cells. This allows us to drift the multipoles forward in time. This is only first-order accurate, but is sufficient in most circumstances, especially since once the particles have moved too much a full reconstruction of the tree (and hence of the multipoles) is triggered. Here, we follow the same logic as employed in many codes (e.g. GADGET Springel 2005) and force a tree reconstruction once a fixed cumulative fraction (typically 1 per cent) of the particles have received an update to their forces.

One final useful expression that enters some of the interactions between tree-leaves is the P2M kernel. This directly applies the potential due to a multipole expansion in cell B to a particle in cell A without using the expansion of the potential F at the centre of mass of cell A . This kernel is obtained by setting \mathbf{r}_a to zero in equation (37), re-defining $\mathbf{R} \equiv \mathbf{x}_a - \mathbf{z}_B$, and constructing the same M and D tensors as for the other kernels:

$$\phi_{Ba}(\mathbf{x}_a) = G \sum_{\mathbf{m}}^p M_{\mathbf{m}} D_{\mathbf{m}}(\mathbf{R}), \quad (44)$$

$$a_x(\mathbf{x}_a) = G \sum_{\mathbf{m}}^p M_{\mathbf{m}} D_{\mathbf{m}+(1,0,0)}(\mathbf{R}). \quad (45)$$

The P2M kernel acts identically to traditional Barnes & Hut (1986) tree-codes, which use solely that kernel to obtain the forces from the multipoles (or often just monopoles, i.e. setting $p = 0$ throughout) to the particles.

With all the kernels defined, we can construct a tree walk by recursively applying the M2L operation in a similar fashion to the double tree-walk introduced by Dehnen (2000).

4.2.3 Implementation choices

All the kernels (equations (40–45)) are rather straightforward to evaluate as they are only made of additions and multiplications (provided D can be evaluated quickly), which are extremely efficient instructions on modern architectures. However, the fully expanded sums can lead to rather large, and prone to typos, expressions. To avoid any mishaps, we use a python script to generate the C code in which all the sums are unrolled, ensuring they are correct by construction. This script is distributed as part of the code repository. In SWIFT, FMM kernels are implemented up to order $p = 5$, more than accurate enough for our purposes (see Section 4.6), but this could be extended to higher order easily. At order $p = 5$, this implies storing 56 numbers per cell for each M and F plus three numbers for the location of the centre of mass. Our default choice is to use multipoles up to order $p = 4$; higher or lower implementations can be chosen at compile time. For leaf-cells with large numbers of particles, as in SWIFT, this is a small memory overhead. One further small improvement consists in choosing \mathbf{z}_A to be the centre of mass of cell A rather than its geometrical centre. The first order multipoles ($M_{100}, M_{010}, M_{001}$) then vanish by construction. This allows us to simplify some of the expressions and helps reduce, albeit by a small fraction, the memory footprint of the tree structure.

4.3 The tree walk and task-parallel implementation

The three main kernels of the FMM methods (equations (40, 41, and 42)) are evaluated in different sections of the code. The construction of the multipoles is done during the tree building phase. This is performed outside of the task-based section of the code. As there is no need to handle dependencies or conflicts during the construction, we use a simple parallelization over the threads for this phase. As is done in other codes, this is achieved by recursively accumulating information from the tree leaves to the root level.

Once the tree and associated multipoles have been constructed, the remaining work to be performed is laid out. In a similar fashion to the hydrodynamics case (Section 2.2), all the calculations (M2L kernels and direct leaf-leaf interactions) can, in principle, be listed. The only difference lies in the definition of which cells need to interact using which kernel. This is based on the distance between the cells and information gathered from the multipoles (see Section 4.4 for the exact expression). In the case of a calculation using multiple nodes, the multipole information of neighbouring cells located on another node is exchanged after the tree construction (see Section 9.2). Whilst in the SPH case, the cells were constructed such that only direct neighbours need to be considered, one may, here, need to consider longer-range pairs of cells.

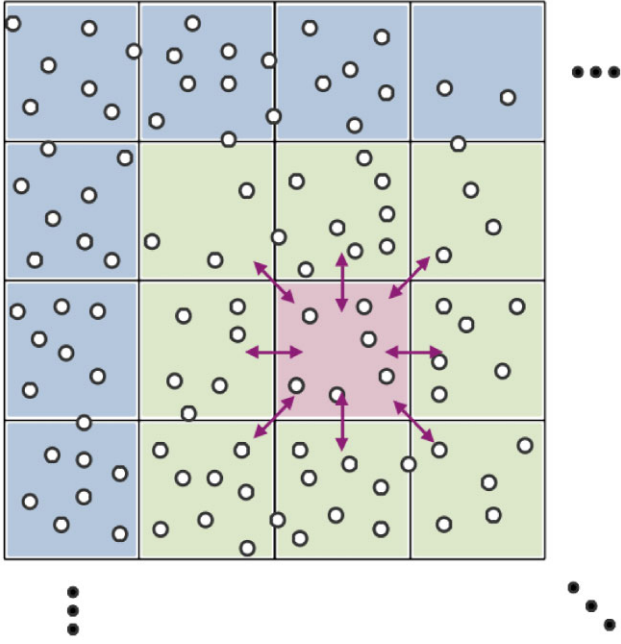


Figure 13. The basic decomposition of the FMM tree-walk into tasks for a set of particles in their cells, shown in 2D for clarity. The operations involving the red cell are as follows: (1) one *self* task computing the gravity kernels within the cell itself, (2) eight *pair* tasks computing the kernels for each pair of the red-green pairs of cells (the arrows), and (3) a single long-range task computing the M2L kernel contribution of all the blue cells to the red cell. In a realistic example, there will be many more blue cells beyond what is depicted here, but all their contributions to the cell of interest’s potential will be handled by a single task looping over all of them. The green cells are too close, based on the criterion of Section 4.4 to use a multipole-multipole (M2L) interaction; their interactions with the red cell are hence treated as individual tasks as they contain a substantial amount of calculation to perform. In some cases, the distance criterion may be such that cells slightly further away also need to be treated by the pair tasks rather than just the directly neighbouring layer. This depends on the exact particle configuration and on the user’s opening angle choices.

In practice, we start from the top-level grid of cells and identify all the pairs of cells that cannot interact via the M2L kernel. We then construct a *pair* task for each of them. Each cell also gets a *self* task which will take care of all the operations inside itself. Finally, for each cell, we create a *long-range* task, which will take care of all the interactions involving this cell and any cell far enough that the M2L kernel can be directly used. This third task is generally very cheap to evaluate as it involves only the evaluation of equation (41). This is illustrated on Fig. 13 for a simple case.

In most cases, the number of operations to perform within a single *self* or *pair* task is large. These cells are also very likely to be split into smaller cells in the tree. The tasks will hence attempt to recurse down the tree and perform the operations at the level that is most suitable. To this end, they use a double tree-walk logic akin to the one introduced by Dehnen (2002). At each level, we verify whether the children cells are far enough from each other based on the opening angle criterion (Section 4.4). If that is the case, then the M2L kernel is used. If not, then we move further down the tree and follow the same logic at the next level. The algorithm terminates when reaching a leaf cell. At this point, we either apply the M2P kernel, if allowed by the criterion, or default to the basic direct summation (P2P kernel) calculation.

Finally, the L2P kernel is applied on a cell-by-cell basis from the root to the leaves of the tree using a per-cell task. These tasks are only allowed to run once all of the self, pair, and long-range gravity tasks described above have run on the cell of interest. This is achieved using the dependency mechanism of the task scheduling library.

As the gravity calculation updates different particle fields (or even different particles) from the SPH tasks, we do not impose any dependency between the gravity and hydrodynamics operations. Both sets of tasks can run at the same time on the same cells and particles. This differs from other codes where an ordering is imposed. Our choice allows for better load-balancing since we do not need to wait for all the gravity operations (say) to complete before the hydrodynamics ones.

4.4 The multipole acceptance criterion

The main remaining question is to decide when two cells are far enough from each others that the truncated Taylor expansion used as approximation for the potential (equation (37)) is accurate enough. The criterion used to make that decision is called the *multipole acceptance criterion* (MAC).

We know that equation (37) converges towards the correct answer as p increases provided that $1 > |\mathbf{r}_a + \mathbf{r}_b|/|\mathbf{R}|$. This is hence the most basic (and always necessary) MAC that can be designed. If this ratio is lower, the accuracy (at a fixed expansion order) is improved and it is hence common practice to define a critical *opening angle* θ_{cr} and allow the use of the multipole approximation between two cells of size ρ_A and ρ_B if

$$\theta_{cr} > \frac{\rho_A + \rho_B}{|\mathbf{R}|}. \quad (46)$$

This lets users have a second handle on the accuracy on the gravity calculation besides the much more involved change in the expansion order p of the FMM method. Typical values for the opening angle are in the range $[0.3, 0.7]$, with the cost of the simulation growing as θ_{cr} decreases. Note that this MAC reduces to the original Barnes & Hut (1986) criterion when individual particles are considered (i.e. $\rho_A = 0$).

This method has the drawback of using a uniform criterion across the entire simulation volume and time evolution, which means that the chosen value of θ_{cr} could be too small in some regions (leading to too many operations for the expected accuracy) and too large in some other ones (leading to a lower level of accuracy than expected). SWIFT instead uses a more adaptive criterion to decide when the multipole approximation can be used. This is based on the error analysis of FMM by Dehnen (2014) and is summarized below for completeness.¹⁷ The key idea is to exploit the additional information about the distribution of particles that is encoded in the higher-order multipole terms.

We start by defining the scalar quantity $P_{A,n}$, the *power* of the multipole of order n of the particles in cell A , via

$$P_{A,n}^2 = \sum_{|\mathbf{m}|=n} \frac{\mathbf{m}!}{|\mathbf{m}|!} M_{A,\mathbf{m}}^2, \quad (47)$$

¹⁷See also Springel et al. (2001) for similar ideas in the regular tree case, based on the detailed error analysis of the tree code by Salmon & Warren (1994).

where the sum runs over all multipole terms of order n in the cell.¹⁸ This quantity is a simple upper bound for the amplitude of the multipole ($M_{A,m} < P_{A,|m|}/|\mathbf{m}|!$) and can hence be used to estimate the importance of the terms of a given order in the Taylor series of the potential. Following Dehnen (2014), we then consider a sink cell A and a source cell B (Fig. 12) for which we evaluate at order p the scalar

$$E_{BA,p} = \frac{1}{M_B |\mathbf{R}|^p} \sum_{n=0}^p \binom{p}{n} P_{B,n} \rho_A^{p-n}, \quad (48)$$

with $M_B \equiv M_{B,(0,0,0)}$, the sum of the mass of the particles in cell B . Note that since $P_{B,n} \leq M_B \rho_B^n$, we have $E_{BA,p} \leq ((\rho_A + \rho_B)/|\mathbf{R}|)^p$, where the right-hand side is the expression used in the basic opening angle condition (equation 46). We finally scale the $E_{BA,p}$'s by the relative size of the two cells to define the error estimator $\tilde{E}_{BA,p}$:

$$\tilde{E}_{BA,p} = 8 \frac{\max(\rho_A, \rho_B)}{\rho_A + \rho_B} E_{BA,p}. \quad (49)$$

As shown by Dehnen (2014), these quantities are excellent estimators of the error made in computing the accelerations between two cells using the M2L and M2P kernels at a given order. We can hence use this property to design a new MAC by demanding that the estimated acceleration error is no larger than a certain fraction of the smallest acceleration in the sink cell A . This means we can use the FMM approximation to obtain the accelerations in cell A due to the particles in cell B if

$$\tilde{E}_{BA,p} \frac{M_B}{|\mathbf{R}|^2} < \epsilon_{\text{FMM}} \min_{a \in A} (|\mathbf{a}_a|) \quad \text{and} \quad \frac{\rho_A + \rho_B}{|\mathbf{R}|} < 1, \quad (50)$$

where \mathbf{a}_a is the acceleration of the particles in cell A and ϵ_{FMM} is a tolerance parameter. Since this is self-referencing (i.e. we need the accelerations to decide how to compute the accelerations), we need to use an estimator of $|\mathbf{a}_a|$. In SWIFT, we follow the strategy commonly used in other software packages and use the acceleration of the previous time-step.¹⁹ The minimal norm of the acceleration in a given cell can be computed at the same time as the P2M kernels which are obtained in the tree construction phase. The second condition in equation 50 is necessary to ensure the convergence of the Taylor expansion.

One important difference between this criterion and the purely geometric one (equation (46)) is that it is not symmetric in $A \leftrightarrow B$ (i.e. $E_{AB,p} \neq E_{BA,p}$). This implies that there are cases where a multipole in cell A can be used to compute the field tensors in cell B but the multipole in B cannot be used to compute the F values of cell A and vice versa. This affects the tree walk by breaking the symmetry and potentially leading to cells of different sizes interacting. That is handled smoothly by the tasking mechanism which naturally adapts to the amount of work required. Note that an alternative approach would be to force the symmetry by allowing the multipoles to interact at a given level only if the criterion is satisfied in both directions. We additionally remark that this breaking of the symmetry formally leads to a breaking of the momentum-conserving property of the FMM method. We, however, do not regard this as an important issue as the momentum conservation is already broken by the use of per-particle time-step sizes.

¹⁸Note that $P_0 \equiv M_{(0,0,0)}$ is just the mass of the cell and since SWIFT uses the centre of mass as the centre of expansion of the multipoles, $P_1 = 0$.

¹⁹On the first time-step of a simulation this value has not been computed yet. We hence run a fake 'zereth' time-step with the simpler MAC (equation (46)), which is good enough to obtain approximations of the accelerations.

4.5 Coupling the FMM to a mesh for periodic long-range forces

To account for periodic boundary conditions in the gravity solver, the two main techniques present in the literature are: (1) apply an Ewald (1921)-type correction to every interaction (e.g. Hernquist & Katz 1989; Klessen 1997; Springel et al. 2001, 2021; Springel 2005; Hubber et al. 2011; Potter et al. 2017; Garrison et al. 2021); and (2) split the potential in two (or more) components with one of them solved for in Fourier space and thus accounting for the periodicity (e.g. Xu 1995; Bagla 2002; Springel 2005; Habib et al. 2016; Springel et al. 2021). We implement the latter of these two options in SWIFT and follow the same formalism as presented by Bagla & Ray (2003), adapted for FMM.

We start by truncating the potential and forces computed via the FMM using a smooth function that drops quickly to zero at some scale r_s set by the size of the gravity mesh. The Newtonian potential in equation (36) is effectively replaced by

$$\phi_s(r) = \frac{1}{r} \cdot \chi(r, r_s) \equiv \frac{1}{r} \cdot \text{erfc}\left(\frac{1}{2} \frac{r}{r_s}\right), \quad (51)$$

where the subscript s indicates that this is the short-range part of the potential. As $\chi(r, r_s)$ rapidly drops to negligible values, the potential and forces need only be computed via the tree walk for distances up to $r_{\text{cut}} = \beta r_s$; interactions at larger distances are considered to contribute exactly zero to the potential. Following Springel (2005), we use $\beta = 4.5$ as our default.²⁰ This maximal distance for tree interaction means that the long-range task (the one taking care of all the blue cells in Fig. 13) only needs to iterate over the cells up to a distance βr_s . This reduces further the amount of work to be performed for the long-range operations by the tree.

The long-range part of the potential ($\phi_l(r) = \frac{1}{r} \times \text{erf}\left(\frac{1}{2} \frac{r}{r_s}\right)$) is solved using a traditional particle-mesh (PM, see Hockney & Eastwood 1988) method. We assign all the particles onto a regular grid of N_{mesh}^3 cells using a cloud-in-cell (CIC) algorithm. The mesh also sets the cut-off size $r_s \equiv \alpha L/N_{\text{mesh}}$, where α is a dimensionless order-unity factor and L is the size-length of the simulation volume. We use $\alpha = 1.25$ as our default parameter value. In a second phase, we apply a Fourier transform to this density field using the Fast-Fourier-Transform (FFT) algorithm implemented in the `fftw` library (Frigo & Johnson 2005).

With the potential in Fourier space, Poisson's equation is solved by multiplying each cell's value by the transform of the long-range potential

$$\hat{\phi}_l(k) = -\frac{4\pi G_N}{|\mathbf{k}|^2} \cdot \exp(-|\mathbf{k}|^2 r_s^2). \quad (52)$$

We then deconvolve the CIC kernel twice (once for the assignment, once for the potential interpolation) and apply an inverse (fast) Fourier transform to recover the potential in real space on the mesh. Finally, the particles' individual potential and forces are obtained by interpolating from the mesh using the CIC method.

The functional form of equation (51) might, at first, appear sub-optimal. The error function is notoriously expensive to evaluate numerically. In our formulation, we must evaluate it for every pair of interactions (P2P or M2L) at every step. On the other hand, equation (52) needs to be evaluated only N_{mesh}^3 times at every global step (see below). Typically, $N_{\text{mesh}} \sim N^{1/3}$ but each of the N particles will perform many P2P kernel calls every single step. Using a simpler

²⁰At this distance, the suppression is almost three orders of magnitude already, as $\chi(4.5r_s, r_s) < 1.5 \times 10^{-3}$.

form for χ in real space with a more expensive one to evaluate correction in k -space may hence seem like an improvement. We experimented with sigmoid-like options such as

$$\chi(r, r_s) = \left[2 - 2\sigma\left(\frac{2r}{r_s}\right) \right], \quad \sigma(w) \equiv \frac{e^w}{1 + e^w} \quad (53)$$

but found little benefit overall. The solution we adopted instead is to stick with equation (51) and use an approximation to erfc sufficient for our needs. Specifically, we used equation 7.1.26 of Abramowitz & Stegun (1965). Over the range of interest, ($r \leq 4.5r_s$), this approximation has a relative error of less than 10^{-4} and the error tends to 0 as $r \rightarrow 0$. An alternative would be to store exact values in a table and interpolate between entries, but that approach has the disadvantage of requiring non-local memory accesses to this table shared between threads. Comparing simulations run with an exact erfc to simulations using the approximation above, we find no differences in the results.

Time integration of the forces arising from the long-range gravitational potential is performed using a long time step and the symplectic algorithm for sub-cycling of Duncan et al. (1998). We split the Hamiltonian in long and short timescales, corresponding to the long- and short-range gravity forces. The short-range Hamiltonian also contains the hydrodynamics forces. The time-steps then follow a sequence of kick & drift operators for the short-range forces embedded in-between two long-range kick operators (see also Quinn et al. 1997; Springel 2005; Springel et al. 2021).

As the mesh forces involve all particles and require all compute cores to perform the FFT together, we decided to implement the PM calculation (i.e. the CIC density interpolation, the calculation of the potential via Fourier space, and the interpolation of the accelerations back onto the particles) outside of the tasking system. In large calculations, the PM steps are rare (i.e. the long-range, global, time-step size is long compared to the smallest individual particle short-range time-step sizes). These steps are also where all particles will have to update their short-range forces, which will trigger a full tree rebuild. Having the PM calculation then perform a global operation outside of the tasking framework whilst locking all the threads is hence not an issue. To speed up operations, the PM calculation also uses parallel operations. The assignment of the particles onto the density grid is performed using a simple threading mechanism on each compute node. The Fourier transforms themselves are then performed using the `MPI + threads` version of the `fftw` library. All nodes and cores participate in the calculation. Once the potential grid has been obtained, the assignment of accelerations to the particles is done using the same basic per-node threading mechanism used for the construction of the density.

4.6 Convergence tests

The fast multipole method has been thoroughly tested both in the context of collisional dynamics and for collisionless applications (see e.g. Dehnen 2014; Springel et al. 2021). Many tests of simple scenarios, including cells with uniform particle distributions or isolated halos with different profiles can be found in the literature. As the behaviour of the method is well established and since our implementation does not differ from other reference codes besides the parallelization aspects, we do not repeat such a detailed study here. We report having successfully tested the FMM implementation in SWIFT on a wide range of cases, most of which are distributed as part of the examples in the code. We thus verified that the code converges towards the correct solution and presents the correct behaviour when the free parameters (e.g. the MAC or the gravity

mesh parameters) are varied. We report here on one such experiment with potential relevance to end users. Our test setup is a snapshot from a cosmological simulation of the EAGLE (Schaye et al. 2015) suite. We take the $z = 0.1$ snapshot from their $(25 \text{ Mpc})^3$ volume. This setup comprises $2 \times 376^3 \approx 10^7$ particles with a very high degree of clustering and is hence directly relevant to all galaxy formation applications of the code. The combination of haloes and voids present in the test allows us to test SWIFT's accuracy in a variety of regimes. We randomly select 1 per cent of the particles for which the exact forces are computed using a direct summation algorithm. An Ewald (1921) correction is applied to take into account the periodicity of the volume. We then run SWIFT and compute the forces via the FMM-PM code described above. We finally compute the relative force error for our sample of particles and evaluate the 99th percentile (f_{99}) of the error distribution. We chose to show the 99th percentile error over lower ones as it provides better guidance for users for their accuracy requirements by taking into account outliers. We show this error percentile as a function of the opening angle parameters in Fig. 14 for the case where periodic boundary conditions have been switched off. In this test, only the FMM part of the code is thus exercised. The left panel corresponds to the case of a purely geometric MAC (equation 46) and the right panel to the case of the adaptive MAC (equation 50). On both panels, we show different orders of the method using different line colours. The dotted line is used to indicate the 1 per cent-error level. We find that, as expected, the forces converge towards the correct, direct-summation-based, solution when the accuracy parameters are tightened. Similarly, when using the geometric MAC the relationship between f_{99} and θ_{cr} is found to be a power law whose slope steepens for higher values of p as predicted by theoretical arguments (e.g. Dehnen 2014; Springel et al. 2021). These expectations are displayed on the figure using thin dash-dotted lines. In the geometric case, the expected behaviour is recovered. The deviation from a power law at $\theta_{\text{cr}} < 0.3$ for $p = 5$ is taking place in the regime where the results start to be affected by single precision floating-point truncation. We have verified that when switching to double precision the power-law behaviour continues for smaller values of θ_{cr} , demonstrating that our implementation of the FMM algorithm matches theoretical expectations. In practice, this truncation error takes place much below the regime used in production runs. In the adaptive MAC case, the theoretical expectation is for the scheme to converge as $f_{99} \propto \epsilon_{\text{FMM}}$ for all orders p . This is shown as a thin black dash-dotted line on the figure. The current SWIFT implementation converges at a rate below these theoretical predictions. Our recommended default value for the adaptive MAC parameter is shown as a green arrow on the right panel. Using our default setup where we construct multipoles to fourth order, 99 per cent of the particles have a relative error of less than 5×10^{-3} for their force calculation. For comparison with the often used in the literature 90th percentile of the error (e.g. Springel et al. 2021), we additionally show it using a dashed line on the right panel for our default fourth-order FMM setup.

We repeat the same exercise but with periodic boundaries switched on and display the results in Fig. 15. The FMM part of the algorithm is unchanged, we only additionally add the PM part using a grid of 512^3 cells and a smoothing factor of $a_{\text{smooth}} = 1.25$ (our default value). In this case, the force error reaches a plateau for low values of the opening angle θ_{cr} or adaptive MAC parameter ϵ_{FMM} . This is where the algorithm reaches the accuracy limit of the PM part of the method. This is illustrated on the right panel by the dashed line which corresponds to the same run but with $a_{\text{smooth}} = 3$. In our default setup (fourth-order FMM, $\epsilon_{\text{FMM}} = 10^{-3}$, $a_{\text{smooth}} = 1.25$) indicated

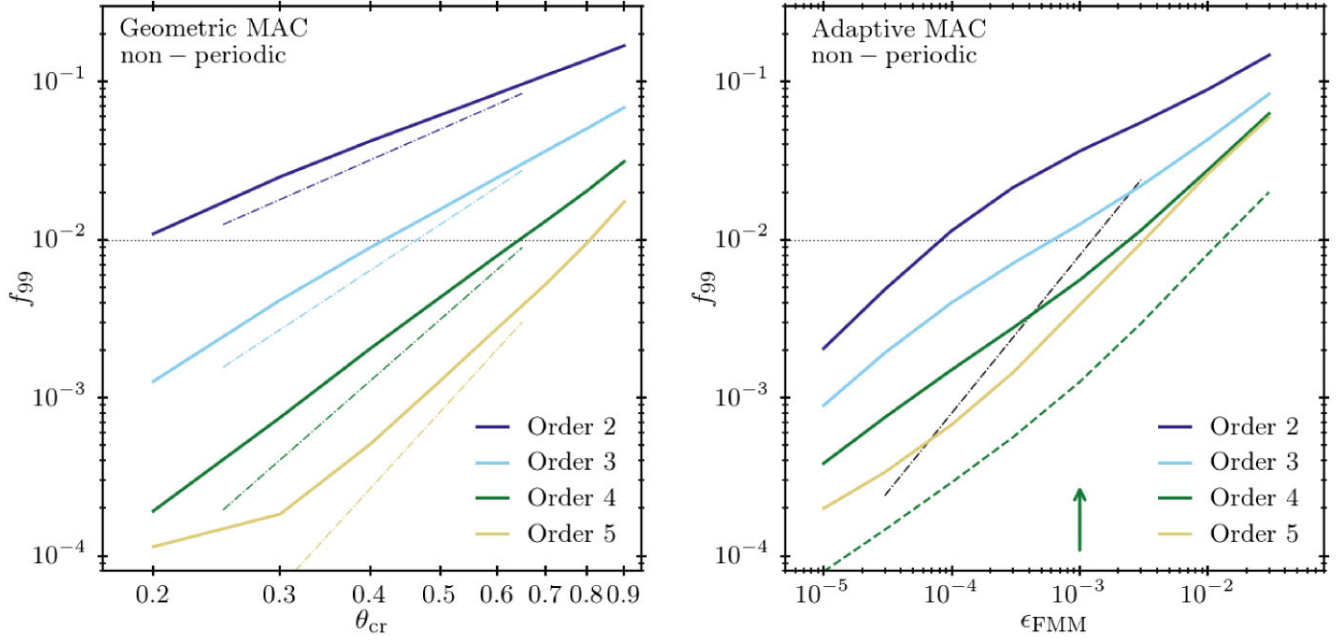


Figure 14. Accuracy of the gravity calculation (solid lines) for the two multipole acceptance criteria (MAC) on a low-redshift ($z = 0.1$) 2×376^3 particles, 25 Mpc cosmological hydrodynamical simulation extracted from the EAGLE suite. For 1 in every 100 particles, we calculated the exact forces using direct summation for comparison with the FMM-obtained prediction. We switch off periodic boundary conditions, and hence the gravity mesh, for this test. The 99th percentile of the relative force error distribution is plotted against the geometric MAC, the classic tree opening angle, on the left, and against the adaptive MAC parameter on the right. Various multipole calculation orders p are shown using different colours. Theoretical predictions for the convergence rates ($f_{99} \propto \theta^p$ for the geometric and $f_{99} \propto \epsilon_{\text{FMM}}$ for the adaptive case at all orders) are shown using thin dot-dashed lines in the background (only one line for the adaptive case as the predictions is independent of p). The horizontal dotted line indicates where 99 per cent of the particles achieve a relative accuracy of better than 1 per cent, a commonly adopted accuracy target. Our default MAC choice, indicated by an arrow on the right panel, corresponds to a 99th percentile of the relative error of 5×10^{-3} for our standard setup using the 4th order FMM implementation. We additionally show the 90th percentile of the error (f_{90}) for the order four adaptive MAC case using a dashed line. The SWIFT implementation converges at a lower rate than theoretical expectations in the adaptive case. In the geometric case, the deviation from the theoretically expected power-law behaviour for $\theta_{\text{cr}} < 0.3$ and $p = 5$ is due to truncation errors in single precision.

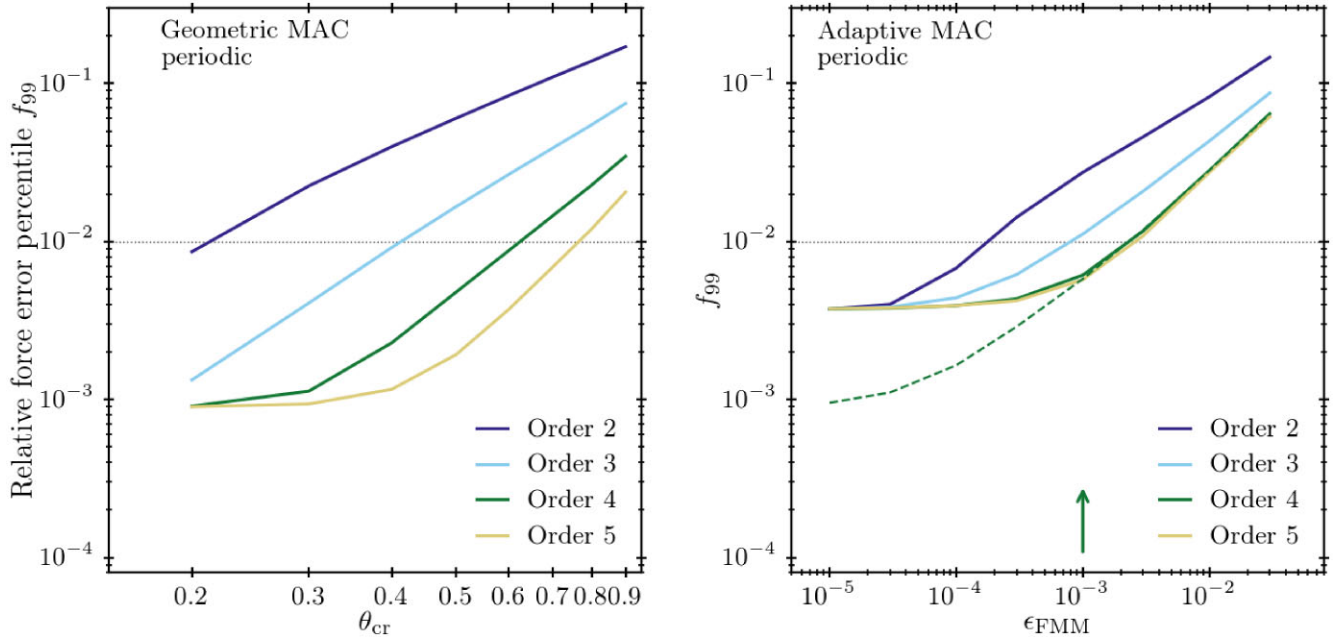


Figure 15. The same as Fig. 14, but now considering periodic boundary conditions. A gravity mesh of size $N_{\text{mesh}} = 512$ with $a_{\text{smooth}} = 1.25$ was used. The 99th percentile of the relative error rapidly reaches a plateau set by the accuracy of the force calculations computed by the PM part of the algorithm. The dashed line on the right panel corresponds to the order four scheme but using $a_{\text{smooth}} = 3$, illustrating the effect of the mesh parameters on the calculation's accuracy. For our default setup (green arrow), the scheme reaches a relative force accuracy of better than 6×10^{-3} for 99 per cent of the particles, a level only reached with very small opening angle values in the geometric case.

by the green arrow, 99 per cent of the particles have a relative force accuracy of better than 6×10^{-3} .

4.7 Treatment of massive neutrinos

Accurately modelling neutrinos is of great interest for large-scale structure simulations, due to their outsized effect on matter clustering (see Lesgourgues & Pastor 2006 for a review). We implemented two schemes for the treatment of neutrino effects in SWIFT: one based on the linear response method (Ali-Haïmoud & Bird 2013) and another based on the δf method (Elbers et al. 2021). In terms of the total matter power spectrum they produce, the two schemes are in good agreement.

The linear response method is a grid-based approach that accounts for the presence of neutrino perturbations by applying a linear correction factor in Fourier space to the long-range gravitational potential:

$$\hat{\phi}_l(\mathbf{k}) = \hat{\phi}_{l,\text{cb}}(\mathbf{k}) \cdot \left[1 + \frac{f_\nu}{f_{\text{cb}}} \frac{\delta_\nu^{\text{lin}}(k)}{\delta_{\text{cb}}^{\text{lin}}(k)} \right], \quad (54)$$

where $\hat{\phi}_{l,\text{cb}}$ is the long-range gravitational potential computed from the cold dark matter and baryon particles (Section 4.5). The correction factor depends on the ratio of linear theory transfer functions (δ) for neutrinos and cold dark matter plus baryons, as well as their relative mass fractions (f).

The second scheme, based on the δf method, actively solves for the neutrino perturbations. It is a hybrid approach that combines a particle-based Monte Carlo sampling of the neutrino phase-space distribution with an analytical background solution. The aim is to solve for the nonlinear gravitational evolution of the neutrinos, while suppressing the shot noise that plagues traditional particle implementations. In this method, the non-linear phase-space density f of neutrinos is decomposed as

$$f(\mathbf{x}, \mathbf{p}, t) = \bar{f}(p, t) + \delta f(\mathbf{x}, \mathbf{p}, t), \quad (55)$$

where $\bar{f}(p, t) = (1 + \exp(p/k_B T_\nu))^{-1}$ is the background Fermi–Dirac distribution (expressed in terms of the neutrino temperature T_ν) and δf is a possibly non-linear perturbation. In contrast to traditional, pure particle, implementations, only δf is estimated from the particles hence reducing the shot noise. To achieve this decomposition, the contribution of neutrino particles to the mass density is statistically weighted. The weight of particle i is given by

$$w_i = \frac{\delta f_i}{f_i} = \frac{f_i - \bar{f}_i}{f_i}, \quad (56)$$

where f_i is the phase-space density at its location. Weights express the deviation from the background, they can be positive or negative, and are ideally small. The reduction in shot noise is proportional to $\langle w^2 \rangle$ for the neutrino power spectrum. The weights must be updated on the fly, which involves a single loop over neutrino particles. We make use of the fact that \bar{f}_i depends only on the current particle momentum, while the value of f_i is conserved. To avoid storing f_i , SWIFT uses the particle ID as a deterministic pseudo-random seed to sample the initial Fermi–Dirac momentum. The value of f_i is then recomputed when needed. As a result, the memory footprint of neutrinos is identical to that of cold dark matter particles. The neutrino particles then enter the gravity calculation identically to all the other species but see their mass multiplied by their weight.

The possibility of negatively weighted particles requires some attention. In exceptional circumstances, which nevertheless occur for simulations involving billions of particles and thousands of steps, the centre of mass of a group of neutrinos can lie far beyond the geometric

perimeter of the particles. Since SWIFT uses a multipole expansion around the centre of mass, this possibility causes a breakdown of the multipole expansion in equation (38), when truncated at finite p . Although the multipole expansion could, in principle, be performed around another point (Elbers et al. 2021), we instead additionally implemented a version of the δf method that only applies the weights in the long-range PM gravity calculation. This choice ensures that the spurious back-reaction of neutrino shot noise, which is most prominent on large scales and therefore feeds through the long-range force, is eliminated, while the possibility of neutrinos affecting smaller scales through short-range forces is not excluded. An added benefit is that PM steps are rare for large calculations, such that the computational overhead of the δf step is minimal.

In addition, the δf weights are always used to reduce the noise in on-the-fly power spectra and are provided in snapshots for use in post processing.

A final point concerns the relativistic nature of neutrino particles at high redshift. To ensure that neutrino velocities do not exceed the speed of light and to recover the correct free streaming lengths, we apply the relativistic correction factor $c/\sqrt{c^2 + (v/a)^2}$ to neutrino drifts, where v is the internal velocity variable described in Section 5.3 and a is the scale factor. Relativistic corrections to the acceleration can be neglected in the time frame typical for cosmological simulations (Elbers 2022).

5 COSMOLOGICAL INTEGRATION

5.1 Background evolution

In SWIFT, we assume a standard FLRW metric for the evolution of the background density of the Universe and use the Friedmann equations to describe the evolution of the scale-factor $a(t)$. We scale a such that its present-day value is $a_0 \equiv a(t = t_{\text{now}}) = 1$. We also define redshift $z \equiv 1/a - 1$ and the Hubble parameter

$$H(t) \equiv \frac{\dot{a}(t)}{a(t)}, \quad (57)$$

with its present-day value denoted as $H_0 \equiv H(t = t_{\text{now}})$. Following usual conventions we write $H_0 = 100h \text{ km} \cdot \text{s}^{-1} \cdot \text{Mpc}^{-1}$ and use h as the input parameter for the Hubble constant.

To allow for general expansion histories, we use the full Friedmann equations and write

$$H(a) \equiv H_0 E(a), \quad (58)$$

$$E(a) \equiv \sqrt{\Omega_m a^{-3} + \Omega_r a^{-4} + \Omega_k a^{-2} + \Omega_\Lambda \exp(3\tilde{w}(a))}, \quad (59)$$

$$\tilde{w}(a) = (a - 1)w_a - (1 + w_0 + w_a) \log(a), \quad (60)$$

where we followed Linder & Jenkins (2003) to parametrize the evolution of the dark-energy equation of state²¹ as:

$$w(a) \equiv w_0 + w_a (1 - a). \quad (61)$$

The cosmological model is hence fully defined by specifying the dimensionless constants Ω_m , Ω_r , Ω_k , Ω_Λ , h , w_0 , and w_a as well as the starting redshift (or scale-factor of the simulation) a_{start} and final time a_{end} .

²¹Note that $\tilde{w}(z) \equiv \int_0^z \frac{1+w(z')}{1+z'} dz'$, which leads to the analytic expression we use.

At any scale-factor a_{age} , the time t_{age} since the Big Bang (age of the Universe) is computed as (e.g. Wright 2006):

$$t_{\text{age}} = \int_0^{a_{\text{age}}} dt = \int_0^{a_{\text{age}}} \frac{da}{aH(a)} = \frac{1}{H_0} \int_0^{a_{\text{age}}} \frac{da}{aE(a)}. \quad (62)$$

For a general set of cosmological parameters, this integral can only be evaluated numerically, which is too slow to be evaluated accurately during a run. At the start of the simulation we tabulate this integral for 10^4 values of a_{age} equally spaced between $\log(a_{\text{start}})$ and $\log(a_{\text{end}})$. The values are obtained via adaptive quadrature using the 61-points Gauss–Konrod rule implemented in the GSL library (Gough 2009) with a relative error limit of $\epsilon = 10^{-10}$. The value for a specific a (over the course of a simulation run) is then obtained by linear interpolation of the table.

5.2 Addition of neutrinos

Massive neutrinos behave like radiation at early times, but become non-relativistic around $a^{-1} \approx 1890(m_\nu/1\text{eV})$. This changes the Hubble rate $E(a)$ and therefore most integrated quantities described in the previous section. We optionally include this effect by specifying the number of massive neutrino species N_ν and their non-zero neutrino masses $m_{\nu,i}$ in eV ($m_{\nu,i} \neq 0, i = 1, \dots, N_\nu$). Multiple species with the same mass can be included efficiently by specifying mass degeneracies g_i . In addition, the present-day neutrino temperature $T_{\nu,0}$ must also be set²² as well as an effective number of ultra-relativistic (massless) species N_{ur} . Together with the present-day CMB temperature $T_{\text{CMB},0}$, these parameters are used to compute the photon density Ω_γ , the ultra-relativistic species density Ω_{ur} , and the massive neutrino density $\Omega_\nu(a)$, replacing the total radiation density parameter Ω_r . In our conventions, the massive neutrino contribution at $a = 1$ is *not* included in the present-day matter density $\Omega_m = \Omega_{\text{cdm}} + \Omega_b$. The radiation term appearing in equation (59) is simply replaced by

$$\Omega_r a^{-4} = [\Omega_\gamma + \Omega_{\text{ur}} + \Omega_\nu(a)] a^{-4}. \quad (63)$$

In this expression, the constant Ω_γ describes the CMB density and is given by

$$\Omega_\gamma = \frac{\pi^2 (k_B T_{\text{CMB},0})^4}{15 (hc)^3} \frac{1}{\rho_{\text{crit}} c^2}, \quad (64)$$

while the ultra-relativistic neutrino density is given by

$$\Omega_{\text{ur}} = \frac{7}{8} \left(\frac{4}{11} \right)^{4/3} N_{\text{ur}} \Omega_\gamma. \quad (65)$$

Note that we assume instantaneous decoupling for the ultra-relativistic species. The time-dependent massive neutrino density parameter is (Zennaro et al. 2017):

$$\Omega_\nu(a) = \Omega_\gamma \sum_{i=1}^{N_\nu} \frac{15}{\pi^4} g_i \left(\frac{T_{\nu,0}}{T_{\text{CMB}}} \right)^4 \mathcal{F} \left(\frac{am_{\nu,i}}{k_B T_{\nu,0}} \right), \quad (66)$$

where the function \mathcal{F} is given by the momentum integral

$$\mathcal{F}(y) = \int_0^\infty \frac{x^2 \sqrt{x^2 + y^2}}{1 + e^x} dx. \quad (67)$$

As $\Omega_\nu(a)$ is needed to compute other cosmological integrals, this function should be calculated with sufficient accuracy. At the start of

²²To match the neutrino density from an accurate calculation of decoupling (Mangano et al. 2005), one can use the value $T_{\nu,0}/T_{\text{CMB},0} = 0.71599$ (Lesgourgues & Tram 2011).

the simulation, values of equation (66) are tabulated on a piece-wise linear grid of $2 \times 3 \times 10^4$ values of a spaced between $\log(a_{\nu, \text{begin}})$, $\log(a_{\nu, \text{mid}})$, and $\log(a_{\nu, \text{end}}) = \log(1) = 0$. The value of $a_{\nu, \text{begin}}$ is automatically chosen such that the neutrinos are still relativistic at the start of the table. The value of $\log(a_{\nu, \text{mid}})$ is chosen just before the start of the simulation. The integrals $\mathcal{F}(y)$ are evaluated using the 61-points Gauss–Konrod rule implemented in the GSL library with a relative error limit of $\epsilon = 10^{-13}$. Tabulated values are then linearly interpolated whenever $E(a)$ is computed.

Besides affecting the background evolution, neutrinos also play a role at the perturbation level. These effects can be included in SWIFT using the linear response method of Ali-Haïmoud & Bird (2013) or the particle-based δf method of Elbers et al. (2021), as described in Section 4.7.

5.3 Choice of co-moving coordinates

Note that, unlike many other solvers, we do not express quantities with ‘little h’ (h) included²³; for instance units of length are expressed in units of Mpc and not Mpc/ h . As a consequence, the time integration operators (see below) also include an h -factor via the explicit appearance of the Hubble constant.

In physical coordinates, the Lagrangian for a particle i in an energy-based flavour of SPH with gravity reads

$$\mathcal{L} = \frac{1}{2} m_i \dot{\mathbf{r}}_i^2 - m_i u_i - m_i \phi_i. \quad (68)$$

Introducing the comoving positions \mathbf{r}' such that $\mathbf{r} = a(t)\mathbf{r}'$, we get

$$\mathcal{L} = \frac{1}{2} m_i (a\dot{\mathbf{r}}'_i + \dot{a}\mathbf{r}'_i)^2 - m_i \frac{u'_i}{a^{3(\gamma-1)}} - m_i \phi, \quad (69)$$

where the comoving internal energy $u' = ua^{3(\gamma-1)}$ is *chosen* such that the equation of state for the gas and thermodynamic relations between quantities have the same form (i.e. are scale-factor free) in the primed frame as well. Together with the definition of comoving densities $\rho' \equiv a^3(t)\rho$, this implies

$$P' = a^{3\gamma} P, \quad A' = A, \quad c' = a^{3(\gamma-1)/2} c, \quad (70)$$

for the pressure, entropy, and sound-speed respectively. Following Peebles (1980; chapter 7), we introduce the gauge transformation $\mathcal{L} \rightarrow \mathcal{L} + \frac{d}{dt} \Psi$ with $\Psi \equiv \frac{1}{2} a \dot{a} \mathbf{r}'_i^2$ and obtain

$$\mathcal{L} = \frac{1}{2} m_i a^2 \dot{\mathbf{r}}'_i{}^2 - m_i \frac{u'_i}{a^{3(\gamma-1)}} - \frac{\phi'}{a}, \quad (71)$$

$$\phi' = a\phi + \frac{1}{2} a^2 \ddot{a} \mathbf{r}'_i{}^2,$$

and call ϕ' the peculiar potential. Finally, we introduce the velocities used internally by the code:

$$\mathbf{v}' \equiv a^2 \dot{\mathbf{r}}', \quad (72)$$

allowing us to simplify the first term in the Lagrangian. Note that these velocities *do not* have a direct physical interpretation. We caution that they are not the peculiar velocities ($\mathbf{v}_p \equiv a\dot{\mathbf{r}}' = \frac{1}{a}\mathbf{v}'$), nor the Hubble flow ($\mathbf{v}_H \equiv \dot{a}\mathbf{r}'$), nor the total velocities ($\mathbf{v}_{\text{tot}} \equiv \mathbf{v}_p + \mathbf{v}_H = \dot{a}\mathbf{r}' + \frac{1}{a}\mathbf{v}'$) and also differ from the convention used in outputs produced by GADGET (Springel 2005; Springel et al. 2021) and other related simulation codes ($\mathbf{v}_{\text{out,Gadget}} = \sqrt{a}\mathbf{r}'$).²⁴

²³See e.g. Croton (2013) for a rational.

²⁴One inconvenience of our choice of generalized coordinates is that our velocities \mathbf{v}' and sound-speed c' do not have the same dependencies on the

5.3.1 SPH equations

Using the SPH definition of density, $\rho'_i = \sum_j m_j W(\mathbf{r}'_j - \mathbf{r}'_i, h'_i) = \sum_j m_j W'_{ij}(h'_i)$, we follow Price (2012) and apply the Euler-Lagrange equations to write

$$\dot{\mathbf{r}}'_i = \frac{1}{a^2} \mathbf{v}'_i, \quad (73)$$

$$\dot{\mathbf{v}}'_i = - \sum_j m_j \left[\frac{1}{a^{3(\gamma-1)}} f'_i P'_i \rho_i'^{-2} \nabla'_i W'_{ij}(h_i) + \frac{1}{a^{3(\gamma-1)}} f'_j P'_j \rho_j'^{-2} \nabla'_i W'_{ij}(h_j) + \frac{1}{a} \nabla'_i \phi' \right], \quad (74)$$

with

$$f'_i = \left[1 + \frac{h'_i}{3\rho'_i} \frac{\partial \rho'_i}{\partial h'_i} \right]^{-1}, \quad \nabla'_i \equiv \frac{\partial}{\partial \mathbf{r}'_i}.$$

These correspond to the equations of motion for density-entropy SPH (e.g. Equation 14 of Hopkins 2013) with cosmological and gravitational terms. Similarly, the equation of motion describing the evolution of u' is expressed as:

$$\dot{u}'_i = \frac{1}{a^2} \frac{P'_i}{\rho_i'^2} f'_i \sum_j m_j (\mathbf{v}'_i - \mathbf{v}'_j) \cdot \nabla'_i W'_{ij}(h_i). \quad (75)$$

In all these cases, the scale-factors appearing in the equations are later absorbed in the time-integration operators such that the RHS of the equations of motions is identical for the primed quantities to the ones obtained in the non-cosmological case for the physical quantities. Additional terms in the SPH equations of motion (e.g. viscosity switches) often rely on the velocity divergence and curl. We do not give a full derivation here but the co-moving version of all these terms can easily be constructed following the same procedure we employed here.

5.4 Time-integration operators

For the choice of cosmological coordinates made in SWIFT, the normal *kick* and *drift* operators get modified to account for the expansion of the Universe. The rest of the leapfrog algorithm is identical to the non-comoving case. The derivation of these operators from the system's Lagrangian is given in appendix A of Quinn et al. (1997) for the collisionless case. We do not repeat that derivation here but, for completeness, give the expressions we use as well as the ones used for the hydrodynamics. The drift operator gets modified such that Δt for a time-step running from a scale-factor a_n to a_{n+1} becomes

$$\Delta t_{\text{drift}} \equiv \int_{a_n}^{a_{n+1}} \frac{dt}{a^2} = \frac{1}{H_0} \int_{a_n}^{a_{n+1}} \frac{da}{a^3 E(a)}, \quad (76)$$

with $E(a)$ given by equation (60) and the a^{-2} chosen to absorb the one appearing in equation (73). Similarly, the time-step-entering kick operator for collisionless acceleration reads

$$\Delta t_{\text{kick,g}} \equiv \int_{a_n}^{a_{n+1}} \frac{dt}{a} = \frac{1}{H_0} \int_{a_n}^{a_{n+1}} \frac{da}{a^2 E(a)}. \quad (77)$$

However, for the case of gas dynamics, given our choice of coordinates, the kick operator has a second variant that reads

$$\Delta t_{\text{kick,h}} \equiv \int_{a_n}^{a_{n+1}} \frac{dt}{a^{3(\gamma-1)}} = \frac{1}{H_0} \int_{a_n}^{a_{n+1}} \frac{da}{a^{3\gamma-2} E(a)}. \quad (78)$$

scale-factor. The signal velocity entering the time-step calculation will hence read $v_{\text{sig}} = a\mathbf{r}' + c = \frac{1}{a} (|\mathbf{v}'| + a^{(5-3\gamma)/2} c')$.

Accelerations arising from hydrodynamic forces (first and second term in equation (74)) are integrated forward in time using $\Delta t_{\text{kick,h}}$, whilst the accelerations given by the gravity forces [third term in equation (74)] use $\Delta t_{\text{kick,g}}$. The internal energy (equation (75)) is integrated forward in time using $\Delta t_{\text{kick,u}} = \Delta t_{\text{drift}}$.

Following the same method as for the age of the Universe (Section 5.1), these three non-trivial integrals are evaluated numerically at the start of the simulation for a series 10^4 values of a placed at regular intervals between $\log(a_{\text{begin}})$ and $\log(a_{\text{end}})$. The values for a specific pair of scale-factors a_n and a_{n+1} are then obtained by interpolating that table linearly.

5.5 Validation

To assess the level of accuracy of SWIFT, it is important to compare results with other codes. This lets us assess the level of systematic differences and uncertainties left in the code. This is especially important for the studies of non-linear structure formation, as there is no possibility to use an exact solution to compare against. One such benchmark was proposed by Schneider et al. (2016) in the context of the preparation for the EUCLID survey. Their goal was to assess whether cosmological codes can converge towards the same solution, within the targeted 1 per cent accuracy of the survey. They focused on the matter density power spectrum as their observable and used three different N -body codes for their study. Importantly, their work utilized three codes using three different algorithms to solve for the gravity forces: RAMSES (Teyssier 2002, multigrid technique), PKDGRAV3 (Potter et al. 2017, FMM tree algorithm), and GADGET-3 (Springel 2005, tree-PM technique). The setup evolves a cosmological simulation in a $(500 \text{ Mpc}/h)^3$ volume from $z = 49$ to $z = 0$, assuming a Λ CDM cosmology, sampled using 2048^3 particles. The setup only considers gravitational interactions and comoving time integration. The same setup was later adopted by Garrison, Eisenstein & Pinto (2019) to compare their ABACUS code and by Springel et al. (2021) for the GADGET-4 code.²⁵ It is a testimony to the advances of the field in general and to the increase in available computing power that a run akin to the then-record-breaking MILLENNIUM simulation (Springel et al. 2005b) is nowadays used as a mere benchmarking exercise.

We ran SWIFT on the same initial conditions and analysed the results as described below. The exact configuration used for the SWIFT run is released as part of the code package, namely: a 2048^3 gravity mesh for the PM code, the adaptive MAC with $\epsilon_{\text{FMM}} = 10^{-3}$, and a Plummer-equivalent softening length $\epsilon = 10/h$ kpc. The top-left panel of Fig. 1 shows the projection of the matter density field in a $10 \text{ Mpc } h^{-1}$ slice rendered using the SWIFTS imIO tool (Borrow & Borrisov 2020). To ease the comparison to published results and eliminate any possible discrepancy coming from binning choices or exact definitions, we used the power-spectrum measurement tool embedded in the GADGET-4 code on our output to allow for a direct comparison with the data presented by Springel et al. (2021) (who had also reanalysed the other runs with their tool). We show our results alongside the published measurements from other codes in Fig. 16, each presented as ratios to the SWIFT prediction. The shaded regions correspond to ± 0.25 per cent and ± 1 per cent differences with respect to our results. Over the range of wavelengths of interest for this problem, the SWIFT results are in excellent agreement with the other codes. This agreement extends from the linear regime to the

²⁵We thank Lehman Garrison and Volker Springel for graciously providing their data and analysis tools.

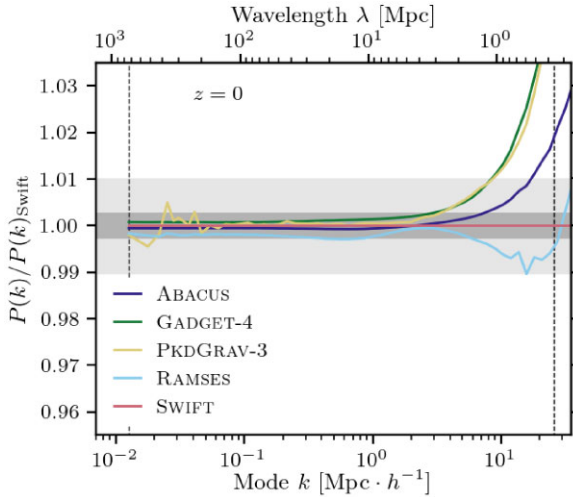


Figure 16. Comparison of the matter power-spectra as a function of scale for four different n -body codes (see text) relative to the SWIFT prediction on the test problem introduced by Schneider et al. (2016). The simulation evolves 2048^3 dark matter particles in a $(500 \text{ Mpc } h)^{-3}$ volume run from $z = 49$ to $z = 0$ assuming a Λ CDM cosmology. All power spectra were measured using the same tool (see text). The dark- and light-shaded regions correspond to ± 0.25 per cent and ± 1 per cent level agreement between codes. The fundamental mode (left) and the Nyquist frequency (right) are indicated using vertical dashed lines. Over the range of interest for modern cosmological applications, all codes agree to within 1 per cent.

non-linear regime ($k \gtrsim 0.1 \text{ Mpc}/h$). This confirms SWIFT’s ability to make solid predictions for modern cosmological applications.

Note also that a similar exercise was independently presented by Grove et al. (2022) in the context of the DESI survey code comparison effort, for which SWIFT, ABACUS, and GADGET-2 were compared. Comparing outputs at $z = 1$ and $z = 2$, they obtained results in excellent agreement with the ones presented here.

6 INPUT AND OUTPUT STRATEGY

We now turn our attention towards the input and output strategy used by the SWIFT code.

6.1 Initial conditions

To ease the use of the code and given the large number of legacy initial conditions (ICs) existing in the community using this format, we adopt the same file format for input as the ‘mode 3’ option of the GADGET-2 code (Springel 2005), i.e. the mode based on the `hdf5` library (The HDF Group 2022). SWIFT is fully compatible with any valid GADGET-2 set of initial conditions, but we also provide additional optional features. First, we allow for different units to be used internally and in the ICs. SWIFT would then perform a conversion upon start-up to the internal units. This can be convenient when a certain set of ICs uses a range of values problematic when represented in single-precision. Secondly, for cosmological runs, SWIFT can also apply the necessary h -factor and a -factor corrections (see Section 5.3) to convert to the system of co-moving coordinates adopted internally. A departure from the strict GADGET-2 format is that SWIFT only allows for the data to be distributed over a single file; we do, however, provide scripts to transform such distributed input files to our format.

Some tools also exist to directly generate SWIFT ICs with all the optional features added. The `SWIFTSIMIO`²⁶ python package (Borrow & Borrisov 2020) can be used to generate simple setups. The `SEAGen`²⁷ (Kegerreis et al. 2019) and `WoMa`²⁸ (Ruiz-Bonilla et al. 2021) packages are designed to generate spherical or spinning planetary bodies in equilibrium for collision problems (See Section 8.5). For cosmological simulations, the public version of the state-of-the-art ICs code `MONOFONIC` (Hahn, Rampf & Uhlemann 2021; Michaux et al. 2021) has been extended to be able to produce files that are directly compatible with the format expected by SWIFT. In particular, information about the adopted cosmological parameters, phases, and all the information required to re-generate the ICs are added to the files, read by SWIFT, and propagated to the snapshots. This allows for runs to be reproduced based solely on the information given in the SWIFT outputs.

6.2 Snapshots

For the same convenience reasons as for the ICs, we also adopt an output file format designed as a fully-compatible extension to the GADGET-2 (Springel 2005) ‘mode 3’ format based on the `hdf5` library (The HDF Group 2022). We extend the format by creating new particle groups for the species not existing in the original GADGET-2 code. We also add to the snapshots a full copy of the parameters used to perform the simulation, information about the version of the code, details of the cosmological models, and information about the ICs. Another noteworthy extension is the extensive use of units metadata in the snapshots. We attach full units information to every field in the snapshots. That information includes human-friendly and machine-readable conversion factors to the cgs system, as well as the conversion factor needed to move between the co-moving and physical frame (see Section 5.3). These metadata can be read by python packages such as `SWIFTSIMIO` (Borrow & Borrisov 2020) to then propagate this information through the simulation analysis. This mechanism is based on the `unyt` (Goldbaum et al. 2018) library. The particles are stored in the snapshots in order of the domain cells they belong to (see Section 9.1). Efficiently retrieving the particles located in a small sub-region of the computational domain is hence possible; for instance extracting the particles in the region around a single halo only. In large simulations, this is much more efficient than reading all the randomly ordered particles and then masking out the ones that do not fall in the region of interest. Metadata to ease such reading patterns are added to the snapshots. That information is picked up by tools such as `SWIFTSIMIO` to aid analysis of these massive simulations. The commonly used visualization package `yt`²⁹ (Turk et al. 2011) has also been extended to directly read in SWIFT snapshots, including the relevant meta-data.

The snapshots can either be written into one single file, with all nodes writing collectively to the same data set in parallel, or by splitting the data such that each node writes a file with its local subset of particles. That second option is preferable when using file systems that are not able to handle parallel writes to a single file efficiently. When writing such a distributed snapshot, an additional meta-snapshot is written; it contains all the information of a regular single-file snapshot, but uses `hdf5`’s virtual data set infrastructure to present the data distributed over many files as a

²⁶<https://github.com/SWIFTSIM/swiftsimio>

²⁷<https://github.com/jkeger/seagen>

²⁸<https://github.com/srbonilla/WoMa>

²⁹<https://yt-project.org/>

single contiguous array. The links between files are handled in the background by the library. These meta-snapshots can then be read as if they were standard snapshots, for instance via tools like `Gadgetviewer`³⁰ SWIFT can also optionally apply lossless compression to the snapshots (via `hdf5`'s own `gzip` filter) as well as a per-field lossy compression where the number of bits in the mantissa of the numbers can be reduced to save disk space. This option is particularly interesting when considering particle fields where the 23 bits of relative precision (i.e. ≈ 7 decimal digits) of a standard `float` type are more than sufficient for standard analysis.³¹ Similar filters can be applied to `double`-precision variables. Finally, SWIFT implements an option to down-sample the particles of a given type in the snapshots by writing only a fraction of the particles chosen at random.

As an example of i/o performance in a realistic scenario, the snapshots for the recent flagship FLAMINGO run (Schaye et al. 2023) were written in 200 seconds. They contain 2.65×10^{11} particles of different types spread over 960 files totalling 39 terabytes of data. This corresponds to a writing speed of 200 GB/s. As this test only used 65 per cent of the systems' nodes, this compares favourably to the raw capability (350 GB/s) of the full cluster. Compressing the data using both lossy and lossless filters reduces the snapshot size to 11 terabytes but the writing time increases to 1260 s. This corresponds to a sustained writing speed of 9 GB/s; the difference is due to the compression algorithm embedded within the `hdf5` library. Additionally, by making use of the library's parallel writing capability, we can repeat the uncompressed test but with all nodes writing to a single file. In this configuration, we require 463 s, effectively achieving a sustained parallel writing speed of 86 GB/s.

Snapshots can be written at regular intervals in time or change in scale-factor. Alternatively, the user can provide a list of outputs in order to specify output times more precisely. This list can be accompanied by a list of fields (or of entire particle types) the user does not want to be written to a snapshot. This allows for the production of reduced snapshots at high-frequency; for instance to finely track black holes. Any of the structure finders (Section 7) can be run prior to the data being written to disk to include halo membership information of the particles in the outputs.

6.3 Check-pointing mechanism

When running simulations at large computing centres, limits on the length of a given compute job are often imposed. Many simulations will need to run for longer than these limits and a mechanism to cleanly stop and resume a simulation is thus needed. This *check-pointing* mechanism can also be used to store backups of the simulation's progress in case one needs to recover from a software or hardware failure. Such a mechanism is different from the writing of science-ready snapshots as all the information currently in the memory needs to be saved; not just the interesting fields carried by the particles. These outputs are thus typically much larger than the snapshots and are of the same size as the memory used for the run.

In SWIFT, we choose to write one file per MPI rank. No pre-processing of any kind is done during writing. Each of the code's modules writes its current memory state one after the other. This includes the raw particle arrays, the cells, the tasks, and the content of the extensions (see Section 8) among many other objects. At the start each module's writing job we include a small header with

some information about the size of the data written. This allows us to verify that the data was read in properly when resuming a simulation. As these are simple, unformatted, large, and distributed writing operations, the code typically achieves close to the maximal writing speed of the system. For the same FLAMINGO run mentioned above, the whole procedure took 260 s for 64 TB of data in 960 files. This corresponds to a raw writing speed of 250 GB/s. As the check-pointing is fast, it is convenient to write files at regular intervals (e.g. every few hours) to serve as a backup.

When restarting a simulation from a check-point file, the opposite operation is performed. Each rank reads one file and restores the content of the memory. At this point, the simulation is in exactly the same state as it was when the files were written. The regular operations can thus resume as if no stoppage and restarting operation had ever occurred.

As is the case in many software packages, our implementation is augmented with a few practical options such as the ability to stop an on-going run or to ask the simulation to run for a set wall-clock time before writing a check-point file and stopping itself.

6.4 Line-of-sight outputs

In addition to full-box snapshots, SWIFT can also produce so-called *line-of-sight* outputs. Randomly positioned rays (typically perpendicular to a face) are cast through the simulation volume and all gas particles whose volumes are crossed by the infinitely thin rays are stored in a list. We then write all the properties of these particles for each ray to a snapshot with a format similar to the one described above but much reduced in volume. These outputs can then be used to produce spectra via tools such as `SPECWIZARD` (Schaye et al. 2003; Tepper-García et al. 2011). Thanks to their small data footprints, these line-of-sight snapshots are typically produced at high time frequencies over the course of a run. This type of output is particularly interesting for simulations of the IGM and Lyman- α forest (see Section 8.4).

6.5 Lightcone outputs

To bring the cosmological simulation outputs closer to observation mock catalogs, SWIFT implements two separate mechanisms to record information as particles cross the past light cone of a selection of observers placed in the simulation box. The first mechanism writes the particles to disk as they reach a distance from the observer corresponding to the light-travel distance of the look-back time to the outputs. The second mechanism accumulates particle information in redshift shells onto pixels to directly construct maps as the simulation runs. See the appendix of Schaye et al. (2023) for a detailed use case of both these mechanisms.

6.5.1 Particle data

The position of each observer, the redshift range over which light-cone particle output will be generated, and the opening angle of the cone are specified at run time. At each time-step we compute the earliest and latest times that any particles could be integrated forward to and the corresponding co-moving distances. This defines a shell around each observer in which particles might cross the past light cone as a result of drift operations carried out during this time-step. An additional boundary layer is added to the inside of the shell to account for particles that move during the time-step and assuming that they have sub-luminal speeds.

³⁰<https://github.com/jchelly/gadgetviewer/>

³¹Classic examples are the temperature field or the particles' metallicity.

For simulations employing periodic boundary conditions, we must additionally output any periodic copy of a particle which crosses the observer's light cone. We therefore generate a list of all periodic copies of the simulation volume that overlap the shell around the observer. Then, whenever a particle is moved, we check every periodic copy for a possible overlap with any of the shells. If so, the particle's position is interpolated to the exact redshift at which it crossed the lightcone and the particle is added to a writing buffer. When the buffer reaches a pre-defined size, we write out the particles including all their properties to disk.

To optimize the whole process, we take advantage of the way that SWIFT internally arranges the particles in a cubic grid of cells (Section 9.1). We can use this structure to identify which tree cells overlap with the current lightcone shells. This allows us to reduce the number of periodic replications to check for every particle. Only the particles in the cells previously identified need to undergo this process.

In most cases, the raw data generated by the particle lightcone requires some post-processing; for instance to reorganize the particles inside the files in terms of angular coordinates on the sky and redshift.

6.5.2 HEALPix maps

Light-cone particle outputs as well as the internal memory requirement rapidly grow in size as the upper redshift limit is increased, especially if many box replications occur, and can become impractical to store. SWIFT therefore also contains a scheme to store spherical maps of arbitrary quantities on the light cone with user specified opening angle, angular resolution, and redshift bins.

To this end, the observer's past light cone is split into a set of concentric spherical shells in co-moving distance. For each shell we create one full-sky HEALPix (Górski et al. 2005) map for each quantity to be recorded. Whenever a particle is found to have entered one of these shells, we accumulate the particles' contributions to the HEALPix maps for that shell. Typical examples are the construction of mass or luminosity maps. Particles can also, optionally, be smoothed onto the maps using an SPH kernel.

As the maps do not overlap in redshift, it is not necessary to store all of the shells simultaneously in memory. Each map is only allocated and initialized when the simulation first reaches the time corresponding to the outer edge of the shell. It is then written to disk and its memory freed once all the particles have been integrated to times past that corresponding to the light travel time to the inner edge of the shell. In practice, the code will hence only need to have a maximum of two maps in memory at any point in time.

6.6 On-the-fly power spectra

Finally, SWIFT can compute a variety of auto- and cross- power spectra at user-specified intervals. These include the mass density in different particle species (and combinations thereof) as well as the electron pressure. For the neutrino density, we also implement the option to randomly select one half of the particles only or the other. This helps reduce the shot-noise by computing a cross-spectrum between the two halves.

The calculation is performed on a regular grid (usually of size 256^3 and hence allowing for the Fourier transform to be performed on a single node). Foldings (Jenkins et al. 1998) are used to extend the range probed to smaller scales with a typical folding factor of 4 between iterations. Different window functions from nearest-grid-point, to CIC, to triangular-shaped-clouds can be used and are

compensated for self-consistently (see e.g. Colombi et al. 2009). This could easily be extended to higher-order schemes and to more particle properties.

6.7 Continuous non-blocking adaptive output strategy

In SWIFT we also include a novel output strategy called the *Continuous Simulation Data Stream* (CSDS), described by Hausammann, Gonnet & Schaller (2022). The key principles are summarized here (for related ideas, see Faber et al. 2010; Rein & Tamayo 2017).

In classic output strategies (Section 6.2), the simulation is stopped at fixed time intervals and the current state of the system is written to disk, similar to the frames of a movie. This is an expensive operation where all the compute nodes suddenly stop processing the physics and instead put an enormous stress on the communication network and file-system. During these operations, the state of the system is not advanced, leading to an overall loss in performance as the whole simulation has to wait until the i/o operations have completed. Furthermore, in simulations with deep time-step hierarchies, only few particles are active on most steps, with most particles just drifting forward. In a cosmological context, a large fraction of the particles have fairly simple trajectories, barely departing from first- or second-order perturbation theory tracks. Only the small fraction of particles deep inside haloes follow complex trajectories. For the first group of particles, simulations typically have more snapshots than necessary to trace them, whilst for the second group, even one thousand snapshots (say) over a Hubble time may not be sufficient to accurately re-create their trajectory. It is hence natural to consider a more adaptive approach.

The CSDS departs from the snapshot idea by instead creating a data base of updates. At the start of a simulation an entry is written for each particle. We then start the simulation and progress the particles along. In its simplest form, the CSDS then adds an entry for a particle to the data base every few (~ 10) particle updates. As the writing is done on a particle-by-particle basis, it can easily be embedded in the tasking system. Writing is no longer a global operation where the whole simulation stops; rather updates are made continuously. By writing an update every few particle steps, the trajectory of each particle is, by construction, well-sampled, irrespective of whether it is in a very active region (e.g. haloes) or not (e.g. in voids). With this mechanism, particles outside of structures can have as little as two entries (start time and end time of the simulation) whilst some particles will have thousands of entries. Since the time-step size of a particle is designed to correctly evolve a particle, relying on this information to decide when to write a data base entry guarantees that the particles' evolution can later be faithfully recreated. Each entry for a particle contains a pointer to the previous entry such that particles can easily be followed in time.

An improved version of this approach would be to write a data base entry every time a particle field has changed by some pre-defined fraction ϵ . This is an important philosophical change; instead of creating frames at fixed intervals, we can demand that the evolution of *any* quantity be reconstructed to some accuracy from the output and get the CSDS to create the individual particle entries at the required times. The somewhat arbitrary choice of time interval between snapshots is hence replaced by an objective accuracy threshold.

This data base of particle updates allows for many new simulation analysis options. The trajectory and evolution of any particle can be reconstructed to the desired accuracy; that is we have all the information for a high time-resolution tracking of all the objects in a run. The first use is to produce classic snapshots at any position in time. We simply interpolate all the particle entries to that fixed time.

But, one can also demand to construct slices in space-time, i.e. a light-cone from the output. New possibilities arising from this new output format will undoubtedly appear in the future. Tools to perform the basic operations described here are part of the CSDS package linked to SWIFT. The tools, and most of the analysis performed thus far, are currently focused on dark-matter simulations, but we expect to extend this to more complex scenarios in the future.

7 STRUCTURE FINDING

7.1 Friends-Of-Friends group finder

The classic algorithm to identify structures in simulations is *Friends-Of-Friends* (FOF, see e.g. Davis et al. 1985). Particles are linked together if they are within a fixed distance (linking length) of each other. Chains of links form groups, which in a cosmological context are identified as haloes. For a linking length of 0.2 of the mean inter-particle separation, the haloes found are close (by mass) to the virialized structures identified by more sophisticated methods. The FOF method falls into the wider class of *Union-Find* algorithms (Galler & Fisher 1964) and very efficient implementations have been proposed over the last decade for a variety of computing architectures (e.g. Creasey 2018).

The implementation in SWIFT is fully described by Willis et al. (2020). In brief, the algorithm operates on a list of disjoint sets. The *Union* operation merges two sets and the *Find* operation identifies the set a given element resides in. Initially, each set contains a single element (one particle), which plays the role of the set identifier. The algorithm then searches for any two pairs of particles within range of each other. When such a pair is identified, the *Find* operation is used to identify which set they belong to. The *Union* operation is then performed to merge the sets if the particles do not already belong to the same one. To speed-up the pair-finding process, we use the same base principles as the ones discussed in Section 2. More precisely, by using the linking length as the search radius, we can construct a series of nested grids down to that scale. The search for links between particles can then be split between interactions within cells and between pairs of neighbouring cells. The tasking infrastructure can then be used to distribute the work over the various threads and nodes. When running a simulation over multiple compute nodes, the group search is first performed locally, then fragments of groups are merged together across domains in a second phase. This is however very different from other particle-particle interactions like the ones used for e.g. hydrodynamics, where the interactions are performed simultaneously, i.e. strictly within a single phase. Additional optimizations are described by Willis et al. (2020), alongside scaling results demonstrating excellent strong and weak scaling of the implementation.

Structures identified via SWIFT’s internal FOF can either be used to seed black holes (see Section 8.1.4) or be written as a halo or group catalogue output. Additionally, the FOF code can be run as stand-alone software to post-process an existing snapshot and produce the corresponding group catalogue.

7.2 Coupling to VELOCIRaptor

Many algorithms have been proposed to identify bound structures and sub-structures inside FOF objects (for a review, see Knebe et al. 2013). Many of them can be run on simulation snapshots in a post-processing phase. However, that is often inefficient as it involves substantial i/o work. In some cases, it can also be beneficial to have access to some of the (sub-)halo membership

information of a particle inside the simulation itself. For these reasons, the SWIFT code contains an interface to couple with the VELOCIRaptor code (Elahi, Thacker & Widrow 2011; Elahi et al. 2019). VELOCIRaptor uses phase-space information to identify structures using a 6D FOF algorithm. An initial 3D FOF is performed to identify haloes, however, this process may artificially join haloes together via a single particle, which is known as a *particle bridge*. These haloes are split apart by running a 6D FOF to identify particle bridges based upon their velocity dispersion. Large mergers are then identified in an iterative search for dense phase-space cores. Gravitationally unbound particles can optionally be removed from the identified structures. Such a substructure algorithm has the advantage over pure configuration-space algorithms of being able to identify sub-haloes deep within a host halo, where the density (or potential) contrasts relative to the background are small.

Over the course of a SWIFT run, the VELOCIRaptor code can be invoked to identify haloes and sub-haloes. To this end, the public version of the structure finder was modified to be used as a library. At user-specified intervals (typically at the same time as snapshots), SWIFT will create a copy of the particle information and format it to be passed to VELOCIRaptor. This process leads to some duplication of data but the overheads are small as only a small subset of the full particle-carried information is required to perform the phase-space finding. This is particularly the case for simulations which employ a full galaxy-formation model, where particles carry many additional tracers irrelevant to this process.

When the structure identification is completed, the list of structures and the particle membership information is passed back from the library to SWIFT. This information can then either be added to snapshots or be acted upon if any of the sub-grid models so require.

As an example, we ran SWIFT with VELOCIRaptor halo finding on the benchmark simulation of Schneider et al. (2016) introduced in Section 5.5. The resulting halo mass function is shown on Fig. 17 alongside the reference fitting function of Tinker et al. (2010) for the same cosmology. Our results are in excellent agreement with the predictions from the literature.

8 EXTENSIONS

Besides the coupled hydrodynamics and gravity solver, the SWIFT code also contains a series of extensions. These include complete galaxy formation models, AGN models, multimaterial planetary models, and a series of external potentials. These features are briefly summarized over the next pages.

8.1 The SWIFT-EAGLE galaxy formation model

An implementation of an evolution of the sub-grid models used for the EAGLE project (Crain et al. 2015; Schaye et al. 2015) is part of the SWIFT code. The model is broadly similar to the original GADGET-based implementation but was improved in several areas. Some of these changes also arose from the change of SPH flavour from a pressure-based formulation (see Schaller et al. 2015, for the version used in EAGLE) to the SPHENIX energy-based flavour tailored specifically for galaxy formation simulations (Section 3.3). We summarize here the main components of the model. All the parameters presented below have values that can be adjusted for specific simulation campaigns and are stored in parameter files that SWIFT reads in upon startup. The example parameter files provided in the SWIFT repository contain the parameter values for this model that were obtained via the calibration procedure of Borrow et al. (in preparation).

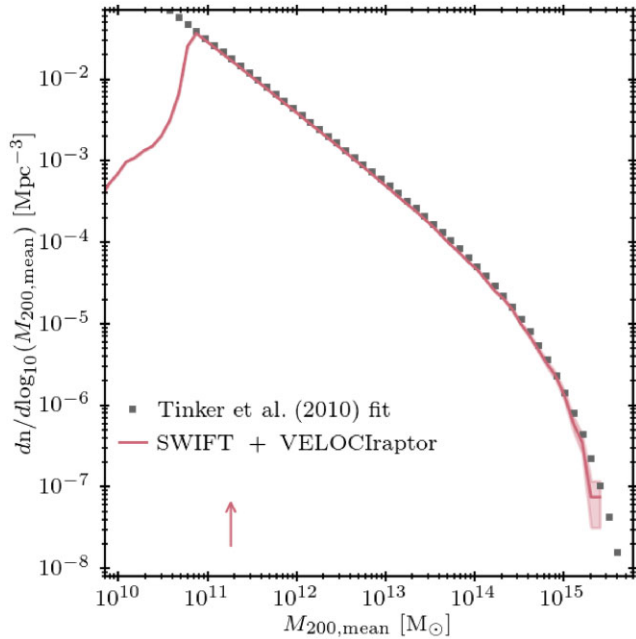


Figure 17. The halo mass function, computed using VELOCtraptor as the structure finder, extracted from the benchmark cosmological simulation of Schneider et al. (2016) run with SWIFT (See Section 5.5) and compared with the fitting function of Tinker et al. (2010). The shaded region depicts the $1 - \sigma$ Poisson errors on the counts, while the arrow indicates the mass corresponding to 100 particles.

8.1.1 Radiative cooling and heating

The radiative cooling and heating rates are pre-calculated on an element-by-element basis given the element abundance of each particle. The gas mass fractions of H, He, C, N, O, Ne, Mg, Si, and Fe are explicitly tracked in the code and directly affected by metal enrichment, while the abundance of S and Ca is assumed to scale with the abundance of Si using solar abundance ratios. SWIFT can use the tabulated cooling rates from Wiersma, Schaye & Smith (2009a) (W09) for optically thin gas from the original EAGLE runs, as well as the various public tables from Ploekinger & Schaye (2020; PS20). Compared to W09, the PS20 tables are computed with a more recent version of Cloudy: c07 (Ferland et al. 1998) in W09 and c17 (Ferland et al. 2017) in PS20, use an updated version of the UV and X-ray background (Haardt & Madau (2001) in W09 and a background based on Faucher-Giguère (2020) in PS20 and include physical processes relevant for optically thick gas, such as cosmic rays, dust, molecules, self shielding, and an interstellar radiation field.

8.1.2 Entropy floor and star formation

In typical EAGLE-like simulations, the resolution of the model is not sufficient to resolve the cold dense phase of the ISM, its fragmentation, and the star formation that ensues. We hence implement an entropy floor following Schaye & Dalla Vecchia (2008), which is typically set with a normalization of 8000 K at a density of $n_{\text{H}} = 0.1 \text{ cm}^{-3}$ with a slope expressed by the equation of state for pressure as $P \propto \rho^{4/3}$.

The star formation model uses the pressure-law model of Schaye & Dalla Vecchia (2008) which relates the star formation rates to the surface density of gas. Particles are made eligible for star formation

based on two different models. The first one follows EAGLE and uses a metallicity-dependent density threshold based on the results of Schaye (2004). The second model exploits the Ploekinger & Schaye (2020) tables. By assuming pressure equilibrium, we find the density and temperatures on the thermal equilibrium curve for the particles limited by the entropy floor. A combination of density and temperature threshold is then used with these sub-grid quantities (typically $n_{\text{H}} > 10 \text{ cm}^{-3}$ and $T < 1000 \text{ K}$). In practice, both models lead to broadly similar results.

Once a gas particle has passed the threshold for star formation, we compute its star formation rate based on two different models. We either assume a Schmidt (1959) law with a fixed efficiency per free-fall time, or use the pressure-law of Schaye & Dalla Vecchia (2008), which is designed to reproduce the Kennicutt (1998) relation. Based on the particle masses and computed star formation rate, random numbers are then drawn to decide whether the particles will indeed be converted into a star particle or not. The star particles formed in this manner inherit the metal content and unique ID of their parent gas particle.

8.1.3 Stellar enrichment & feedback

Stellar enrichment is implemented for the SNIa, core-collapse, and AGB channels using the age- and metal-dependent yields compilation of Wiersma et al. (2009b). The light emitted by the stars in various filters, based on the model of Trayford et al. (2015), is written to the snapshots. Stellar feedback is implemented using a stochastic thermal form (Dalla Vecchia & Schaye 2012) with various options to choose which neighbour in a star particle’s kernel to heat (Chaikin et al. 2022). The energy per supernova injection can either be kept fixed or be modulated by the local metallicity or density (Crain et al. 2015). Additionally, SWIFT includes the modified version of the stochastic kinetic feedback model of Chaikin et al. (2023) that was used in the FLAMINGO simulations (Kugel et al. 2023; Schaye et al. 2023). The SNe can either inject their energy after a fixed delay or can stochastically sample the stars’ lifetimes. The energy injection from SNIa is done by heating all the particles in the stars’ SPH kernel during each enrichment step.

8.1.4 Black holes & AGN feedback

Black hole (BH) particles are created by converting the densest gas particle in FOF-identified haloes (see Section 7.1) that do not yet contain a BH and are above a user-defined mass threshold. BHs grow by accreting mass from their neighbourhood, using a Bondi (1952) model, possibly augmented by density-dependent boosting terms (Booth & Schaye 2009) or angular-momentum terms (Rosas-Guevara et al. 2015). BH particles can swallow neighbouring gas particles when they have accreted enough mass or can ‘nibble’ small amounts of mass from them (see Bahé et al. 2022). Feedback from AGN is implemented using a stochastic thermal heating mechanism where energy is first stored into a reservoir until a pre-defined number of particles can be heated to a set temperature (Booth & Schaye 2009). Finally, the various modes of repositioning BHs presented in Bahé et al. (2022) are available as part of the EAGLE model in SWIFT.

8.1.5 Results

The model and the calibration of its free parameters are fully described by Borrow et al. (in preparation), alongside a com-

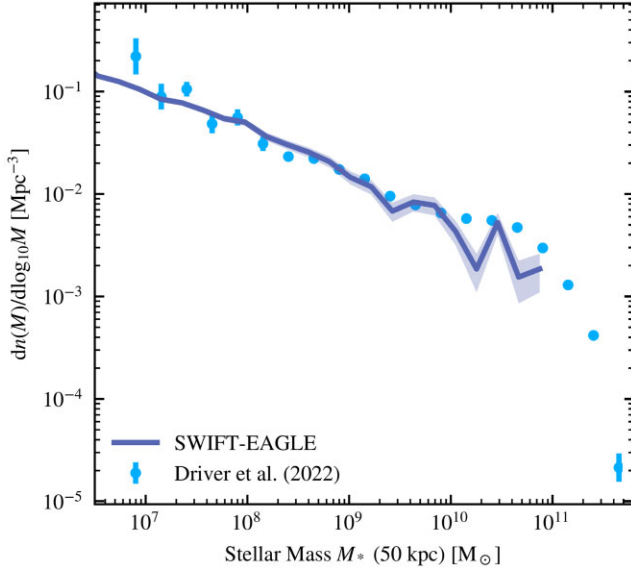


Figure 18. The galaxy stellar mass function, computed using `VELOCIRAPTOR` as the structure finder and measured in 50 kpc spherical apertures, extracted from a $(25 \text{ Mpc})^3$ volume run with SWIFT-EAGLE model and compared to the Driver et al. (2022) data inferred from the GAMA survey. The shaded region on the simulation corresponds to Poisson error counts in each 0.2 dex mass bin.

prehensive set of results. For completeness, we show here the $z = 0$ galaxy stellar mass function measured in 50 kpc spherical apertures (see appendix of de Graaff et al. 2022) from a $(25 \text{ Mpc})^3$ simulation with 2×376^3 particles in Fig. 18. The baryon particle mass in this simulation is $m_{\text{gas}} = 1.81 \times 10^6 M_{\odot}$, the resolution of the EAGLE simulations and the resolution at which the model was calibrated. For comparison, we show the Driver et al. (2022) estimates of the mass function obtained from the GAMA survey. Over the range where the masses are resolved and the galaxies are not too rare to feature in such a small volume, the SWIFT-EAGLE model produces is in good agreement with the data. That same model was used by Altamura et al. (2023) for their studies of groups and clusters; a map of the gas temperature weighted by its velocity dispersion extracted from one of their simulated clusters is displayed on panel (b) of Fig. 1.

We note that the exact parameters and initial conditions for this simulation are provided as part of the code release.

8.2 GEAR-like galaxy formation model

The GEAR physical model implemented in SWIFT is based on the model initially implemented in the GEAR code (Revaz & Jablonka 2012; Revaz et al. 2016; Revaz & Jablonka 2018), a fully parallel chemo-dynamical Tree/SPH code based on GADGET-2 (Springel 2005). While GEAR can be used to simulate Milky Way-like galaxies (Kim et al. 2016; Roca-Fàbrega et al. 2021) its physical model has been mainly calibrated to reproduce Local Group dwarf galaxies (Harvey et al. 2018; Revaz & Jablonka 2018; Hausammann, Revaz & Jablonka 2019; Sanati et al. 2020) and ultra-faint dwarfs (Sanati et al. 2023). We review hereafter the main features of the model; more details about the SWIFT implementation can be found in Hausammann (2021). An example galaxy from the *Agora*-suite (Kim et al. 2016) run using SWIFT-GEAR is displayed in panel (c) of Fig. 1.

8.2.1 Gas radiative cooling and heating

Radiative gas cooling and heating is computed using the `Grackle` library (Smith et al. 2017). In addition to primordial gas cooling, it includes metal-lines cooling, obtained by interpolating tables, and scaled according to the gas metallicity. `Grackle` also includes UV-background radiation heating based on the prediction from Haardt & Madau (2012). Hydrogen self-shielding against the ionizing radiation is incorporated. Two shielding options can be used: (1) the UV-background heating for gas densities above $n_{\text{H}} = 0.007 \text{ cm}^{-3}$ (Aubert & Teyssier 2010), and (2) the semi-analytic prescriptions of Rahmati et al. (2013) directly included in the `Grackle` cooling tables.

8.2.2 Pressure floor

To prevent gas from artificially fragmenting at high density and low temperature, i.e. when the Jeans length is not resolved (Bate & Burkert 1997; Owen & Villumsen 1997; Truelove et al. 1997), the gas' normal adiabatic equation of state is supplemented by a non-thermal pressure term. This additional term, interpreted as the non-thermal pressure of the unresolved ISM turbulence, artificially increases the Jeans length to make it comparable to the gas resolution (Robertson & Kravtsov 2008; Schaye & Dalla Vecchia 2008). The GEAR model uses the following pressure floor, a modified version of the formulation proposed by Hopkins, Quataert & Murray (2011):

$$P_{\text{Jeans}} = \frac{\rho}{\gamma} \left(\frac{4}{\pi} G h^2 \rho N_{\text{Jeans}}^{2/3} - \sigma^2 \right), \quad (79)$$

where G is the universal gravitational constant and, γ the adiabatic index of the gas fixed to 5/3. h , ρ , and σ are respectively the SPH smoothing length, density, and velocity dispersion of the gas particle. The parameter N_{Jeans} (usually set to 10) is the ratio between the SPH mass resolution and the Jeans mass.

8.2.3 Star formation and pressure floor

Star formation is modelled using a modified version of the stochastic prescription proposed by Katz (1992) and Katz, Weinberg & Hernquist (1996) that reproduces the Schmidt (1959) law. In the GEAR model star formation proceeds only in dense and cold gas phases where the physics is unresolved, i.e. where the artificial Jeans pressure dominates. Inverting equation (79), the temperature and resolution-dependent density threshold that delimits the resolved and unresolved gas phases is defined:

$$\rho_{\text{SFR},i} = \frac{\pi}{4} G^{-1} N_{\text{Jeans}}^{-2/3} h_i^{-2} \left(\gamma \frac{k_{\text{B}}}{\mu m_{\text{H}}} T + \sigma_i^2 \right). \quad (80)$$

Above this limit, the gas particles are eligible to form stars. It is possible to supplement this threshold with a constant density threshold, which prevents the stars from forming in cold and low-density gas regions, or by a temperature threshold, which prevents stars from forming in hot phases. Finally, only particles with a negative divergence of the velocity are eligible to form stars.

Once a particle of mass m_{g} is eligible, it will have a probability p_{\star} to form a stellar particle of mass m_{\star} during a time interval Δt (Springel & Hernquist 2003):

$$p_{\star} = \frac{m_{\text{g}}}{m_{\star}} \left[1 - \exp \left(-\frac{c_{\star}}{t_{\text{g}}} \Delta t \right) \right], \quad (81)$$

where c_* is a free parameter and t_g the local free fall time. Each gas particle can form a maximal number N_* of stellar particles over the whole simulation. N_* is a free parameter set by default to 4.

The GEAR model can use a critical metallicity $[\text{Fe}/\text{H}]_c$ parameter to differentiate stellar populations. Below $[\text{Fe}/\text{H}]_c$, a stellar particle will represent a Pop III (metal-free) population and above the critical metallicity, it will be considered a Pop II star. Both populations are characterized by different initial mass functions (IMF), stellar yields, stellar lifetimes, and energies of supernova explosions. All this information is provided to SWIFT by tables that can be generated by the PYCHEM.³² utility.

8.2.4 Stellar feedback, chemical evolution, and metal mixing

At each time step following the creation of a stellar particle, the IMF and stellar lifetimes-dependent number of exploding supernova (core collapse and Type Ia) is computed. This number that can be less than one and is turned into an integer number using a stochastic procedure called the random discrete IMF sampling (RIMFS) scheme in which the IMF is considered as a probability distribution (Revaz et al. 2016). Once a supernova explodes, its energy and synthesized elements are injected into the surrounding gas particles using weights provided by the SPH kernel. A parameter ϵ_{SN} may be used to decide the effective energy that will impact the ISM, implicitly assuming that the remainder will be radiated away.

To avoid instantaneous radiation of the injected energy, the delayed cooling method, which consists in disabling gas cooling for a short period of time of about 5 Myr (Stinson et al. 2006), is used.

The released chemical elements are further mixed in the ISM using either the smooth metallicity scheme (Okamoto et al. 2005; Tornatore et al. 2007; Wiersma et al. 2009b) or explicitly solving a diffusion equation using the method proposed by Greif et al. (2009).

8.3 Spin-driven AGN jet feedback

This model for AGN feedback is fully described by Huško et al. (2022) and Huško et al. (2024). We summarize here its main features. This sub-grid model only contains a prescription for AGN and can be used in combination with the EAGLE-like model described above for the rest of the galaxy formation processes.

In this model for AGN feedback, additional sub-grid physics related to accretion disks is included, allowing the evolution of spin (angular momentum) for each black hole in the simulation. This in turn means that one can use the spin-dependent radiative efficiency, instead of using a constant value (e.g. 10 per cent) for the thermal feedback channel employed in the fiducial model. More significantly, tracking black hole spins also allows for the inclusion of an additional mode of AGN feedback in the form of kinetic jets. The hydrodynamic aspects of the jets and their interaction with the CGM were tested by Huško & Lacey (2023). These jets are included in a self-consistent way by using realistic jet efficiencies (that depend strongly on spin), and by accounting for the jet-induced spindown of black holes. In the standard version of the model, at high accretion rates it is assumed that thermal feedback corresponds to radiation from sub-grid thin, radiatively-efficient accretion discs (Shakura & Sunyaev 1973). At low accretion rates, jets are launched from unresolved, thick, advection-dominated accretion discs (Narayan & Yi 1994). In more complicated flavours of the model, jets are also launched at high accretion rates and radiation (thermal feedback) at low accretion

rates, as well as strong jets and thermal feedback from slim discs at super-Eddington accretion rates—all of which is motivated by either observational findings or simulations.

These modifications to the AGN feedback may lead to more realistic populations of galaxies, although they probably have a stronger impact on the properties of the CGM/ICM. Although the model comes with the price of a more complicated feedback prescription (which involves some number of free parameters), it also opens an avenue for further observational comparisons between simulations and observations. The model yields predictions such as the spin–mass relation for black holes or the AGN radio luminosity function. These relations can be used to constrain or discriminate between versions of the model.

8.4 Quick-Lyman-alpha implementation

Besides galaxy formation models, another popular application of cosmological hydrodynamical simulations is the study of the intergalactic medium (IGM) via the Lyman- α forest. So-called ‘Quick-Lyman-alpha’ codes have been developed (e.g. Viel, Haehnelt & Springel 2004; Regan, Haehnelt & Viel 2007) to simulate the relevant physics. As the focus of such simulations is largely on the low-density regions of the cosmic web, a very simplified network of sub-grid model can be employed. In particular, for basic applications at least, the chemistry and cooling can be limited to only take into account the primordial elements. Similarly, any high-density gas can be turned into dark matter particles as soon as the gas reaches a certain overdensity (typically $\Delta = 1000$). In such a case, no computing time is wasted on evolving the interior of haloes, which allows for a much shallower time-step hierarchy than in a full galaxy formation model and thus much shorter run times.

We implement such a model in SWIFT. The ‘star formation’ is designed as described above: any gas particle reaching an overdensity larger than a certain threshold is turned into a dark matter particle. The cooling makes use of the table interpolation originally designed for the SWIFT-EAGLE model (Section 8.1). Either the W09 or the P20 tables can be used. Of particular interest for Quick-Lyman-alpha applications, these are based on two different models of the evolution of the UV background: Haardt & Madau (2001) and Faucher-Giguère (2020), respectively. A simulation using the W09 tables would be similar to the ones performed by Garzilli et al. (2019).

8.5 Material extensions and planetary applications

SWIFT also includes features that can be used to model systems with more complicated and/or multiple equations of state (EoS), and to better deal with density discontinuities. They are organized under a nominal ‘planetary’ label, given their initial application to giant impacts (Kegerreis et al. 2019). These extensions can be applied either onto a ‘MINIMAL’-like solver, with the inclusion of the Balsara (1995) viscosity switch, or in combination with the other, more sophisticated SPH modifications described below.

8.5.1 Equations of state

Many applications of SPH involve materials for which an ideal gas is not appropriate, and may also require multiple different materials. Included in SWIFT are a wide variety of EoS, which use either direct formulae (e.g. Tillotson 1962) or interpolation of tabulated data (e.g. Stewart et al. 2020; Chabrier & Debras 2021) to compute the required

³²<http://astro.epfl.ch/projects/PyChem>

thermodynamic variables. Each individual SPH particle is assigned a material ID that determines the EoS it will use. By default, no special treatment is applied when particles of different EoS are neighbours: the smoothed densities are estimated as before, and the pressure, sound speed, and other thermodynamic variables are then computed by each particle using its own EoS.

Currently implemented are EoS for several types of rocks, metals, ices, and gases. Custom user-provided EoS can also be used. Some materials can, for example, yield much more dramatic changes in the pressure for moderate changes in density than an ideal gas, and can also account for multiple phase states. In practice, in spite of the comparative complexity of some of these EoS, invoking them does not have a significant effect on the simulation run speed, because they are called only by individual particles instead of scaling over multiple neighbours.

Some input EoS may include a tension regime, where the pressure is negative for a cold, low-density material. This is usually undesired behaviour in a typical SPH simulation and/or implies an unphysical representation of the material in this state as a fluid, and can lead to particles accelerating towards each other and overlapping in space. As such, by default, a minimum pressure of zero for these EoS is applied.

8.5.2 Special treatment for initial conditions

Prior to running a simulation, it is a common practice to first perform a ‘settling’ run to relax the initial configuration of particles. This is particularly pertinent to planetary and similar applications, where the attempted placement of particles to model a spherical or spinning body will inevitably lead to imperfect initial SPH densities (Kegerreis et al. 2019; Ruiz-Bonilla et al. 2021). If the applied EoS includes specific entropies, then SWIFT can explicitly enforce the settling to be adiabatic, which may be a convenient way to maintain an entropy profile while the particles relax towards equilibrium.

8.5.3 Improvements for mixing and discontinuities

Standard SPH formulations assume a continuous density field, so can struggle to model contact discontinuities and to resolve mixing across them (e.g. Price 2008). However, density discontinuities appear frequently in nature. For example, in a planetary context, sharp density jumps might appear both between a core and mantle of different materials, and at the outer vacuum boundary. Smoothing particles’ densities over these desired discontinuities can lead to large, spurious pressure jumps, especially with complex EoS.

We have developed two approaches to alleviate these issues in SWIFT, briefly summarized here, in addition to the significant benefits of using more SPH particles for higher resolutions than were previously feasible. First, a simple statistic can be used to identify particles near to material and/or density discontinuities and to modify their estimated densities to mitigate the artificial forces and suppressed mixing (Ruiz-Bonilla et al. 2022). This method is most effective when combined with the geometric density-average force (GDF) equations of motion (Wadsley et al. 2017).

Second, a more advanced scheme in which density discontinuities are addressed by directly reducing the effects of established sources of SPH error (Sandnes et al., in preparation). This combines a range of novel methods with recent SPH developments, such as gradient estimates based on linear-order reproducing kernels (Frontiere, Raskin & Owen 2017). The treatment of mixing in simulations with either one or multiple equations of state is significantly improved both in

standard hydrodynamics tests such as Kelvin–Helmholtz instabilities and in planetary applications (Sandnes et al., in preparation).

Each of these modifications may be switched on and off in SWIFT in isolation. Further improvements are also in active development—including the implementation of additional features such as material strength models.

8.6 External potentials

Several external potentials intended for use in idealized simulations are implemented in SWIFT. The simplest external potentials include an unsoftened point mass, a softened point mass (i.e. a Plummer (1911) sphere), an isothermal sphere, a Navarro, Frenk & White (1997) (NFW) halo, and a constant gravitational field.

Besides these traditional options, SWIFT includes two Hernquist (1990) profiles that are matched to a NFW potential. The matching can be performed in one of two ways: (1) we demand that the mass within $R_{200, \text{cr}}$ is $M_{200, \text{cr}}^{33}$ for the Hernquist (1990) profile, i.e. $M_{\text{Hern}}(R_{\text{match}}) = M_{\text{NFW}}(R_{200, \text{cr}})$ at some specific matching radius. (2) We demand that the density profile in the centre is equivalent i.e. $\rho_{\text{Hern}}(r) = \rho_{\text{NFW}}(r)$ for $r \ll R_{200, \text{cr}}/c$, where c is the NFW concentration of the halo.

The first of these profiles follows Springel, Di Matteo & Hernquist (2005a) and uses $M_{\text{Hern}}(r \rightarrow \infty) = M_{\text{NFW}}(R_{200, \text{cr}}) = M_{200, \text{cr}}$ and $\rho_{\text{Hern}}(r) = \rho_{\text{NFW}}(r)$. Using this they can derive a matched scale factor with the assumption that $a/R_{200, \text{cr}} \ll 1$ of the halo given by $a = \sqrt{b}R_{200, \text{cr}}$ where

$$b = \frac{2}{c^2} \left(\ln(1+c) - \frac{c}{1+c} \right) \quad (82)$$

The second profile follows Nobels et al. (2023), who match $M_{\text{Hern}}(R_{200, \text{cr}}) = M_{\text{NFW}}(R_{200, \text{cr}})$, $\rho_{\text{Hern}}(r) = \rho_{\text{NFW}}(r)$ and do not assume a $a/R_{200, \text{cr}} \ll 1$. This gives a different Hernquist (1990) scale length and $M_{\text{Hern}}(R_{200, \text{cr}})$, producing a better match with the NFW profile. Both approaches are similar for haloes with large concentration parameters.

In order to reduce errors in the integration of orbits, each of the spherically-symmetric potentials optionally imposes a minimum time-step to each particle (see e.g. Nobels et al. 2022). We compute the distance from the centre r of each particle and the corresponding circular velocity $V_{\text{circ}}(r)$. We then impose a minimum time-step of $\Delta t_{\text{pot}} = \varepsilon_{\text{pot}} \frac{r}{V_{\text{circ}}(r)}$, where ε_{pot} is a free parameter typically defaulting to $\varepsilon_{\text{pot}} = 0.01$ (i.e. 100 time-steps per orbit).

9 IMPLEMENTATION DETAILS & PARALLELIZATION

In this Section, we present some of the important implementation details, especially surrounding the multi-node parallelism, and discuss the results of a scaling test on a realistic problem testing the entirety of the code modules.

9.1 Details of the cells & tasking system

The basic decomposition of the computational domain in meaningfully-sized cells was introduced in Section 2.1. We present some more technical details here.

³³ $M_{200, \text{cr}}$ is the mass within the radius $R_{200, \text{cr}}$, at which the average internal density $\langle \rho \rangle = 200 \rho_{\text{crit}}$, and ρ_{crit} is the critical density of the Universe.

In all the calculations we perform, we start by laying a Cartesian grid on top of the domain. This defines the most basic level in the cell hierarchy and is referred to as the top-level grid.³⁴ The size of this grid varies from about 8 cells on a side for small simple test runs to 64 elements for large calculations. In most cases, there will be many thousands or millions of particles per cell. We then use a standard oct-tree construction method to recursively split the cells into 8 children cells until we reach a number of particles per cell smaller than a set limit, typically 400. This leads to a relatively shallow tree when compared to other codes which create tree nodes (cells) down to a single particle, and implies a much smaller memory footprint for the tree itself than for other codes. As discussed in Section 2.1, SWIFT can perform interactions between cells of different size.

Once the tree has been fully constructed, we sort the particles into their cells. By using a depth-first ordering, we can guarantee that the particles occupy a contiguous section of memory for all the cells in the tree and at any level. This greatly helps streamline operations on single or pairs of cells as all the particles will simply be located between two known addresses in memory; no speculative walk will be necessary to find all the particles we need for a set of interactions. This sorting of particles can be relatively expensive on the very first step as we inherit whatever order the particles were listed in the initial conditions. However, in the subsequent constructions, this will be much cheaper because the particles only move by small amounts with respect to their cells in between constructions. This is also thanks to the relatively shallow tree we build, which permits for comparatively large cell sizes. For this reason, we use a *parallel merge sort* here to sort the particles in their cells as it is an efficient way to sort almost-sorted lists, which is the case in all but the first step. Recall also that we do not need to sort the particles very finely, thanks to the high number of them we accept in tree leaves. Whilst this operation is technically a sort, we refer to it as binning of the particles in what follows to avoid confusion with the sorting of particles on the interaction axis used by the pseudo-Verlet algorithm.

With the tree constructed and the particles all in their cell hierarchies, we have all the information required to decide which cells will need to interact for SPH (based on the cells' maximum smoothing lengths) and for gravity (based on the multipoles). All the quantities required for this decision making were gathered while binning the particles. We start by constructing the tasks on the top-level grid only, as described in Section 2.2 and Section 4.3 for SPH and gravity respectively. In most non-trivial cases, however, this will lead to tasks with very large numbers of particles and hence a large amount of work to perform. If there are only a few expensive tasks, then the scheduler will not be able to load-balance the work optimally as its options are limited. We ideally want significantly more tasks to be enqueued and waiting for execution than there are compute cores. It is hence key to fine-grain the problem further. To achieve this, we attempt to split the tasks into smaller units. For instance, a task acting on a single cell might be split into eight tasks, each acting on its eight children cells independently. For some tasks, in particular when there are no particle-particle interactions involved, this is trivially done (e.g. time integration or for a cooling sub-grid model) but other tasks may lead to more complex scenarios. An SPH task for instance cannot be split into smaller tasks if the smoothing length of the particles is larger than the size of the children cells. In most non-pathological cases, however, the tasks can be moved down the tree by several levels,

³⁴Note that this grid is not related to the one used for the periodic gravity calculation (Section 4.5). It is, however, the base grid used to retrieve particles efficiently in small sections of the snapshots (Section 6.2).

thus multiplying their overall number many times over and ultimately satisfying our request to have many more tasks than computing units. In cases where more than one loop over the neighbours are needed, only the tasks corresponding to the first loop are moved down the tree levels by assessing whether refinement criteria are met. The tasks corresponding to the subsequent interaction loops however are created by duplicating the already existing tasks of the first loop. As an example, the SPH *force* loop is created by copying all the tasks needed for the *density* loop and relabelling them. Similarly, all the sub-grid feedback or black hole-related loops are created in this fashion. This approach has the advantage of keeping the task-creation code as simple as possible. While duplicating the loops, we also set dependencies between tasks to impose the logical order of operations between them (see Fig. 4).

With the tasks created, the code is ready to perform many time-steps. That is, we can re-use the infrastructure created above until the geometrical conditions are violated by particle movement. For SPH, these conditions would be too large a change in smoothing length or a particle moving too far out of its cell meaning that the assumption that all the neighbours are in the same cell or any directly adjacent one is broken. For gravity, this would be too large a particle movement, leading to it being impossible to recompute multipoles without changing the cell geometry. Our shallow tree with large leaves has the advantage of remaining valid for many steps. We also note that other criteria (such as a global mesh gravity step or a certain number of particle updates leading to a tree rebuild) do, in practice, trigger a tree and tasks construction more often than these.

At the start of each step, we perform a quick tree walk starting, in parallel, in each of the many top-level cells. In this walk, we simply identify which cells contain active particles (i.e. particles which need to be integrated forward in time on this step) and activate the corresponding tasks. This operation is very rapid (much less than 1 percent of the total runtime in production runs) and can easily be parallelized given the large number of cells present in a run. Once all the tasks have been activated, they are handed over to the `QuickSched` engine which will launch them when ready.

As described by Gonnet et al. (2016), the tasks whose dependencies are all satisfied (i.e. for which all the tasks taking place earlier in the graph have already run) are placed in queues. We typically use one of these queues per thread and assign the tasks to the queues (and hence threads) either randomly or based on their physical location in the compute domain. The threads then run through their queues and attempt to fetch a task. When doing so they have to verify that the tasks they get are not conflicting with another, already-running operation. To this end, a mechanism of per-cell locks and semaphores is used. If a thread cannot acquire the lock on a given cell, it abandons this task and attempts to fetch the next one in the queue. If it can acquire a task, it will run the physics operations and upon completion will unlock all the dependencies associated with this task, hence enabling the next tasks to be placed in the queues. We highlight once more that the physics operations themselves are taking place inside a single thread and that no other thread can access the same data at the same time. This places the physics and maths operations taking place in a very safe space, allowing users with only limited programming experience to easily modify or extend the physics contained inside the tasks. No intimate knowledge of parallel programming or even of task-based parallelism is needed to alter the content of a task. If a thread reaches the end of its queue, it starts again from the beginning until there are no more tasks it can process. When that happens, the thread will attempt to steal work from the other threads' queues, a unique feature, at the time this project started, of the `QuickSched` library. Once all tasks in all queues have been processed, the time-

step has been completed and the threads are paused until the start of the next step.

9.2 Multi-node strategy

The top-level grid described in the previous section serves as the base decomposition unit of the simulated domain. When decomposing the problem into multiple domains, which would be necessary to run a simulation over multiple compute nodes, we assign a certain number of these cells to each of them. The tree construction algorithm is then run in parallel in each domain for each cell. The only addition is the possible exchange of particles which have left their domain entirely. They are sent to their new region and placed in the appropriate cells.

With the tree fully constructed, we send the sections of the trees (the cell geometry information and multipoles, not the particles) that border a domain to the nodes on the other side of the divide. Each compute node has henceforth full knowledge of its own trees and of any of the directly adjacent ones. With that information in hand, each node will be able to construct all of its tasks, as described above. It will do so for all the purely local cells as well as for the pair tasks operating on one local cell and one foreign cell. The compute node on the other side of the divide will create the exact same task as it bases its decision-making on exactly the same information. The only remaining operation is the creation of send and receive tasks for each task pair overlapping with a domain edge. By adding the appropriate dependencies, we create a task graph similar to the one depicted in Fig. 8.

With this logic, any task spanning a pair of cells that belong to the same partition needs only to be evaluated on that rank/partition, whilst tasks spanning more than one partition need to be evaluated on both ranks/partitions. This is done in the shallow tree walk that performs the task activation at the start of a step. A minor optimization can be used in the cases where only one of the two cells in a pair task contains active particles. In that situation, we can skip the sending and receiving of data to the node hosting the inactive cell since it will not be using it for any local updates.

All the tasks are put in queues in exactly the same way as in the single-node case. The only difference applies to the communication tasks. These are treated slightly differently. As soon as their dependencies are satisfied, the data is sent *asynchronously*. Similarly, as soon as the receiving node is ready, it will post a call to an asynchronous receive operation. Note that these communication tasks are treated like any other task; in particular, any of the threads can act on them and thus perform the inter-node communications. We then use the conflict mechanism of the queues to ask the MPI communication library whether the data has effectively been sent or received, respectively. Once that has happened, we simply unlock the corresponding tasks' dependencies and the received data can safely be used from that point onward. This allows us to effectively hide all the communications in the background and perform local work while the data move. We also note that once the data have arrived, nothing distinguishes them from data that were always on that node. This means that the physics operations in tasks can be agnostic of which data they work on. There is no need for special treatment when dealing with remote data; once more helping developers of physics modules to focus on the equations they implement rather than on the technicalities of distributed parallelism.

9.3 Domain decomposition

When running a large simulation over MPI using many ranks, an important question is how to share the workload across all the

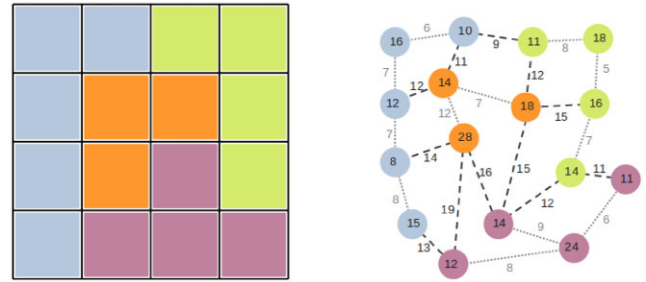


Figure 19. The representation of the top-level cells as a graph to be split over domains. The cells of the grid (on the left) correspond to the vertices of the graph (on the right), while the tasks spanning two cells constitute its edges (dashed and dotted lines). For simplicity, we consider here a 4×4 non-periodic grid in 2D and only show the pair tasks for cells that share an edge. Each vertex and graph edge has a weight associated with it, shown here as the numbers on each vertex and edge. The weights correspond to the cost of the task execution. If a pair operation is taking place over the network (shown here using dashed lines), its cost will be increased since communications will have to take place and the task will be executed on both of the involved ranks. The domain decomposition algorithm splits the graph so that the work (vertices and edges) is as evenly distributed as possible among all computing ranks (the four colours), minimizing the total cost by creating as few communications as possible. In the case shown here, this corresponds to the domain decomposition presented on the left. Note in particular that the number of cells assigned to each domain may not necessarily be the same.

ranks and their host compute nodes. This is important, beyond the obvious reasons like limited memory and CPU cores per node, as the progression of a simulation with synchronization points is determined by the slowest part.

The simulation workload consists of not just particles and their memory, but also the associated computation, which can vary depending on the types of particles, the current state and environment of the particles, as well as the costs of inter-node communication. All these elements play their part.

A representation of the workload and communication can be constructed by considering the hyper-graph of all top-level cells, where graph vertices represent cells and the edges represent the connections to the nearest neighbours (so each vertex has up to 26 edges). In this graph the vertices represent the computation done by the cell's tasks and the edges represent only the computation done in pair-interaction tasks. This follows since pair interactions are the only ones that could involve non-local data, so the computation in tasks spanning an edge should be related to the communication needed. Now, any partition of this graph represents a partition of the computation and communication, i.e. the graph nodes belonging to each partition will belong to an MPI rank, and the data belonging to each cell resides on the rank to which it was assigned. Such a decomposition is shown in Fig. 19 for a simple toy example.

The weighting of the vertices and edges now needs to reflect the actual work and time expected to be used for communication. Initially, the only knowledge we have of the necessary weights is the association of particles and cells, so we only have vertex weights. However, when a simulation is running, every task is timed to CPU tick accuracy and thus has a direct wall-clock measurement to reflect the computation. This will never be perfect, as other effects like interruptions from other processes will add time, but should be good enough. Note that it also naturally accounts for unknowns, like CPU speed and compiler optimizations, that a non-timed system would need to know about for all the different task types. So, once all the

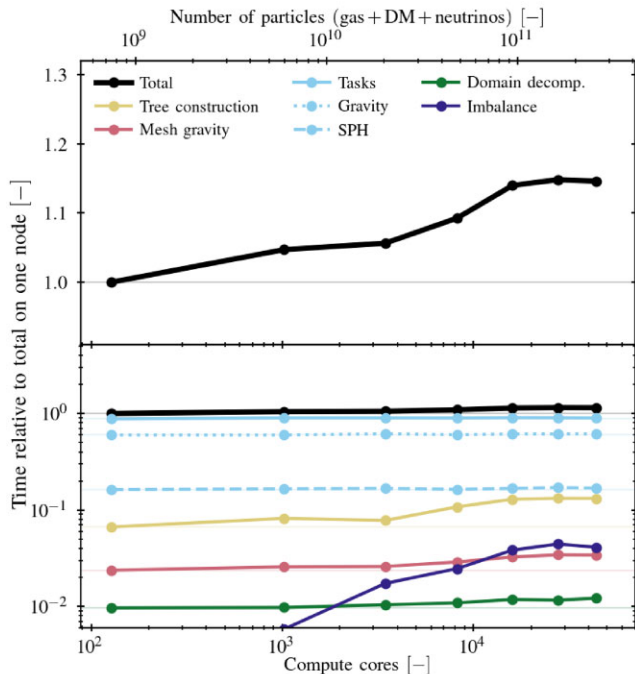


Figure 20. Weak-scaling performance of the SWIFT code on a representative cosmological simulation test problem. We use a 400^3 Mpc 3 volume extracted from the FLAMINGO series with 720^3 baryon, 720^3 dark matter, and 144^3 neutrino particles at $z = 1$. That base unit is then replicated periodically in all three directions; the top-level grid, as well as the gravity mesh, are also scaled alongside the replications. The number of compute nodes is grown proportionally, starting from a single node (128 cores) for the base volume. The top axis indicates the total number of particles used in each of the tests. When scaling the problem by a factor $7^3 = 343$, the total runtime (black line) increases by only 15 per cent, as shown on the top panel (note the linear y-axis). The bottom panel shows the breakdown of the total time in different categories (note the log y-axis). The time spent in the tasks (aka. actually solving physics equations, blue line) is remarkably constant as the problem size increases. The task time can be further subdivided in gravity (the FMM part) and SPH operations (dotted and dashed lines); all other tasks, including the sub-grid operations, correspond to a negligible fraction of the runtime. The ‘mesh gravity’ category corresponds to all the operations performed by the PM-part of the algorithm. The loss of performance is dominated by the lack of scalability of some operations within the tree construction (yellow) as well as by the accumulation of residual imbalance between nodes (purple). The domain decomposition itself (green) only requires a negligible amount of time.

tasks of a simulation have run, we then know how long they take and can then use these real-world weights in the graph.

Decomposing such graphs is a standard problem in computer science and multiple packages exist in the literature. We chose to use *Metis* and *ParMetis* (Karypis & Kumar 1998).

Using this simple weights scheme is sufficient, as shown in the next section. Note also that we are not demanding a perfect partition of the graph. In typical simulations, the workload evolves with time (which task times naturally take into account), and it is hence counterproductive to spend a large amount of time identifying the perfect partition. We prefer to use a partition that is good enough but quick to obtain. For realistic simulations, we find that we can maintain the imbalance between compute domains to less than 10 per cent (see also Schaller et al. 2016, and Fig. 20 below). We caution that this approach does not explicitly consider any geometric constraints, nor does it attempt to distribute the data uniformly. The only criterion

is the relative computational cost of each domain, for which the task decomposition provides a convenient model. We are therefore partitioning the computation, as opposed to just the data. There could, in principle, be cases where the work-based decomposition leads to problematic data distributions leading to the code running out of memory on a given compute node. We have so far never encountered such a situation in practice.

In addition to this default mechanism, SWIFT also offers other domain decomposition algorithms. The first one just attempts to split the data evenly between the compute nodes, so maintains the initial state. This is similar to what other simulation packages do, though here it is based on the top-level cells. This is also used as a backup mechanism in case the work-based decomposition leads to too much data imbalance. Finally, a mode where the top-level grid is simply split into regular chunks is also implemented. This is never recommended but the code will default to this if the *Metis* library is not available.

9.4 Scaling results AND code performance

The scaling performance of the SWIFT code on various test problems has been reported in different publications thus far. We give a quick overview here and complement it with a test exploiting the full cosmological simulation engine in a realistic scenario.

In their original SWIFT feasibility study, Schaller et al. (2016) analysed the original SPH-only code’s performance on cosmological test boxes. They reported a strong-scaling efficiency of 60 per cent when scaling a problem from 512 cores to 131 072 cores of a BlueGene system. This demonstrated the viability of the task-based approach combined with a graph-based domain decomposition mechanism and set the foundation for the current version of the code.

In their analysis, Borrow et al. (2018) took low-redshift cosmological simulations from the EAGLE suite and ran strong- and weak-scaling tests of the code. They focused on the scaling of the SPH operations by running only the hydrodynamics tasks. However, by using late-time cosmological boxes, they analysed the performance of the code with a realistic density (and hence time-step) distribution. They demonstrated the importance of running the drift operation only on the region of the volumes that directly contribute to the calculation.

Finally, Rogers et al. (2022) analysed the performance of SWIFT in the context of future exa-scale developments with engineering-type SPH applications in mind. To this end, they ran a fixed time-step, fairly uniform, test volume with more than 5.5×10^{11} gas particles and demonstrated excellent weak-scaling performance up to the size of their test cluster ($\approx 50\,000$ cores).

To complement these earlier tests, we present here a scaling test exploiting all the main physics modules, including a galaxy formation model. To be as representative as possible, we use a $z = 1$ setup such that the density structure and hence time-step hierarchy is well developed. We use a 400^3 Mpc 3 volume with 720^3 baryon, 720^3 dark matter, and 144^3 neutrino particles extracted from the FLAMINGO (Schaye et al. 2023) suite and run it for 1024 time-steps. The sub-grid model is broadly similar to the one described in Section 8.1 but with parameters calibrated to match observational data sets at a lower resolution than EAGLE did (for details, see Kugel et al. 2023). We use this volume as a base unit and run it on a single node (128 cores) of the *cosma-8* system.³⁵ We use 4 MPI ranks per node

³⁵The *cosma-8* system is run by DiRAC (www.dirac.ac.uk) and hosted by the University of Durham, UK. The system is made of 360 compute nodes with 1 TB RAM and dual 64-core AMD EPYC 7H12 at 2.6 GHz (4 NUMA

even when running on a single node to include the MPI overheads also in the smallest run. The 4 MPI ranks are distributed over the various NUMA regions of the node. We then scale up the problem by replicating the box periodically along the three axes and increasing the number of nodes proportionally. We also use 8 top-level cells per unit volume and an FFT gravity mesh of size 512^3 . Both are scaled up when increasing the problem size. We increase the problem size by a factor $7^3 = 343$, which corresponds to the largest setup we can fit on the system. The results of this test are shown in Fig. 20, where we plot the time to solution in units of the time taken on one node. Perfect weak-scaling hence corresponds to horizontal lines. When the problem size is increased by a factor 343, the performance loss is only 15 per cent. We also decompose the time spent in the main code sections. The tasks (i.e. physics operations, blue line) dominate the run time and display an excellent scaling performance. Decomposing the task work into the gravity and SPH parts, we see that gravity is the dominant component, validating the hydrodynamics-first approach of the overall code design. All other operations, including all of the sub-grid model tasks, are a negligible contribution to the total. The loss of performance when scaling up comes from the tree construction (orange) and from the overall imbalance between the different nodes (purple) due to an imperfect domain decomposition leading to slightly non-uniform work-load between the nodes despite the problem being theoretically identical. As discussed in Section 9.3, we can maintain the node-to-node imbalance below 10 per cent. We also report that the time spent deciding how to distribute the domains and performing the corresponding exchange of particles (green line) is a negligible fraction of the total runtime.

Finally, we note that the right-most points in Fig. 20 correspond to a test as large as the largest cosmological hydrodynamical simulation (by particle number) ever run to $z = 0$ (the flagship 2×5040^3 FLAMINGO volume of Schaye et al. 2023), demonstrating SWIFT's capability to tackle the largest problems of interest to the community.

We started the presentation of the design decisions that lead to the architecture of SWIFT in Section 2 by a brief discussion of the performance of the previous generation of cosmological hydrodynamical simulations and in particular of the EAGLE suite. To demonstrate improvements we could have repeated the flagship simulation of Schaye et al. (2015) with SWIFT using our updated SPH implementation and the EAGLE-like model of Section 8.1. Even with SWIFT's enhanced performance, this would still be a large commitment of resources for a benchmarking exercise, so we decided to instead compare the time taken by the codes on a smaller simulation volume using the same model. The $(25 \text{ Mpc})^3$ volume run with 2×376^3 particles presented in Section 8.1.5 took 159 hours using 28 compute cores of the *cosma-7* system³⁶; this corresponds to a total of 4452 CPU core hours. The GADGET-based run, using the same initial conditions, from the original EAGLE suite took 32 900 CPU core hours, meaning that our software is $>7 \times$ faster on that problem. Recall however, that the flavours of SPH and the implementation of the sub-grid models are different

regions / CPU) with AVX2 vector capability. The interconnect is Mellanox HDR, 200Gbit/s, with a non-blocking fat-tree topology. The machine has a theoretical 1.9 PF peak performance and achieved 1.3 PF on the standard HPL benchmark.

³⁶The *cosma-7* system is run by DiRAC (www.dirac.ac.uk) and hosted by the University of Durham, UK. The system is made of 448 compute nodes with 512 GB RAM and dual 14-core Intel Xeon Gold 5120 CPU at 2.2 GHz (1 NUMA region/CPU) with AVX512 vector capability. The interconnect is Mellanox EDR, 100Gbit/s, using a fat tree topology with a 2:1 blocking configuration.

from the original EAGLE code making a more detailed comparison difficult.

We also note that this SWIFT-based EAGLE-like run only required 92 GB of memory meaning that it would easily fit in the memory of a single compute node of most modern facilities. By contrast, the GADGET-based EAGLE run required 345 GB of memory; a factor of nearly 4x more.

9.5 Random number generator

Many extensions of the base solvers, in particular sub-grid models for galaxy formation, make use of (pseudo-)random numbers in their algorithms. Examples of this are stochastic star formation models or feedback processes (see Sections 8.1.2 and 8.1.3 for such models in SWIFT). Simulation packages can generate random numbers in various ways, often based on direct calls to a generator such as the base one part of UNIX or the more advanced ones in GSL (Gough 2009). To speed things up or to make the sequence independent of the number of MPI nodes, these calls can then be bundled into tables and regenerated every so often. The particles and physics modules then access these tables to retrieve a random number. This approach can lead to different issues of reproducibility between runs if the particles or modules are not calling the generator in the same order. These issues can arise due to task ordering choices.³⁷ Additionally, when bundling random numbers in small tables, great care has to be taken to make sure the indexing mechanism is sufficiently uniform so as to not bias the results.³⁸

In SWIFT, despite the intrinsic lack of ordering of the operations due to the tasking, we decided to avoid these pitfalls by viewing the generation of random numbers as a hashing of four unique quantities which are then used to construct the mantissa of a number in the interval $[0,1)$. We combine the ID of the particle (64-bit), the current location on the integer timeline (64-bit), a unique identifier for this random process (64-bit), and a general seed (16-bit). By doing so, we always get the same random number for a given particle at the same point in simulation time. Since each process also gets a unique identifier, we can draw uncorrelated numbers between modules for the same particle in the same step. Finally, the global seed can be altered if one wanted to actually change the whole sequence to study the effect of a particular set of randoms (see Borrow et al. 2023, for an example using SWIFT and the EAGLE-like model). The combined 144 bits thus generated are passed through a succession of XOR and random generator seed evolution functions to create a final source of entropy. We use this source as a seed for our last UNIX random number call, `erand48()`, whose output bits are interpreted as the mantissa of our result.

We have thoroughly verified that this entire mechanism generates perfectly uniform numbers. We also verified that there is no correlation between calls using the same particle and time-step but varying the identifier of the random process.

³⁷Note that in MPI codes, the same order-of-operations-issue can also occur if rounding choices change the time-step size of a particle, thus altering the sequence of numbers. The ordering of operations is not guaranteed for reduction operations, or in the directly SWIFT-relevant case, for asynchronous communications in a multi-threaded environment, unless the developers implemented explicit mechanisms to force this (often slower) behaviour.

³⁸A common mistake is to index the tables based on particle IDs when these IDs themselves encode some information (e.g. only even numbers for gas, or a position in the ICs).

10 SUMMARY AND CONCLUSION

10.1 Summary

In this paper, we have presented the algorithms and numerical methods exploited in the open-source astrophysical code, SWIFT. We have presented various test problems performed with the code, as well as demonstrated its scaling capability to reach the largest problems targeted by the community. In addition, we described the sub-grid models and other features made available alongside the code, and the various output strategies allowing the users to make the most efficient use of their simulations.

The core design strategy of the SWIFT code was to focus on a hydrodynamics-first approach, with a gravity solver added on top. In tandem with this, the parallelization strategy departs from traditional methods by exploiting a task-based parallelism method with dependencies and conflicts. This allows for the efficient load-balancing of problems by letting the runtime scheduler dynamically shift work between the different compute units. This approach, coupled to a domain decomposition method focusing on distributing work and not data, is specifically designed to adhere to the best practices for efficient use of modern hardware.

Various modern flavours of SPHs are implemented, alongside two sets of flexible sub-grid models for galaxy formation, a modern way of evolving cosmological neutrinos, and extensions to handle planetary simulations. These additional components are presented and released publicly along with the base code.

Besides testing and benchmarking (in simulations using more than 2×10^{12} particles), the SWIFT software package has already been exploited to perform extremely challenging scientific calculations. These include the very large dark-matter-only ‘zoom-in’ ($>10^{11}$ particles in the high resolution region) of the SIBELIUS project (McAlpine et al. 2022), the large cosmological hydrodynamics runs (up to 2×5040^3 particles) of the FLAMINGO project (Schaye et al. 2023), and the highest ever resolution Moon-formation simulations (Kegerreis et al. 2022). We envision that the public release of the code and its future developments will lead to more projects adopting it as their backbone solver for the most difficult and largest numerical astrophysics and cosmology problems.

10.2 Future developments

The SWIFT code is in constant development and we expect it to evolve considerably in the future. This paper describes the first full public release of the software and we expect improvements to the numerical aspects to be made, new models to be added, as well as new computer architectures to be targeted in the future.

One of the current grand challenges in high-performance computing is the jump towards so-called exa-scale systems. It is widely believed that such computing power can only be reached via the use of accelerators such as GPUs. This is a challenge for methods such as SPH and generally for algorithms including deep time-step hierarchies due to the low arithmetic intensity of these methods and the use of largely irregular memory access patterns. In the context of SWIFT, exploiting efficiently both CPUs and GPUs via a unified tasking approach is an additional challenge. Some avenues and possible solutions are discussed by Bower, Rogers & Schaller (2022), where some early work porting specific computationally-intensive tasks to GPUs is also described.

In terms of physics models, we expect the public code to be soon expanded to include the self-interacting dark matter model of Correa et al. (2022). This will expand the range of cosmological models that can be probed with the SWIFT package. Work on other extensions

beyond vanilla Λ CDM will likely follow. Similarly, additional sub-grid models for galaxy formation and cosmological applications are in the process of being included in the main code base and will be released in the future.

The code is also being expanded to include material strength models, as well as further new equations of state, for planetary and other applications.

The various hydrodynamics solvers in the code are currently all variations of SPH. This family of methods is known to have some limitations in the rate of convergence towards analytic solutions in certain scenarios. In future releases of the SWIFT package, we thus intend to supplement this with additional SPH variations (e.g. Rosswog 2020), renormalized mesh-free methods (e.g. Vila 1999; Hopkins 2015; Alonso Asensio et al. 2023), and a moving mesh implementation akin to Vandenbroucke & De Rijcke (2016). These methods all use unstructured particles with neighbourhoods as their base algorithmic tool, which makes them very suitable to fit within the framework currently existing in the SWIFT code. Developments on top of the SPH flavours to include magneto-hydrodynamics terms are also under way both using a direct induction formulation (e.g. Price et al. 2018) and a vector-potential formulation (e.g. Stasyszyn & Elstner 2015).

The code is also being expanded to include radiative transfer modules, starting with the SPH-based formalism of Chan et al. (2021) based on the M1-closure method and a coupling to the CHIMES non-equilibrium thermo-chemical solver (Richings, Schaye & Oppenheimer 2014a, b). Developments to include sub-cycling steps, in an even deeper hierarchy than in the gravity + hydro case (Duncan et al. 1998), for the exchange of photons are also on-going, which coupled to the task-based approach embraced by SWIFT should lead to significant gains over more classic methods (Ivkovic 2023).

Finally, an improved domain decomposition strategy for the special case of zoom-in simulations with high-resolution regions small compared to the parent box but too large to find in a single node’s memory will be introduced by Roper et al. (in preparation) [; see also chapter 2 of Roper (2023) for a preliminary discussion].

By publicly releasing the code and its extensions to the community, we also hope to encourage external contributors to share their models built on top of the version described here to other researchers by themselves making their work public.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the significant contribution to this project and paper that the late Richard G. Bower made over the years. His unbounded enthusiasm and immense expertise as well as his mentorship and guidance will be sorely missed.

We are indebted to the support and admin staff running the DiRAC COSMA facility at Durham, in particular to Lydia Heck and Alastair Basden. Their trust in the project, as well as their help running, debugging, scaling, and testing our code on the machines at scale, have been invaluable.

We thank Joop Schaye for the long-term support, the detailed discussions on physics modelling, and the guidance this kind of large project requires. Adopting SWIFT early on for multiple large projects has also been crucial to bring the code to its current mature stage. We thank Carlos Frenk for support and motivation in the early stages of this project.

We gratefully acknowledge useful discussions with Edoardo Altamura, Andres Arámburo-García, Stefan Arridge, Zorry Belcheva, Alejandro Benítez-Llambay, Heinrich Bockhorst, Alexei Borissov, Peter Boyle, Joey Braspenning, Jemima Briggs, Florian Cabot, Shaun Cole, Rob Crain, Claudio Dalla Vecchia, Massimiliano Culpio,

Vincent Eke, Pascal Elahi, Azadeh Fattahi, Johnathan Frawley, Victor Forouhar, Daniel Giles, Cameron Grove, Oliver Hahn, Patrick Hirling, Fabien Jeanquartier, Adrian Jenkins, Sarah Johnston, Orestis Karapiperis, Ashley Kelly, Euthymios Kotsialos, Roi Kugel, Claudia Lagos, Angus Lepper, Bàrbara Levering, Aaron Ludlow, Ian McCarthy, Abouzied Nasar, Rüdiger Pakmor, John Pennycook, Oliver Perks, Joel Pfeffer, Chris Power, Daniel Price, Lieuwe de Regt, John Regan, Alex Richings, Darwin Roudit, Chris Rowe, Jaime Salcido, Nikyta Shchutskyi, Volker Springel, Joachim Stadel, Federico Stasyszyn, Luís Teodoro, Tom Theuns, Rodrigo Tobar, James Trayford, and Tobias Weinzierl.

This work used the DiRAC@Durham facility managed by the Institute for Computational Cosmology on behalf of the STFC DiRAC HPC Facility (www.dirac.ac.uk). The equipment was funded by BEIS capital funding via STFC capital grants ST/K00042X/1, ST/P002293/1, ST/R002371/1 and ST/S002502/1, Durham University and STFC operations grant ST/R000832/1. DiRAC is part of the National e-Infrastructure. This work is supported by INTEL through the establishment of the Institute for Computational Cosmology as an INTEL parallel computing centre (IPCC). We acknowledge research software engineering support for this project from the STFC DiRAC High Performance Computing Facility which helped port the code on different architectures and performed thorough benchmarking. This work was supported by the Swiss Federal Institute of Technology in Lausanne (EPFL) through the use of the facilities of its Scientific IT and Application Support Center (SCITAS) and the University of Geneva through the usage of Yggdrasil. MS acknowledges support from NWO under Veni grant number 639.041.749. PD is supported by STFC consolidated grant ST/T000244/1. MI has been supported by EPSRC's Excalibur programme through its cross-cutting project EX20-9 *Exposing Parallelism: Task Parallelism* (Grant ESA 10 CDEL) and the DDWG project *PAX-HPC* (Gant EP/W026775/1). YMB acknowledges support from NWO under Veni grant number 639.041.751. EC is supported by funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 860744 (BiD4BEST). TKC is supported by the E. Margaret Burbidge Prize Postdoctoral Fellowship from the Brinson Foundation at the Departments of Astronomy and Astrophysics at the University of Chicago. CC acknowledges the support of the Dutch Research Council (NWO Veni 192.020). FH is supported by the STFC grant ST/P006744/1. JAK acknowledges support from a NASA Postdoctoral Program Fellowship. SP acknowledges support by the Austrian Science Fund (FWF) grant number V 982-N.S. TDS is supported by STFC grants ST/T506047/1 and ST/V506643/1. WJR acknowledges funding from Sussex STFC Consolidated Grant (ST/X001040/1).

DATA AVAILABILITY

The entirety of the software package presented in this paper, including all the extensions and many examples, is fully publicly available. It can be found alongside an extensive documentation on the website of the project: www.swiftsim.com.

REFERENCES

- Abramowitz M., Stegun I. A., 1965, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. US Government printing office, Washington
- Adamek J., Daverio D., Durrer R., Kunz M., 2016, *J. Cosmol. Astropart. Phys.*, 2016, 053
- Agertz O. et al., 2007, *MNRAS*, 380, 963
- Ali-Haimoud Y., Bird S., 2013, *MNRAS*, 428, 3375
- Almgren A. S., Bell J. B., Lijewski M. J., Lukic Z., Van Andel E., 2013, *ApJ*, 765, 39
- Alonso Asensio I., Dalla Vecchia C., Potter D., Stadel J., 2023, *MNRAS*, 519, 300
- Altamura E., Kay S. T., Bower R. G., Schaller M., Bahé Y. M., Schaye J., Borrow J., Towler I., 2023, *MNRAS*, 520, 3164
- Angulo R. E., Hahn O., 2022, *Liv. Rev. Comput. Astrophys.*, 8, 1
- Aubert D., Teyssier R., 2010, *ApJ*, 724, 244
- Augonnet C., Thibault S., Namyst R., Wacrenier P.-A., 2011, *Concurrency Comput.*, 23, 187
- Bagla J. S., 2002, *JA&A*, 23, 185
- Bagla J. S., Ray S., 2003, *New Astron.*, 8, 665
- Bahé Y. M. et al., 2022, *MNRAS*, 516, 167
- Balsara D. S., 1989, PhD thesis, Univ. Illinois at Urbana-Champaign
- Balsara D. S., 1995, *J. Comput. Phys.*, 121, 357
- Barnes J., Hut P., 1986, *Nature*, 324, 446
- Bate M. R., Burkert A., 1997, *MNRAS*, 288, 1060
- Blumofe R. D., Joerg C. F., Kuszmaul B. C., Leiserson C. E., Randall K. H., Zhou Y., 1995, *ACM SIGPLAN Notices*, 30, 207
- Boehm B., 2000, *Software Cost Estimation with Cocomo II*. Vol. 1, Prentice Hall
- Bondi H., 1952, *MNRAS*, 112, 195
- Booth C. M., Schaye J., 2009, *MNRAS*, 398, 53
- Borrow J., Borrisov A., 2020, *J. Open Source Softw.*, 5, 2430
- Borrow J., Bower R. G., Draper P. W., Gonnet P., Schaller M., 2018, in *Proc. 13th SPHERIC International Workshop*. p. 44
- Borrow J., Schaller M., Bower R. G., 2021, *MNRAS*, 505, 2316
- Borrow J., Schaller M., Bower R. G., Schaye J., 2022, *MNRAS*, 511, 2367
- Borrow J., Schaller M., Bahé Y. M., Schaye J., Ludlow A. D., Ploeckinger S., Nobels F. S. J., Altamura E., 2023, *MNRAS*, 526, 2441
- Bower R., Rogers B. D., Schaller M., 2022, *Comput. Sci. Eng.*, 24, 14
- Braspenning J., Schaye J., Borrow J., Schaller M., 2023, *MNRAS*, 523, 1280
- Bryan G. L. et al., 2014, *ApJS*, 211, 19
- Chabrier G., Debras F., 2021, *ApJ*, 917, 4
- Chaikin E., Schaye J., Schaller M., Bahé Y. M., Nobels F. S. J., Ploeckinger S., 2022, *MNRAS*, 514, 249
- Chaikin E., Schaye J., Schaller M., Benítez-Llambay A., Nobels F. S. J., Ploeckinger S., 2023, *MNRAS*, 523, 3709
- Chan T. K., Theuns T., Bower R., Frenk C., 2021, *MNRAS*, 505, 5784
- Cheng H., Greengard L., Rokhlin V., 1999, *J. Comput. Phys.*, 155, 468
- Colombi S., Jaffe A., Novikov D., Pichon C., 2009, *MNRAS*, 393, 511
- Correa C. A., Schaller M., Ploeckinger S., Anau Montel N., Weniger C., Ando S., 2022, *MNRAS*, 517, 3045
- Couchman H. M. P., Thomas P. A., Pearce F. R., 1995, *ApJ*, 452, 797
- Crain R. A., van de Voort F., 2023, *ARA&A*, 61, 473
- Crain R. A. et al., 2015, *MNRAS*, 450, 1937
- Creasey P., 2018, *Astron. Comput.*, 25, 159
- Croton D. J., 2013, *Publ. Astron. Soc. Aust.*, 30, e052
- Cullen L., Dehnen W., 2010, *MNRAS*, 408, 669
- Dalla Vecchia C., Schaye J., 2012, *MNRAS*, 426, 140
- Davé R., Dubinski J., Hernquist L., 1997, *New Astron.*, 2, 277
- Davis M., Efstathiou G., Frenk C. S., White S. D. M., 1985, *ApJ*, 292, 371
- Dehnen W., 2000, *ApJ*, 536, L39
- Dehnen W., 2001, *MNRAS*, 324, 273
- Dehnen W., 2002, *J. Comput. Phys.*, 179, 27
- Dehnen W., 2014, *Comput. Astrophys. Cosmol.*, 1, 1
- Dehnen W., Aly H., 2012, *MNRAS*, 425, 1068
- Dehnen W., Read J. I., 2011, *Eur. Phys. J. Plus*, 126, 55
- Driver S. P. et al., 2022, *MNRAS*, 513, 439
- Duncan M. J., Levison H. F., Lee M. H., 1998, *AJ*, 116, 2067
- Durier F., Dalla Vecchia C., 2012, *MNRAS*, 419, 465
- Elahi P. J., Thacker R. J., Widrow L. M., 2011, *MNRAS*, 418, 320
- Elahi P. J., Cañas R., Poulton R. J. J., Tobar R. J., Willis J. S., Lagos C. d. P., Power C., Robotham A. S. G., 2019, *PASA*, 36, 21
- Elbers W., 2022, *J. Cosmol. Astropart. Phys.*, 2022, 058
- Elbers W., Frenk C. S., Jenkins A., Li B., Pascoli S., 2021, *MNRAS*, 507, 2614

- Ewald P. P., 1921, *Ann. Phys.*, 369, 253
- Faber N. T., Stibbe D., Portegies Zwart S., McMillan S. L. W., Boily C. M., 2010, *MNRAS*, 401, 1898
- Faucher-Giguère C.-A., 2020, *MNRAS*, 493, 1614
- Ferland G. J., Korista K. T., Verner D. A., Ferguson J. W., Kingdon J. B., Verner E. M., 1998, *PASP*, 110, 761
- Ferland G. J. et al., 2017, *Rev. Mex. Astron. Astrophys.*, 53, 385
- Frigo M., Johnson S. G., 2005, *Proc. IEEE*, 93, 216
- Frontiere N., Raskin C. D., Owen J. M., 2017, *J. Comput. Phys.*, 332, 160
- Fryxell B. et al., 2000, *ApJS*, 131, 273
- Gaburov E., Nitadori K., 2011, *MNRAS*, 414, 129
- Galler B. A., Fisher M. J., 1964, *Commun. ACM*, 7, 301
- Garrison L. H., Eisenstein D. J., Pinto P. A., 2019, *MNRAS*, 485, 3370
- Garrison L. H., Eisenstein D. J., Ferrer D., Maksimova N. A., Pinto P. A., 2021, *MNRAS*, 508, 575
- Garzilli A., Magalich A., Theuns T., Frenk C. S., Weniger C., Ruchayskiy O., Boyarsky A., 2019, *MNRAS*, 489, 3456
- Gingold R. A., Monaghan J. J., 1977, *MNRAS*, 181, 375
- Goldbaum N. J., Zuhone J. A., Turk M. J., Kowalik K., Rosen A. L., 2018, *J. Open Source Softw.*, 3, 809
- Gonnet P., 2013, *Mol. Simulation*, 39, 721
- Gonnet P., 2015, *SIAM J. Sci. Comput.*, 37, C95
- Gonnet P., Chalk A. B. G., Schaller M., 2016, preprint (arXiv:1601.05384)
- Górski K. M., Hivon E., Banday A. J., Wandelt B. D., Hansen F. K., Reinecke M., Bartelmann M., 2005, *ApJ*, 622, 759
- Gough B., 2009, GNU Scientific Library Reference Manual—Third Edition, 3rd edn. Network Theory Ltd
- de Graaff A., Trayford J., Franx M., Schaller M., Schaye J., van der Wel A., 2022, *MNRAS*, 511, 2544
- Greengard L., Rokhlin V., 1987, *J. Comput. Phys.*, 73, 325
- Greif T. H., Glover S. C. O., Bromm V., Klessen R. S., 2009, *MNRAS*, 392, 1381
- Grove C. et al., 2022, *MNRAS*, 515, 1854
- Haardt F., Madau P., 2001, in Neumann D. M., Tran J. T. V., eds, Clusters of Galaxies and the High Redshift Universe Observed in X-rays. p. 64
- Haardt F., Madau P., 2012, *ApJ*, 746, 125
- Habib S. et al., 2016, *New Astron.*, 42, 49
- Hahn O., Rampf C., Uhlemann C., 2021, *MNRAS*, 503, 426
- Harnois-Déraps J., Pen U.-L., Iliev I. T., Merz H., Emberson J. D., Desjacques V., 2013, *MNRAS*, 436, 540
- Harvey D., Revaz Y., Robertson A., Hausammann L., 2018, *MNRAS*, 481, L89
- Hausammann L., 2021, Ecole Polytechnique Fédérale de Lausanne, PhD thesis, Lausanne
- Hausammann L., Revaz Y., Jablonka P., 2019, *A&A*, 624, A11
- Hausammann L., Gonnet P., Schaller M., 2022, *Astron. Comput.*, 41, 100659
- Heitmann K. et al., 2008, *Comput. Sci. Discov.*, 1, 015003
- Hernquist L., 1990, *ApJ*, 356, 359
- Hernquist L., Katz N., 1989, *ApJS*, 70, 419
- Hietel D., Junk M., Keck R., Teleaga D., 2001, in Proc. GAMM Workshop ‘Discrete Modelling and Discrete Algorithms in Continuum Mechanics’. p. 10
- Hietel D., Junk M., Kuhnert J., Tiwari S., 2005, *Analysis and Numerics for Conservation Laws (G. Warnecke Edt.)*, p. 339
- Hockney R. W., Eastwood J. W., 1988, Computer simulation using particles. CRC Press
- Hopkins P. F., 2013, *MNRAS*, 428, 2840
- Hopkins P. F., 2015, *MNRAS*, 450, 53
- Hopkins P. F., Quataert E., Murray N., 2011, *MNRAS*, 417, 950
- Hopkins P. F., Nadler E. O., Grudić M. Y., Shen X., Sands I., Jiang F., 2023, *MNRAS*, 525, 5951
- Hubber D. A., Batty C. P., McLeod A., Whitworth A. P., 2011, *A&A*, 529, A27
- Huško F., Lacey C. G., 2023, *MNRAS*, 520, 5090
- Huško F., Lacey C. G., Schaye J., Schaller M., Nobels F. S. J., 2022, *MNRAS*, 516, 3750
- Huško F., Lacey C. G., Schaye J., Nobels F. S. J., Schaller M., 2024, *MNRAS*, 527, 5988
- Ishiyama T., Nitadori K., Makino J., 2012, in *SC’12: Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*. p. 1
- Ivanova N. et al., 2013, *Astron. Astrophys. Rev.*, 21, 59
- Ivkovic M., 2023, PhD thesis, EPFL
- Jenkins A. et al., 1998, *ApJ*, 499, 20
- Karypis G., Kumar V., 1998, *SIAM J. Sci. Comput.*, 20, 359
- Katz N., 1992, *ApJ*, 391, 502
- Katz N., Weinberg D. H., Hernquist L., 1996, *ApJS*, 105, 19
- Kegerreis J. A., Eke V. R., Gonnet P., Korycansky D. G., Massey R. J., Schaller M., Teodoro L. F. A., 2019, *MNRAS*, 487, 5029
- Kegerreis J. A., Ruiz-Bonilla S., Eke V. R., Massey R. J., Sandnes T. D., Teodoro L. F. A., 2022, *ApJ*, 937, L40
- Kennicutt Robert C. J., 1998, *ApJ*, 498, 541
- Kim J.-h. et al., 2016, *ApJ*, 833, 202
- Klessen R., 1997, *MNRAS*, 292, 11
- Knebe A. et al., 2013, *MNRAS*, 435, 1618
- Kravtsov A. V., Klypin A. A., Khokhlov A. M., 1997, *ApJS*, 111, 73
- Kugel R. et al., 2023, *MNRAS*, 526, 6103
- Lesgourgues J., Pastor S., 2006, *Phys. Rep.*, 429, 307
- Lesgourgues J., Tram T., 2011, *J. Cosmol. Astropart. Phys.*, 2011, 032
- Linder E. V., Jenkins A., 2003, *MNRAS*, 346, 573
- Lucy L. B., 1977, *AJ*, 82, 1013
- Ludlow A. D., Schaye J., Bower R., 2019, *MNRAS*, 488, 3663
- Mangano G., Miele G., Pastor S., Pinto T., Pisanti O., Serpico P. D., 2005, *Nucl. Phys. B*, 729, 221
- McAlpine S. et al., 2022, *MNRAS*, 512, 5823
- Menon H., Wesolowski L., Zheng G., Jetley P., Kale L., Quinn T., Governato F., 2015, *Comput. Astrophys. Cosmol.*, 2, 1
- Message Passing Interface Forum, 2021, MPI: A Message-Passing Interface Standard Version 4.0.
- Michaux M., Hahn O., Rampf C., Angulo R. E., 2021, *MNRAS*, 500, 663
- Mignone A., Zanni C., Tzeferacos P., van Straalen B., Colella P., Bodo G., 2012, *ApJS*, 198, 7
- Monaghan J. J., 1992, *ARA&A*, 30, 543
- Monaghan J. J., Lattanzio J. C., 1985, *A&A*, 149, 135
- Monaghan J. J., Price D. J., 2001, *MNRAS*, 328, 381
- Morris J. P., Monaghan J. J., 1997, *J. Comput. Phys.*, 136, 41
- Naab T., Ostriker J. P., 2017, *ARA&A*, 55, 59
- Narayan R., Yi I., 1994, *ApJ*, 428, L13
- Navarro J. F., Frenk C. S., White S. D. M., 1997, *ApJ*, 490, 493
- Nelson R. P., Papaloizou J. C. B., 1994, *MNRAS*, 270, 1
- Nobels F. S. J., Schaye J., Schaller M., Bahé Y. M., Chaikin E., 2022, *MNRAS*, 515, 4838
- Nobels F. S. J., Schaye J., Schaller M., Ploekinger S., Chaikin E., Richings A. J., 2023, preprint (arXiv:2309.13750)
- Okamoto T., Eke V. R., Frenk C. S., Jenkins A., 2005, *MNRAS*, 363, 1299
- Owen J. M., Villumsen J. V., 1997, *ApJ*, 481, 1
- Peebles P. J. E., 1980, The large-scale structure of the universe. Princeton Univ. Press
- Perez J. M., Badia R. M., Labarta J., 2008, in IEEE International Conference on Cluster Computing. p. 142
- Ploekinger S., Schaye J., 2020, *MNRAS*, 497, 4857
- Plummer H. C., 1911, *MNRAS*, 71, 460
- Portegies Zwart S., 2020, *Nat. Astron.*, 4, 819
- Potter D., Stadel J., Teyssier R., 2017, *Comput. Astrophys. Cosmol.*, 4, 2
- Power C., Navarro J. F., Jenkins A., Frenk C. S., White S. D. M., Springel V., Stadel J., Quinn T., 2003, *MNRAS*, 338, 14
- Price D. J., 2008, *J. Comput. Phys.*, 227, 10040
- Price D. J., 2012, *J. Comput. Phys.*, 231, 759
- Price D. J., Monaghan J. J., 2007, *MNRAS*, 374, 1347
- Price D. J. et al., 2018, *Publ. Astron. Soc. Aust.*, 35, e031
- Quinn T., Katz N., Stadel J., Lake G., 1997, preprint (arXiv:astro-ph/9710043)
- Rahmati A., Schaye J., Pawlik A. H., Raičević M., 2013, *MNRAS*, 431, 2261
- Ramsey J. P., Haugbølle T., Nordlund Å., 2018, in *Journal of Physics Conference Series*. p. 012021
- Regan J. A., Haehnelt M. G., Viel M., 2007, *MNRAS*, 374, 196
- Rein H., Tamayo D., 2017, *MNRAS*, 467, 2377

- Reinders J., 2007, Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media
- Revaz Y., 2013, Astrophysics Source Code Library, record ascl:1302.004
- Revaz Y., Jablonka P., 2012, *A&A*, 538, A82
- Revaz Y., Jablonka P., 2018, *A&A*, 616, A96
- Revaz Y., Arnaudon A., Nichols M., Bonvin V., Jablonka P., 2016, *A&A*, 588, A21
- Richings A. J., Schaye J., Oppenheimer B. D., 2014a, *MNRAS*, 440, 3349
- Richings A. J., Schaye J., Oppenheimer B. D., 2014b, *MNRAS*, 442, 2780
- Robertson B. E., Kravtsov A. V., 2008, *ApJ*, 680, 1083
- Roca-Fàbrega S. et al., 2021, *ApJ*, 917, 64
- Rogers B. et al., 2022, in G. Bilotta, ed., Proc. 16th SPHERIC International Workshop. p. 391
- Roper W., 2023, PhD thesis, University of Sussex, UK
- Rosas-Guevara Y. M. et al., 2015, *MNRAS*, 454, 1038
- Rosswog S., 2020, *MNRAS*, 498, 4230
- Ruiz-Bonilla S., Eke V. R., Kegerreis J. A., Massey R. J., Teodoro L. F. A., 2021, *MNRAS*, 500, 2861
- Ruiz-Bonilla S., Borrow J., Eke V. R., Kegerreis J. A., Massey R. J., Sandnes T. D., Teodoro L. F. A., 2022, *MNRAS*, 512, 4660
- Saitoh T. R., Makino J., 2009, *ApJ*, 697, L99
- Salmon J. K., Warren M. S., 1994, *J. Comput. Phys.*, 111, 136
- Sanati M., Revaz Y., Schober J., Kunze K. E., Jablonka P., 2020, *A&A*, 643, A54
- Sanati M., Jeanquartier F., Revaz Y., Jablonka P., 2023, *A&A*, 669, A94
- Schaller M., Dalla Vecchia C., Schaye J., Bower R. G., Theuns T., Crain R. A., Furlong M., McCarthy I. G., 2015, *MNRAS*, 454, 2277
- Schaller M., Gonnet P., Chalk A. B. G., Draper P. W., 2016, in Proc. PASC Conference. PASC'16. ACM, New York, NY
- Schaye J., 2004, *ApJ*, 609, 667
- Schaye J., Dalla Vecchia C., 2008, *MNRAS*, 383, 1210
- Schaye J., Aguirre A., Kim T.-S., Theuns T., Rauch M., Sargent W. L. W., 2003, *ApJ*, 596, 768
- Schaye J. et al., 2015, *MNRAS*, 446, 521
- Schaye J. et al., 2023, *MNRAS*, 526, 4978
- Schmidt M., 1959, *ApJ*, 129, 243
- Schneider A. et al., 2016, *J. Cosmol. Astropart. Phys.*, 2016, 047
- Sembolini F. et al., 2016, *MNRAS*, 457, 4063
- Shakura N. I., Sunyaev R. A., 1973, *A&A*, 24, 337
- Smith B. D. et al., 2017, *MNRAS*, 466, 2217
- Somerville R. S., Davé R., 2015, *ARA&A*, 53, 51
- Springel V., 2005, *MNRAS*, 364, 1105
- Springel V., 2010a, *ARA&A*, 48, 391
- Springel V., 2010b, *MNRAS*, 401, 791
- Springel V., Hernquist L., 2002, *MNRAS*, 333, 649
- Springel V., Hernquist L., 2003, *MNRAS*, 339, 289
- Springel V., Yoshida N., White S. D. M., 2001, *New Astron.*, 6, 79
- Springel V., Di Matteo T., Hernquist L., 2005a, *MNRAS*, 361, 776
- Springel V. et al., 2005b, *Nature*, 435, 629
- Springel V., Pakmor R., Zier O., Reinecke M., 2021, *MNRAS*, 506, 2871
- Stasyszyn F. A., Elstner D., 2015, *J. Comput. Phys.*, 282, 148
- Stevens A. R. H., Bellstedt S., Elahi P. J., Murphy M. T., 2020, *Nat. Astron.*, 4, 843
- Stewart S. et al., 2020, in *American Institute of Physics Conference Series*. p. 080003
- Stinson G., Seth A., Katz N., Wadsley J., Governato F., Quinn T., 2006, *MNRAS*, 373, 1074
- Stone J. M., Tomida K., White C. J., Felker K. G., 2020, *ApJS*, 249, 4
- Tepper-García T., Richter P., Schaye J., Booth C. M., Dalla Vecchia C., Theuns T., Wiersma R. P. C., 2011, *MNRAS*, 413, 190
- Teyssier R., 2002, *A&A*, 385, 337
- The HDF Group, 1997-2022, Hierarchical Data Format, version 5, <https://www.hdfgroup.org/HDF5/>
- Tillotson J. H., 1962, General Atomic Report, GA-3216, 141
- Tinker J. L., Robertson B. E., Kravtsov A. V., Klypin A., Warren M. S., Yepes G., Gottlöber S., 2010, *ApJ*, 724, 878
- Tornatore L., Borgani S., Dolag K., Matteucci F., 2007, *MNRAS*, 382, 1050
- Trayford J. W. et al., 2015, *MNRAS*, 452, 2879
- Truelove J. K., Klein R. I., McKee C. F., Holliman John H. I., Howell L. H., Greenough J. A., 1997, *ApJ*, 489, L179
- Turk M. J., Smith B. D., Oishi J. S., Skory S., Skillman S. W., Abel T., Norman M. L., 2011, *ApJS*, 192, 9
- Vandenbroucke B., De Rijcke S., 2016, *Astron. Comput.*, 16, 109
- Verlet L., 1967, *Phys. Rev.*, 159, 98
- Viel M., Haehnelt M. G., Springel V., 2004, *MNRAS*, 354, 684
- Vila J. P., 1999, *Math. Models Methods Appl. Sci.*, 09, 161
- Vogelsberger M., Marinacci F., Torrey P., Puchwein E., 2020, *Nature Rev. Phys.*, 2, 42
- Wadsley J. W., Stadel J., Quinn T., 2004, *New Astron.*, 9, 137
- Wadsley J. W., Veeravalli G., Couchman H. M. P., 2008, *MNRAS*, 387, 427
- Wadsley J. W., Keller B. W., Quinn T. R., 2017, *MNRAS*, 471, 2357
- Warren M. S., 2013, in Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC'13). ACM, New York, NY
- Warren M. S., Salmon J. K., 1995, *Comput. Phys. Commun.*, 87, 266
- Wendland H., 1995, *Adv. Comput. Math.*, 4, 389
- Wiersma R. P. C., Schaye J., Smith B. D., 2009a, *MNRAS*, 393, 99
- Wiersma R. P. C., Schaye J., Theuns T., Dalla Vecchia C., Tornatore L., 2009b, *MNRAS*, 399, 574
- Willis J. S., Schaller M., Gonnet P., Bower R. G., Draper P. W., 2018, in *Parallel Computing is Everywhere*. IOS Press, p. 507
- Willis J. S., Schaller M., Gonnet P., Helly J. C., 2020, in *Parallel Computing: Technology Trends*. IOS Press, p. 263
- Wright E. L., 2006, *PASP*, 118, 1711
- Xu G., 1995, *ApJS*, 98, 355
- Zennaro M., Bel J., Villaescusa-Navarro F., Carbone C., Sefusatti E., Guzzo L., 2017, *MNRAS*, 466, 3244

APPENDIX A: ADDITIONAL SPH SCHEMES

For completeness, we summarize here the equations of motion for the the additional modern SPH schemes present in SWIFT. These are re-implementation of schemes from the literature and can be used to perform comparisons between models in a framework where the rest of the solver's infrastructure is kept exactly fixed.

A1 Pressure-smoothed SPH

Pressure-smoothed SPH solves the same generic equation of motion as described in equation 10, but with a different choice of fundamental variables a and b . In general, instead of smoothing the density $\hat{\rho}$, we introduce a smoothed pressure \hat{P} that is generated through loops over neighbours (as described below). This approach is commonplace in astrophysics, with it described and used in Saitoh & Makino (2013), Hopkins (2013), and Hu et al. (2014), among others.

For the two choices of thermodynamic variable, internal energy (per unit mass) u , or entropy A , we generate two different (but equivalent) smoothed pressures,

$$\hat{P}_i = (\gamma - 1) \sum_j m_j u_j W_{ij}, \quad (\text{A1})$$

$$\hat{P}_i = \left[\sum_j m_j A_j^{1/\gamma} W_{ij} \right]^\gamma, \quad (\text{A2})$$

respectively. As described by Borrow et al. (2021), this then leads to issues integrating the pressure in simulations with multiple time-stepping, especially in scenarios where there is a high \dot{u} (for instance in the presence of a strong cooling term in the sub-grid physics), as we should use

$$\frac{d\hat{P}_i}{dt} = (\gamma - 1) \sum_j m_j \left(W_{ij} \frac{du_j}{dt} + u_j \mathbf{v}_{ij} \cdot \nabla_j W_{ij} \right) \quad (\text{A3})$$

for the evolution of \hat{P}_i , which would formally require an extra loop over the neighbours. As such, we do not recommend these schemes for practical use, but we implement them in SWIFT for cross-compatibility with the original GADGET-based EAGLE code.

The changes in the smoothed variable give rise to a different equation of motion,

$$\frac{d\mathbf{v}_i}{dt} = -u_i(\gamma - 1)^2 \sum_j m_j u_j \left[\frac{f_{ij}}{\hat{P}_i} \nabla_i W_{ij} + \frac{f_{ji}}{\hat{P}_j} \nabla_j W_{ji} \right], \quad (\text{A4})$$

shown for the internal energy variant (Pressure–Energy) only for brevity.³⁹ The factors f_{ij} read

$$f_{ij} = 1 - \frac{1}{m_j u_j} \left[\frac{\partial \hat{P}_i}{\partial h_i} \frac{h_i}{(\gamma - 1) n_d \hat{n}_i} \right] \left[1 + \frac{h_i}{n_d \hat{n}_i} \frac{\partial \hat{n}_i}{\partial h_i} \right]^{-1} \quad (\text{A5})$$

As, in practice, we do not make an additional loop over neighbours to calculate the derivative in the smoothed pressure, we use a simple chain rule,

$$\frac{d\hat{P}_i}{dt} = \rho_i \frac{du_i}{dt} + u_i \frac{d\rho_i}{dt}, \quad (\text{A6})$$

to integrate the smoothed pressure with time. This is commonplace among pressure-SPH schemes implemented in real codes, as it is impractical from a performance perspective to require an additional loop solely for the reconstruction of the smoothed pressure time differential.

There are base Pressure–Entropy and Pressure–Energy schemes available in SWIFT that use the same equations of motion for artificial viscosity as the Density-based schemes (equation 16).

A2 ANARCHY-SPH

In addition to these base schemes, we implement ‘ANARCHY-PU’, which is a Pressure–Energy-based variant of the original ANARCHY scheme used for EAGLE (see Schaller et al. 2015 and appendix A of Schaye et al. 2015) which used entropy as the thermodynamic variable to evolve. We reformulate the base equations of motions in terms of internal energy in SWIFT as described in the previous section.

ANARCHY-PU uses the same artificial viscosity implementation as SPHENIX (equations 22–26) but uses a slightly different value of decay length $\ell = 0.25$.

The artificial conduction differs more markedly. The base equation (equations 27 and 29) remain unchanged w.r.t SPHENIX but three of the ingredients are altered. First, ANARCHY-PU does not pressure-weight the contributions of both interacting particles and thus

$$\alpha_{ij} = \frac{\alpha_{c,i} + \alpha_{c,j}}{2}. \quad (\text{A7})$$

Secondly, the conduction velocity is changed to

$$v_{c,ij} = c_{s,i} + v_{c,j} + \mu_{ij}, \quad (\text{A8})$$

which is similar to the signal velocity entering viscosity but with the sign of μ reversed. Thirdly, the dimensionless constant β_c entering the time evolution of the conduction parameter (equation 29) is lowered to $\beta_c = 0.01$. This is because ANARCHY-PU uses a smoothed-pressure implementation and thus a lower amount of conduction is required.

³⁹Expanded derivations and definitions are available in the theory documentation provided with the SWIFT code.

Finally, the conduction limiter in strong shocks (equation 31) is not used. Our implementation is consistent with the original ANARCHY scheme.

A3 PHANTOM-like flavour

SWIFT includes a reduced, and slightly modified, version of the PHANTOM SPH scheme (Price et al. 2018). It employs the same Density–Energy SPH scheme as SPHENIX and also implements variable artificial conduction and viscosity parameters. At present, our implementation in SWIFT is hydrodynamics only, but an extension to include magnetohydrodynamical effects is planned for the future.

Our PHANTOM artificial viscosity implementation is the same as SPHENIX and ANARCHY, with $\ell = 0.25$. This differs slightly from the original PHANTOM description, where a modified version of the Balsara (1989) switch is also used. For artificial conduction, a fixed $\alpha_c = 1$ is used for all particles, effectively removing the need for equation 29. The conduction speed is given as

$$v_{c,i} = \sqrt{2 \frac{|P_i - P_j|}{\hat{\rho}_i + \hat{\rho}_j}}, \quad (\text{A9})$$

with the PHANTOM implementation only designed for use with purely hydrodynamical simulations. Price et al. (2018) recommend a different conduction speed in simulations involving self-gravity.

A4 GASOLINE-2-like (GDF-like) flavour

SWIFT also includes a re-implementation of the equations of the GASOLINE-2 model presented by Wadsley et al. (2017). The implementation and default parameters follow the paper closely, though there are minor differences. We give the equations here for completeness but refer the reader to the original Wadsley et al. (2017) work for the motivation behind their derivation.

The equation of motion in Gasoline uses the so-called ‘Geometric Density Force’ (GDF) formulation, and is as follows:

$$\frac{d\mathbf{v}_i}{dt} = - \sum_j m_j \left(\frac{P_i + P_j}{\hat{\rho}_i \hat{\rho}_j} \right) \nabla_i \bar{W}_{ij}, \quad (\text{A10})$$

$$\frac{du_i}{dt} = \sum_j m_j \left(\frac{P_i}{\hat{\rho}_i \hat{\rho}_j} \right) \mathbf{v}_{ij} \cdot \nabla_i \bar{W}_{ij}, \quad (\text{A11})$$

where

$$\nabla_i \bar{W}_{ij} = \frac{1}{2} f_i \nabla_i W(r_{ij}, h_i) + \frac{1}{2} f_j \nabla_j W(r_{ij}, h_j), \quad (\text{A12})$$

is the symmetric average of both usual kernel contributions, and the variable smoothing length correction terms read:

$$f_i = \frac{\sum_j \frac{m_j}{\hat{\rho}_i} \mathbf{r}_{ij}^2 W_{ij}}{\sum_j \frac{m_j}{\hat{\rho}_j} \mathbf{r}_{ij}^2 W_{ij}}. \quad (\text{A13})$$

The artificial viscosity and conduction implementations use matrix calculations based on local pressure gradients. Here,

$$\nabla P_i = (\gamma - 1) \sum_j m_j u_j \nabla_i W_{ij}, \quad (\text{A14})$$

$$\mathbf{n}_i = \frac{\nabla P_i}{|\nabla P_i|}, \quad (\text{A15})$$

$$\frac{d\mathbf{v}_i}{dn_i} = \sum_{\alpha, \beta} \alpha_{i, \alpha} \mathbf{v}_{\alpha \beta, i} \mathbf{n}_{i, \beta}, \quad (\text{A16})$$

with the velocity gradient tensor

$$\mathbf{V}_{\alpha \beta, i} = \frac{\sum_j (\mathbf{v}_{\alpha i} - \mathbf{v}_{\alpha j}) (\mathbf{r}_{\beta i} - \mathbf{r}_{\beta j}) m_j W_{ij}}{\frac{1}{3} \sum_j \mathbf{r}_{ij}^2 m_j W_{ij}}, \quad (\text{A17})$$

and the shock detector

$$D_i = \frac{3}{2} \left[\frac{dv_i}{dn_i} + \max \left(-\frac{1}{3} \nabla \cdot \mathbf{v}_i, 0 \right) \right] \quad (\text{A18})$$

with α and β indices along the Cartesian axes in our case. These give rise to the evolution equation for the artificial viscosity parameter, which is evolved in a similar manner to ANARCHY, SPHENIX, and PHANTOM:

$$\alpha_{\text{v,loc},i} = \alpha_{\text{v,max}} \frac{A_i}{A_i + v_{\text{sig},i}^2} \quad (\text{A19})$$

$$A_i = 2h_i^2 B_i \max \left(-\frac{dD_i}{dt}, 0 \right) \quad (\text{A20})$$

$$\frac{d\alpha_i}{dt} = 0.2c_{s,i} (\alpha_{\text{v,loc},i} - \alpha_{\text{v},i}) / h_i. \quad (\text{A21})$$

We note that the SWIFT implementation again uses the Balsara (1989) switch (the B_i term) rather than the Cullen & Dehnen (2010) style limiter used in the original GASOLINE-2 paper.

Artificial conduction is implemented using the trace-free shear tensor,

$$\mathbf{S}_{\alpha,\beta,i}^2 = \frac{\mathbf{V}_{\alpha,\beta,i} + \mathbf{V}_{\beta,\alpha,i}}{2} - \frac{\delta_{\alpha,\beta} \nabla \cdot \mathbf{v}_i}{3}, \quad (\text{A22})$$

and the conduction parameter:

$$\alpha_{c,i} = C |\mathbf{S}| h_i^2, \quad (\text{A23})$$

$$|\mathbf{S}| = \sum_{\alpha,\beta} \mathbf{S}_{\alpha,\beta}^2, \quad (\text{A24})$$

with the fixed parameter $C = 0.03$. Note that unlike the other schemes $\alpha_{c,i}$ is not dimensionless. These then get added to the equation of motion for thermal energy using

$$\frac{du_i}{dt} = - \sum_j m_j \frac{(\alpha_{c,i} + \alpha_{c,j}) (u_i - u_j) (\mathbf{r}_{ij} \cdot \nabla_i \bar{W}_{ij})}{\frac{1}{2} (\rho_i + \rho_j) \mathbf{r}_{ij}^2}, \quad (\text{A25})$$

which is very similar to the other schemes presented above.

APPENDIX B: MULTI-INDEX NOTATION

Following Dehnen (2014), we define a multi-index \mathbf{n} as a triplet of non-negative integers:

$$\mathbf{n} \equiv (n_x, n_y, n_z), \quad n_i \in \mathbb{N}, \quad (\text{B1})$$

with a norm n given by

$$n = |\mathbf{n}| \equiv n_x + n_y + n_z. \quad (\text{B2})$$

We also define the exponentiation of a vector $\mathbf{r} = (r_x, r_y, r_z)$ by a multi-index \mathbf{n} as

$$\mathbf{r}^{\mathbf{n}} \equiv r_x^{n_x} \cdot r_y^{n_y} \cdot r_z^{n_z}, \quad (\text{B3})$$

which for a scalar α reduces to

$$\alpha^{\mathbf{n}} = \alpha^n. \quad (\text{B4})$$

Finally, the factorial of a multi-index is defined to be

$$\mathbf{n}! \equiv n_x! \cdot n_y! \cdot n_z!, \quad (\text{B5})$$

which leads to a simple expression for the binomial coefficients of two multi-indices entering Taylor expansions:

$$\binom{\mathbf{n}}{\mathbf{k}} = \binom{n_x}{k_x} \binom{n_y}{k_y} \binom{n_z}{k_z}. \quad (\text{B6})$$

When appearing as the index in a sum, a multi-index represents all values that the triplet can take up to a given norm. For instance, $\sum_{\mathbf{n}}^p$ indicates that the sum runs over all possible multi-indices whose norm is $\leq p$.

¹Lorentz Institute for Theoretical Physics, Leiden University, PO Box 9506, NL-2300 RA Leiden, the Netherlands

²Leiden Observatory, Leiden University, PO Box 9513, NL-2300 RA Leiden, the Netherlands

³Department of Physics and Astronomy, University of Pennsylvania, 209 South 33rd Street, Philadelphia, PA 19104, USA

⁴Department of Physics and Kavli Institute for Astrophysics and Space Research, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

⁵Institute for Computational Cosmology, Department of Physics, Durham University, South Road, Durham DH1 3LE, UK

⁶Laboratoire d'astrophysique, École Polytechnique Fédérale de Lausanne (EPFL), CH-1290 Sauverny, Switzerland

⁷Observatoire de Genève, Université de Genève, Chemin Pegasi 51, CH-1290 Versoix, Switzerland

⁸Department of Computer Science, Durham University, Upper Mountjoy Campus, Stockton Road, Durham, UK

⁹Department of Physics, University of Helsinki, Gustaf Hällströmin katu 2, FI-00014 Helsinki, Finland

¹⁰The Oskar Klein Centre, Department of Physics, Stockholm University, Albanova University Center, SE-106 91 Stockholm, Sweden

¹¹Sterrenkundig Observatorium, Universiteit Gent, Krijgslaan 281, B-9000 Gent, Belgium

¹²STFC Hartree Centre, Sci-Tech Daresbury, Warrington, WA4 4AD, UK

¹³Department of Physics, The Chinese University of Hong Kong, Shatin, Hong Kong, China

¹⁴Department of Astronomy and Astrophysics, The University of Chicago, Chicago, IL60637, USA

¹⁵Université Paris-Saclay, Université Paris Cité, CEA, CNRS, AIM, F-91191, Gif-sur-Yvette, France

¹⁶GRAPPA Institute, University of Amsterdam, Science Park 904, NL-1098 XH Amsterdam, the Netherlands

¹⁷Google AI Perception, Google Switzerland, CH-8002 Zurich, Switzerland

¹⁸ITS High Performance Computing, Eidgenössische Technische Hochschule Zürich, CH-8092 Zürich, Switzerland

¹⁹NASA Ames Research Center, Moffett Field, CA 94035, USA

²⁰Department of Astrophysics, University of Vienna, Türkenschanzstrasse 17, 1180 Vienna, Austria

²¹Astronomy Centre, University of Sussex, Falmer, Brighton BN1 9QH, UK

²²SciNet HPC Consortium, University of Toronto, Toronto, Ontario, ON M5G 1M1, Canada

²³Space Research and Planetary Sciences, Physikalisches Institut, University of Bern, Bern, 3012, Switzerland

²⁴Institute for Astronomy, University of Edinburgh, Royal Observatory, Blackford Hill, Edinburgh EH9 3HJ, UK

This paper has been typeset from a $\text{\TeX}/\text{\LaTeX}$ file prepared by the author.