

cuRAMSES: Scalable Domain Decomposition and CUDA-enabled AMR for Cosmological Simulations

Juhan Kim^{1*}

¹Center for Advanced Computation, Korea Institute for Advanced Study, 85 Hoegiro, Dongdaemun-gu, Seoul 02455, Republic of Korea

Accepted XXX. Received YYY; in original form ZZZ

ABSTRACT

We present cuRAMSES, a suite of algorithmic and implementation improvements to the RAMSES adaptive mesh refinement (AMR) cosmological simulation code that address the principal bottlenecks encountered in large-scale simulations: communication overhead, memory consumption, and solver efficiency. The central innovation is a recursive k -section domain decomposition that replaces the traditional Hilbert curve ordering with a hierarchical spatial partitioning, dramatically reducing the number of MPI messages per ghost-zone exchange and eliminating all MPI_ALLTOALL calls. A Morton key hash table for octree neighbour lookup removes one of the largest per-rank arrays in the original code, while on-demand allocation strategies for auxiliary arrays further reduce the memory footprint. A hybrid CPU/GPU dispatch model allows OMP threads to dynamically offload compute-intensive routines — including the Godunov solver, gravity force computation, and radiative cooling — to GPU streams at runtime, with automatic fallback to CPU execution when streams are unavailable. We also implement variable- N_{cpu} restart for both HDF5 and native binary output formats, removing the constraint that output files must be read with the same number of MPI ranks as were used to write them. All modifications preserve physical consistency, as verified by conservation-law diagnostics across extensive test suites.

Key words: methods: numerical – cosmology: simulations – hydrodynamics – software: development

1 INTRODUCTION

Cosmological hydrodynamic simulations play a central role in modern astrophysics, connecting the predictions of the Λ CDM paradigm to the observable properties of galaxies, the intergalactic medium, and the large-scale structure of the Universe. Over the past decade, a series of landmark galaxy formation simulations have advanced our understanding of cosmic structure: the Illustris and IllustrisTNG projects (Vogelsberger et al. 2014; Pillepich et al. 2018; Nelson et al. 2019) using the moving-mesh code AREPO (Springel 2010), the EAGLE simulation (Schaye et al. 2015) with the smoothed-particle hydrodynamics code GADGET (Springel 2005), the Horizon-AGN simulation (Dubois et al. 2014) and its high-resolution successor NewHorizon (Dubois et al. 2021) using the adaptive mesh refinement (AMR) code RAMSES (Teyssier 2002), and the FIRE project (Hopkins et al. 2018) with the meshless finite-mass code GIZMO (Hopkins 2015).

Among the numerical approaches employed by these simulations, AMR codes such as RAMSES and Enzo (Bryan et al. 2014) provide a particularly attractive framework: the computational mesh is refined only where the physics demands it, concentrating resources on collapsing haloes and star-forming regions while keeping the cost of smooth, low-density regions manageable.

A key design choice in parallel AMR codes is the domain decomposition strategy. Space-filling curves (SFCs), particularly the Hilbert (Peano–Hilbert) curve, have been widely adopted for this purpose (Warren & Salmon 1993; Springel 2005). The Hilbert curve maps the three-dimensional computational domain to a one-dimensional index, preserving spatial locality so that cells close in physical space remain close along the curve. This enables a simple and effective partitioning: the one-dimensional index range is divided into N_{cpu} contiguous segments, each assigned to an MPI rank. The resulting decomposition naturally produces compact subdomains with relatively small surface-to-volume ratios, minimising the ghost-zone boundary between neighbouring ranks.

However, scaling this approach to the regime of 10^{10} – 10^{11} particles and 10^4 – 10^5 MPI ranks reveals fundamental limitations. The one-dimensional nature of the SFC means that *every* rank may, in principle, border *any other* rank, forcing communication patterns that scale poorly with N_{cpu} . In the standard RAMSES implementation, ghost-zone exchange, grid and particle migration, and sink particle synchronisation all rely on MPI_ALLTOALL to communicate counts among all ranks, leading to $\mathcal{O}(N_{\text{cpu}}^2)$ message complexity and $\mathcal{O}(N_{\text{cpu}})$ per-rank buffer memory — a severe bottleneck when N_{cpu} exceeds $\sim 10^3$ (Teyssier 2002). Furthermore, the Hilbert ordering distributes load based on cell count alone, which becomes increasingly inadequate for cosmological simulations where the particle distribution is highly clustered: a cell hosting 10^4

* E-mail: kjhan@kias.re.kr

52 particles in a dense halo is far more expensive in memory than 114
 53 a void cell with zero particles, yet the standard load balancer 115
 54 treats them equally (Springel 2005). The problem is partic- 116
 55 ularly acute in cosmological zoom-in simulations (Dubois et 117
 56 al. 2021), where the high-resolution region occupies a small 118
 57 fraction of the total volume: the Hilbert curve concentrates 119
 58 nearly all refinement on a few ranks while the remaining ranks 120
 59 are left with low-resolution void cells, leading to severe load 121
 60 imbalance that worsens with increasing zoom factor.

61 Beyond the communication and load-balancing chal- 120
 62 lenges, several other bottlenecks arise. Large per-rank ar- 121
 63 rays scale linearly with N_{gridmax} : the neighbour-pointer ar- 122
 64 ray `nbor(N_{\text{gridmax}}, 6)` alone consumes $48 N_{\text{gridmax}}$ bytes 123
 65 (Teyssier 2002), and the Hilbert key array adds another 124
 66 $16 N_{\text{gridmax}}$ bytes (when compiled with QUADHILBERT), to- 125
 67 gether approaching 1 GB per rank for production configu- 126
 68 rations with $N_{\text{gridmax}} \sim 5 \text{ M}$. The multigrid Poisson solver 127
 69 (Guillet & Teyssier 2011) typically dominates runtime, its 128
 70 per-iteration cost driven by frequent ghost-zone exchanges 129
 71 and repeated hash table lookups for neighbour grids. Finally, 130
 72 RAMSES writes one file per MPI rank, so restarting with a 131
 73 different rank count is practically infeasible: the AMR tree 132
 74 structure — parent-child links, neighbour pointers, and per- 133
 75 rank communication tables — is tightly coupled to the origi- 134
 76 nal domain decomposition and cannot be reconstructed with- 135
 77 out re-reading and redistributing every grid from scratch.

78 The original RAMSES code relies exclusively on MPI for 136
 79 parallelism, assigning one MPI rank per processor core. To 137
 80 exploit the shared-memory bandwidth of modern multi-core 138
 81 nodes, hybrid MPI+OpenMP implementations have been de- 139
 82 veloped: the Horizon Run 5 simulation (Lee et al. 2021) em- 140
 83 ploys OMP-RAMSES, and the NewCluster zoom-in simula- 141
 84 tion (Han et al. 2026) employs RAMSES-yOMP, both adding 142
 85 OpenMP threading within each MPI rank for improved intra- 143
 86 node scalability. However, MPI+OpenMP alone still leaves 144
 87 the GPU compute capability of modern heterogeneous nodes 145
 88 untapped. With the emergence of exascale supercomputers 146
 89 whose floating-point throughput is dominated by GPU ac- 147
 90 celerators, a three-level MPI+OpenMP+CUDA parallelism 148
 91 is essential to fully utilise the available hardware.

92 In this paper we describe CURAMSES, a comprehensive 149
 93 set of modifications to RAMSES that addresses each of these 150
 94 challenges. We introduce a recursive k -section domain de- 151
 95 composition (Section 2) that replaces Hilbert ordering with 152
 96 a hierarchical multi-way spatial partitioning, enabling MPI 153
 97 exchange with $\mathcal{O}(\sum_i k_i)$ messages per operation. A Morton 154
 98 key hash table (Morton 1966) (Section 4) eliminates the 155
 99 `nbor` array entirely and, together with on-demand alloca- 156
 100 tion of redundant large arrays, saves over 1 GB of memory 157
 101 per rank. Algorithmic optimizations to the multigrid Pois- 158
 102 son solver (Section 5) reduce its share of total runtime from 159
 103 55 per cent to 39 per cent, while a spatial hash binning scheme 160
 104 (Section 6) accelerates Type II supernova and AGN feedback 161
 105 by orders of magnitude. We also implement variable- N_{cpu} 162
 106 restart for both HDF5 and binary formats (Section 7). GPU- 163
 107 accelerated solvers (Section 8) dynamically offload compute- 164
 108 intensive routines to GPU streams at runtime. Performance 165
 109 benchmarks are presented in Section 9, and we conclude in 166
 110 Section 10.

111 Throughout this paper, we use the notation of Teyssier 162
 112 (2002): N_{levelmax} is the maximum AMR level, N_{gridmax} is the 163
 113 maximum number of grids per rank, $\text{twotondim} = 2^{N_{\text{dim}}} = 8$ 164

is the number of cells per oct in three dimensions, and N_{cpu} is the total number of MPI ranks.

2 RECURSIVE K-SECTION DOMAIN DECOMPOSITION

2.1 Motivation

The standard RAMSES domain decomposition assigns cells to MPI ranks by sorting them along a Hilbert space-filling curve and partitioning the resulting one-dimensional index range into N_{cpu} contiguous segments. While this preserves spatial locality reasonably well, it has two significant drawbacks for large-scale runs. First, the ghost-zone exchange requires `MPI_ALLTOALL` to communicate emission/reception counts, followed by point-to-point messages to all ranks with non-zero counts; in the worst case, every rank communicates with every other rank, yielding $\mathcal{O}(N_{\text{cpu}}^2)$ total messages. Second, the Hilbert key computation requires a large per-rank array `hilbert_key(1:nCell)` of 16 bytes per cell (when compiled with QUADHILBERT), totalling $\sim 640 \text{ MB}$ at $N_{\text{gridmax}} = 5 \times 10^6$.

Our recursive k -section decomposition replaces the one-dimensional Hilbert partitioning with a recursive spatial partitioning in the original three-dimensional coordinate space. This produces a k -ary tree whose structure directly encodes the communication pattern, enabling hierarchical message routing that scales with the tree depth rather than the total number of ranks.

2.2 Hierarchical Partitioning of Spatial and Communication Domain

Given N_{cpu} MPI ranks, we first compute the prime factorization as

$$N_{\text{cpu}} = p_1^{m_1} \times p_2^{m_2} \times \cdots \times p_r^{m_r}, \quad p_1 > p_2 > \cdots > p_r. \quad (1)$$

The splitting sequence is then

$$\mathbf{k} = (\underbrace{p_1, \dots, p_1}_{m_1}, \underbrace{p_2, \dots, p_2}_{m_2}, \dots, \underbrace{p_r, \dots, p_r}_{m_r}), \quad (2)$$

yielding $L (= \sum_i m_i)$ levels in the tree, which encodes both the domain hierarchy and the communication pattern. The tree is a k -ary structure in which each *node* represents a contiguous group of MPI ranks sharing a spatial sub-domain: the *root* node at level 0 spans all N_{cpu} ranks, and each *leaf* node at level L corresponds to a single rank. At each level l , the domain of a node is split into k_l child nodes along the longest axis of the current bounding box. This longest-axis selection ensures roughly isotropic sub-domains, minimising the surface-to-volume ratio and hence the ghost-zone count.

For example, $N_{\text{cpu}} = 12 = 3 \times 2 \times 2$ produces the splitting sequence $(3, 2, 2)$ with $L = 3$ tree levels: the root is split into 3 slabs along the longest axis, each slab is bisected, and each half is bisected again, yielding 12 leaf nodes, one per rank. Figure 1 illustrates this progressive decomposition for $N_{\text{cpu}} = 12$, from the undivided domain through three successive levels of splitting, with the corresponding k -section tree shown beneath each panel.

The tree is stored as a set of arrays indexed by node identifier. For each internal node, the child indices for each of the k_l

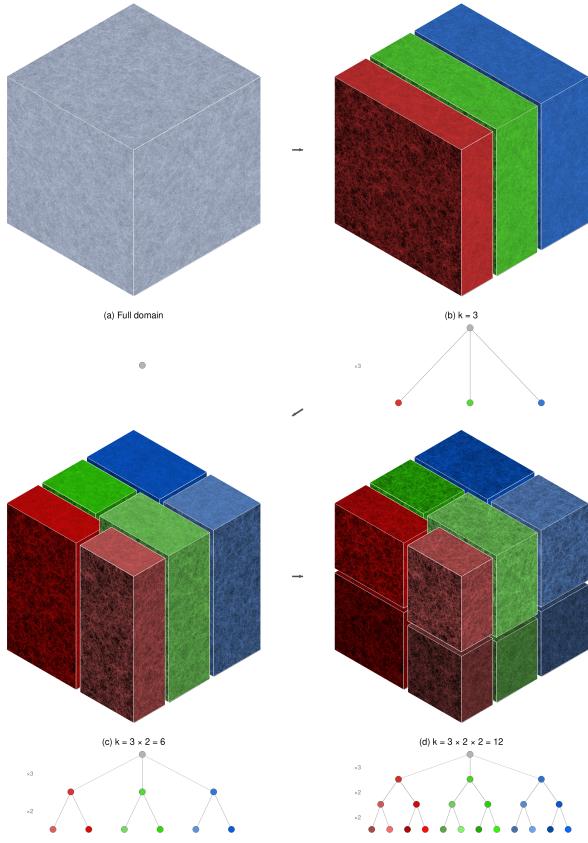


Figure 1. Progressive recursive k -section decomposition for $N_{\text{cpu}} = 12 = 3 \times 2 \times 2$. (a) The undivided simulation domain. (b) First split into $k = 3$ slabs along the longest axis. (c) Each slab bisected along the second axis ($3 \times 2 = 6$ sub-domains). (d) Final bisection along the third axis ($3 \times 2 \times 2 = 12$ leaf domains). Below each panel, the corresponding k -section tree is shown; colours encode x -slab membership (red/green/blue), with saturation and brightness indicating y and z subdivisions. Each face displays projected column density from a 4096^3 TSC density field. Sub-domain volumes vary by $\sim 10\text{--}30\%$, reflecting load-balanced wall placement.

partitions and the spatial coordinates of the partition boundaries are recorded; for each leaf node, the assigned MPI rank is stored. Every node also carries the minimum and maximum rank indices of all leaves in its subtree, enabling rapid range queries during the hierarchical exchange.

The total number of tree nodes, $N_{\text{nodes}} = 1 + \sum_{l=1}^L \prod_{i=1}^l k_i$, is at most a few hundred for practical values of N_{cpu} ($N_{\text{cpu}} \leq 10^5$), representing negligible overhead.

2.3 Load-Balanced Wall Placement

When the tree is updated during load balancing (every n_{remap} coarse steps), the wall positions are adjusted by iterative binary search so that each partition receives a load proportional to the number of ranks it contains.

The procedure operates level by level, top to bottom. At level l :

(i) A histogram is built over the splitting coordinate: for each node at level l , the cells within that node are projected

onto the splitting axis and binned into a cumulative cost histogram with resolution Δx_{hist} .

(ii) For each of the $k_l - 1$ walls within each node, a binary search adjusts the wall position until the cumulative load on the left side matches the target fraction. The target cumulative fraction for wall j in a node spanning ranks $[i_{\min}, i_{\max}]$ with total count $n = i_{\max} - i_{\min} + 1$ is

$$f_j = n^{-1} \sum_{m=1}^j n_m, \quad (3)$$

where n_m is the number of ranks assigned to partition m . Because n ranks cannot always be divided evenly among k_l partitions, $n_m = \lfloor n/k_l \rfloor + 1$ for the first $n \bmod k_l$ partitions (which absorb the remainder), and $n_m = \lfloor n/k_l \rfloor$ for the rest. Here $\lfloor \cdot \rfloor$ denotes the floor function (the largest integer not exceeding the argument).

(iii) An MPI_ALLREDUCE aggregates the local histograms across all N_{cpu} ranks to obtain the global cumulative load at each wall position. The binary search converges when the relative load imbalance $|\hat{L}_j - L_j^{\text{target}}|/L_j^{\text{target}}$ falls below a tolerance ϵ_{tol} (typically 1 per cent), or when the wall position can no longer be resolved at the histogram resolution.

(iv) After wall convergence, the cells are repartitioned (sorted) according to the new wall positions, and the histogram bounds are updated for the next level.

2.4 Memory-Weighted Cost Function

The default RAMSES load balancer weights all cells equally. cuRAMSES supports an optional memory-weighted cost function:

$$C_{\text{cell}} = 2^{-N_{\text{dim}}} (w_{\text{grid}} + n_{\text{part}}(\text{igrid}) \cdot w_{\text{part}}), \quad (4)$$

where w_{grid} is the memory cost per grid slot (default 270 bytes, accounting for hydro, gravity, and AMR bookkeeping arrays), w_{part} is the memory cost per particle slot (default 12 bytes for position, velocity, mass, and linked-list pointers), and $n_{\text{part}}(\text{igrid})$ is the number of particles attached to grid igrid . The division by $2^{N_{\text{dim}}}$ distributes the grid cost evenly among its eight cells.

This cost function ensures that ranks hosting dense haloes (many particles per cell) receive fewer cells, preventing memory exhaustion on particle-heavy ranks. All histogram loads are accumulated in 64-bit integers to avoid overflow when summing costs across millions of cells.

Activating memory-weighted balancing requires setting `memory_balance = .true.` and optionally tuning `mem_weight_grid` and `mem_weight_part` in the namelist. Our tests with 200 M particles on 12 ranks show that memory-weighted balancing reduces the peak-to-mean memory ratio from 2.5 to 1.3 without affecting physics results (identical e_{cons} , e_{pot} , e_{kin} to machine precision).

The parameter `nremp` controls the frequency of load-rebalancing operations (every `nremp` coarse steps). We changed the default from `nremp = 0` (rebalance every step) to `nremp = 5` based on systematic tests with 200 M particles on 12 ranks over 10 coarse steps:

The optimal value `nremp = 5` balances the cost of rebalancing against the growing imbalance that accumulates between rebalancing steps. Higher values (`nremp = 10`) reduce LB overhead further but allow imbalance to grow enough to

Table 1. Effect of `nremap` on total runtime and load-balance overhead. All configurations produce identical physics results ($e_{\text{cons}} = 3.77 \times 10^{-3}$ at step 10).

<code>nremap</code>	Total (s)	LB time (s)	LB fraction
1	303.8	64.4	21.2%
3	269.9	24.7	9.1%
5	249.8	15.7	6.3%
10	258.6	11.6	4.5%

236 slow other operations, resulting in a net increase in total run-
237 time.

238 To identify bottlenecks in the load-balancing procedure,
239 we added internal timing instrumentation that reports
240 the wall-clock time of each phase: (i) `numbp_sync` (MPI
241 synchronization of particle counts for virtual grids), (ii)
242 `cmp_new_cpu_map` (histogram construction and wall finding),
243 (iii) `expand_pass` (ghost-zone expansion after grid migra-
244 tion), (iv) `grid_migration` (actual grid transfer between
245 ranks), (v) `allreduce+cpumap_update` (global reduction and
246 CPU map reconstruction), and (vi) `shrink_pass` (removal
247 of migrated grids from source rank). Profiling reveals that
248 `allreduce+cpumap_update` dominates, consuming approxi-
249 mately 50 per cent of the total load-balancing time. This
250 motivates the `nremap` = 5 default, as reducing the frequency of
251 these expensive global operations has a disproportionate im-
252 pact on total runtime.

253 3 HIERARCHICAL MPI COMMUNICATION

254 The tree structure described in Section 2.2 enables a
255 hierarchical exchange protocol that replaces the global
256 `MPI_ALLTOALL` with a sequence of level-by-level correspond-
257 ent exchanges. We implement two variants.

258 3.1 Exclusive Exchange

259 In the exclusive exchange, each item has a unique destination
260 rank. The algorithm walks the k -section tree from root to
261 leaf, where each *node* represents a contiguous group of MPI
262 ranks at a given tree level. The *root* node encompasses all
263 N_{cpu} ranks and represents the entire computational domain.
264 At each level l , a node is partitioned into k_l child nodes; the
265 leaf nodes at the bottom of the tree correspond to individual
266 MPI ranks. Algorithm 1 summarises the procedure:

Algorithm 1 Exclusive hierarchical exchange

Input: Send buffer **S** with N items, destination ranks **d**
Output: Receive buffer **R** with items destined for this rank

```

1: W  $\leftarrow$  S; D  $\leftarrow$  d; node  $\leftarrow$  root
2: for  $l = 1$  to  $L$  do
3:    $k \leftarrow k_l$ ; my_child  $\leftarrow$  ksec.cpu.path(myid,  $l$ )
4:   Classify items in W by child index (counting sort on
   D)
5:   Identify  $k - 1$  correspondent ranks (one per sibling
   child)
6:   for each correspondent  $p$  do
7:     Exchange count: MPI_ISEND/IRecv (tag  $100 + l$ )
8:     Exchange data: MPI_ISEND/IRecv (tags  $200 + l$ ,  $300 +$ 
    $l$ )
9:   end for
10:  MPI_WAITALL
11:  W  $\leftarrow$  merge(my_child items, received items)
12:  node  $\leftarrow$  ksec.next(node, my_child)
13: end for
14: R  $\leftarrow$  W

```

At each level, each rank communicates with at most $k_l - 1$ correspondent ranks (one from each sibling subtree). The correspondent in a sibling subtree of size s is chosen as $\min(\text{my_pos}, s - 1)$ to distribute load evenly. The total number of messages per rank per exchange is

$$N_{\text{msg}} = \sum_{l=1}^L (k_l - 1) = \sum_i m_i(p_i - 1), \quad (5)$$

which for $N_{\text{cpu}} = 1024 = 2^{10}$ gives $N_{\text{msg}} = 10$ — two orders of magnitude fewer than the ~ 1024 messages required in the original all-to-all pattern.

Figure 2 illustrates the communication pattern for $N_{\text{cpu}} = 12$ ($= 3 \times 2 \times 2$). At level 1 ($k_1 = 3$), the 12 ranks are grouped into three children of four ranks each. Each rank exchanges data with one correspondent in each of the two sibling subtrees, yielding $k_1 - 1 = 2$ communication steps: step 1 pairs children 1 and 2 (e.g. rank 1 \leftrightarrow rank 5), while step 2 simultaneously pairs children 1 and 3 (dark red arcs, e.g. rank 1 \leftrightarrow rank 9) and children 2 and 3 (orange arcs, e.g. rank 5 \leftrightarrow rank 9). At level 2 ($k_2 = 2$), the scope narrows to within each group of four, with each rank contacting one correspondent two positions away (e.g. rank 1 \leftrightarrow rank 3). Finally, at level 3 ($k_3 = 2$), only adjacent pairs communicate (e.g. rank 1 \leftrightarrow rank 2). The progressively shorter arcs reflect the hierarchical narrowing of communication scope: long-range inter-group exchanges are resolved first, and successive levels refine the routing within ever-smaller subtrees. Each rank sends a total of $N_{\text{msg}} = 2 + 1 + 1 = 4$ messages per exchange, independent of N_{cpu} .

The pairing structure at each tree level follows directly from the branching factor. When a node has p children, data from every child must reach every other child. This is accomplished in $p - 1$ sequential steps. In step s ($s = 1, \dots, p - 1$), child $s + 1$ is paired with each of the children $1, \dots, s$: each rank in child $s + 1$ exchanges data with its correspondent in child c for $c = 1, \dots, s$, yielding s concurrent point-to-point exchanges. After step s , every child 1 through $s + 1$ holds the aggregate of all data originating from children 1 through $s + 1$. In particular, after all $p - 1$ steps, every child possesses

the complete data set of the entire subtree. For the $N_{\text{cpu}} = 12$ example in Figure 2, level 1 has $k_1 = 3$, giving $3-1 = 2$ steps: step 1 pairs child 2 with child 1 (1×4 exchanges), and step 2 pairs child 3 with both children 1 and 2 (2×4 exchanges). Levels 2 and 3 each have $k = 2$, requiring only $2-1 = 1$ step per level.

A trade-off of the hierarchical routing is that the total data volume transmitted per rank exceeds that of a direct MPI_ALLTOALL. In the all-to-all pattern, each rank sends each item exactly once to its destination. In the hierarchical exchange, an item may be forwarded through multiple tree levels before reaching its final destination, so the same data can traverse several hops. In the worst case, a rank’s entire send buffer is relayed at every level, giving a per-rank volume of at most $L \times V_{\text{local}}$, where V_{local} is the rank’s data size. However, this upper bound is rarely approached in practice because items are filtered by child index at each level: only the subset destined for a sibling subtree is actually transmitted, and successive levels operate on progressively smaller subsets. The key advantage, reducing the number of communication partners from $\mathcal{O}(N_{\text{cpu}})$ to $\mathcal{O}(\sum_l k_l)$, more than compensates for the modest increase in aggregate volume, particularly at large N_{cpu} where message startup latency dominates.

Working buffers are managed with Fortran 2003 `move_alloc` for zero-copy buffer swaps at each level, and per-level arrays (child counts, peer lists, MPI request handles) are pre-allocated with `save` attributes to eliminate allocation/deallocation overhead on repeated calls.

3.2 Ghost-Zone Exchange via K-Section

The ghost-zone (virtual boundary) exchange is the most communication-intensive operation in RAMSES, called multiple times per fine time step for hydro, gravity, and particle updates. We replace the standard all-to-all pattern with four k -section-based variants. First, a forward exchange sends data from emission grids to reception grids, and a corresponding reverse accumulation adds received values back into the emission grids. Because the hierarchical routing internally uses double-precision buffers, an integer variant packs integer data (e.g. `cpu_map`, `flag1`) into double-precision words before transport and unpacks them on receipt, reusing the same tree-walk machinery without a separate integer communication path.

The data packing format for each ghost grid is:

$$\text{sendbuf}(1 : 2^{N_{\text{dim}}} + 2, i) = \{u_1, \dots, u_{2^{N_{\text{dim}}}}, \text{sender_id}, \text{index}\}, \quad (6)$$

where u_j are the cell data values, `sender_id` identifies the source rank, and `index` is the emission or reception array index used for scatter at the receiver. This self-describing format enables the receiver to place incoming data without maintaining separate communication tables.

For multi-variable exchanges (e.g. all hydro conserved variables), we provide bulk variants that pack all N_{var} columns of a 2D array into a single k -section exchange call:

$$\text{sendbuf}((v - 1)2^{N_{\text{dim}}} + j, i) = \text{xx}(v, \text{cell}_{i,j}), \quad (7)$$

for $v = 1, \dots, N_{\text{var}}$ and $j = 1, \dots, 2^{N_{\text{dim}}}$, plus two metadata entries. This amortises the tree-walk overhead and MPI latency over N_{var} variables, yielding a significant reduction in

Hierarchical exchange communication pattern for $N_{\text{cpu}} = 12 (= 3 \times 2 \times 2)$

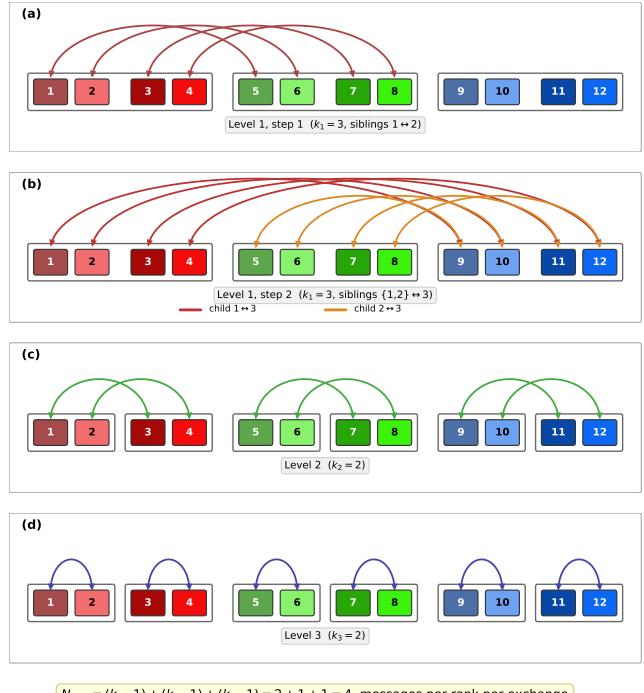


Figure 2. Hierarchical exchange communication pattern for $N_{\text{cpu}} = 12 (= 3 \times 2 \times 2)$. Coloured rectangles represent MPI ranks numbered 1–12, using the same colour scheme as Figure 1. Arcs denote bidirectional point-to-point exchanges. At level 1 ($k_1 = 3$), two steps connect each rank with correspondents in the two sibling subtrees; the two colours in step 2 distinguish the children $1 \leftrightarrow 3$ (dark red) and children $2 \leftrightarrow 3$ (orange) pairings that proceed concurrently. At levels 2 and 3 ($k_2 = k_3 = 2$), the communication range contracts to within each group and then to adjacent pairs. The total message count per rank is $N_{\text{msg}} = (3-1) + (2-1) + (2-1) = 4$.

the number of exchange calls per time step (from N_{var} individual calls to a single bulk call at each of the five call sites in `amr_step`).

3.3 Communication Structure Construction

The construction of the communication structure, which determines which grids must be exchanged as ghost zones, was itself based on MPI_ALLTOALL in the original RAMSES. We replace this with a k -section exchange: each rank packs its reception grids as triplets (sender identifier, reception index, grid address), sends them via the exclusive hierarchical exchange, and the receiver reconstructs its emission arrays from the incoming data. This eliminates the last remaining all-to-all communication pattern in the AMR infrastructure.

The k -section exchange routines and virtual boundary functions contain numerous small arrays (child counts, peer lists, MPI request handles, receive buffers) that are allocated and deallocated on every call. At 100+ calls per time step, the cumulative allocation overhead becomes non-negligible. We convert these to `save` variables with grow-only seman-

Table 2. Communication complexity per ghost-zone exchange operation. N_{ghost} is the total number of ghost grids per rank; k_l are the branching factors at tree level l .

	Original RAMSES	cURAMSES
Message count	$\mathcal{O}(N_{\text{cpu}})$	$\mathcal{O}(\sum_l k_l)$
Buffer memory	$\mathcal{O}(N_{\text{cpu}} \cdot N_{\text{ghost}})$	$\mathcal{O}(k_{\max} \cdot N_{\text{ghost}})$
MPI_ALLTOALL calls	≥ 1 per exchange	0

tics: the buffer is allocated on first use and grown (but never shrunk) when a larger size is needed. The receive buffer’s first dimension must match the `nprops` parameter exactly (for correct MPI stride), so reallocation is triggered when either the capacity or the property count changes. This optimization eliminates approximately 100 allocation/deallocation pairs per exchange call.

3.4 Complexity Analysis

Table 2 summarizes the communication complexity of the original RAMSES and cURAMSES.

The integer coordinates of a grid at level l are computed from its floating-point centre position \mathbf{x}_g as

$$i_d = \lfloor x_{g,d} \cdot 2^{l-1} \rfloor, \quad d \in \{x, y, z\}, \quad (9)$$

where $\lfloor \cdot \rfloor$ denotes the floor function and coordinates are in units of the coarse grid spacing. Note that AMR does not populate all possible grid positions at a given level: only regions that satisfy the refinement criteria contain grids. The Morton key therefore serves as a unique spatial address for each *existing* grid; the hash table (Section 4.3) stores only the grids that are actually allocated, making the look-up cost independent of the total number of potential grid positions at that level.

The neighbour of a grid in direction j (using the RAMSES convention $1 = -x, 2 = +x, 3 = -y, 4 = +y, 5 = -z, 6 = +z$) is obtained by: (i) decoding the Morton key via the inverse map $(i_x, i_y, i_z) = M^{-1}(\text{key})$; (ii) incrementing or decrementing the appropriate coordinate; (iii) applying periodic wrapping if the coordinate exceeds $[0, n_d \cdot 2^{l-1}]$; (iv) re-encoding to obtain the neighbour’s Morton key. The parent key is obtained by a 3-bit right shift: $M_{\text{parent}} = M \gg 3$. A child key is obtained by a 3-bit left shift plus the child index (0–7): $M_{\text{child}} = (M \ll 3) | i_{\text{child}}$.

4 MORTON KEY HASH TABLE AND MEMORY OPTIMIZATIONS

4.1 The Nbor Array Problem

RAMSES stores the octree connectivity in several arrays, the largest of which is `nbor(1:ngridmax, 1:twondim)` — a six-column integer array that records, for each grid, the cell index of its neighbour in each of the six Cartesian directions ($\pm x, \pm y, \pm z$). Each entry is a 64-bit integer occupying 8 bytes, so this array consumes $48 N_{\text{gridmax}}$ bytes — 240 MB for $N_{\text{gridmax}} = 5$ M. Moreover, the `nbor` array must be maintained during grid creation, deletion, defragmentation, and inter-rank migration — a significant source of code complexity and a potential source of bugs.

4.2 Morton Key Encoding and Neighbour Arithmetic

A Morton key (also known as a Z-order key) is a 64-bit integer formed by interleaving the bits of the three-dimensional integer coordinates (i_x, i_y, i_z) of a grid at its AMR level:

$$M(i_x, i_y, i_z) = \sum_{b=0}^{B-1} \left[\text{bit}_b(i_x) \cdot 2^{3b} + \text{bit}_b(i_y) \cdot 2^{3b+1} + \text{bit}_b(i_z) \cdot 2^{3b+2} \right] \quad (8)$$

where $B = 21$ bits per coordinate and $\text{bit}_b(n)$ extracts bit b of integer n . Here n_x denotes the number of coarse-level grid cells per dimension (a RAMSES parameter, typically 1–4). At AMR level l , there are $n_x \cdot 2^{l-1}$ grid cells per dimension, so the integer coordinate range is $[0, n_x \cdot 2^{l-1}]$. With $B = 21$ bits the maximum representable coordinate is $2^{21} - 1 = 2097151$, which accommodates up to level 22 for $n_x = 1$ or level 20 for $n_x = 4$. The encoding and decoding (the inverse map $(i_x, i_y, i_z) = M^{-1}(\text{key})$) are implemented with simple bit-shift loops.

4.3 Per-Level Hash Table and Replacement Functions

We maintain one open-addressing hash table (probing consecutive slots on collision) per AMR level, mapping Morton keys M to grid indices `igrid`.

The hash function uses multiplicative hashing with an additional mixing step:

$$h(M) = [((M \times \phi_1) \oplus (M \gg 16)) \times \phi_2] \oplus (h \gg 13), \quad (10)$$

where $\phi_1 = 2654435761$ and $\phi_2 = 0x9E3779B97F4A7C15$ are constants chosen for good bit mixing, and the table capacity is always a power of two to allow bitmask modular arithmetic. Collisions are resolved by linear probing; the load factor is kept below 0.7 by automatic rebasing (doubling capacity).

The hash table is maintained incrementally:

- `morton_hash_insert`: called during `make_grid_coarse` and `make_grid_fine`;
- `morton_hash_delete`: called during `kill_grid`;
- Full rebuild after defragmentation (`morton_hash_rebuild`).

A companion array `grid_level(igrid)` stores the AMR level of each grid, enabling Morton key computation from the grid index alone.

Two wrapper functions provide drop-in replacements for the original `nbor`-based access patterns:

• `morton_nbor_grid(igrid, ilevel, j)`: returns the grid index of the same-level neighbour in direction j , replacing the pattern `son(nbor(igrid, j))`. Implemented as: compute Morton key, shift by direction, look up in hash table.

• `morton_nbor_cell(igrid, ilevel, j)`: returns the father cell index of the neighbour, replacing the pattern `nbor(igrid, j)`. For level 1, returns the coarse cell index directly; for finer levels, computes the parent grid via the hash table at level $l - 1$ and the octant index from the coordinate parity.

Table 3. Memory savings per MPI rank for $N_{\text{gridmax}} = 5 \text{ M}$. Savings marked with * are conditional on using k -section ordering.

Optimization	MB	Availability
nbor removal (Morton hash)	240	Always
hilbert_key removal*	640	Steady state
On-demand bisec.ind.cell*	160	Between LB
On-demand cell.level*	160	Between LB
Defrag scratch (local)	40	Between defrag
Total	>1200	

assume QUADHILBERT; for standard 64-bit keys the savings would be 320 MB. Figure 4 visualizes the individual contributions.

We implement a diagnostic routine `writemem_minmax` that reports the minimum and maximum resident set size across all ranks at each coarse step, providing runtime verification of the memory savings.

5 MULTIGRID POISSON SOLVER OPTIMIZATIONS

The multigrid (MG) Poisson solver consumes a large fraction of the total runtime in self-gravitating cosmological simulations. In baseline RAMSES, profiling reveals that the MG solver accounts for approximately 55 per cent of the total wall-clock time per coarse step. We describe several optimizations that reduce this fraction to approximately 39 per cent.

5.1 Neighbour Grid Precomputation

The Gauss–Seidel (GS) smoother and residual computation both require access to the six Cartesian neighbours of each grid. In the Morton hash table approach (Section 4), each neighbour lookup involves a hash table query. While individual lookups are $\mathcal{O}(1)$, the GS kernel accesses 6 neighbours per grid, 8 cells per grid, and typically 4–5 V-cycle iterations, resulting in hundreds of hash lookups per grid per solve.

We precompute all neighbour grids into a contiguous array before entering the V-cycle iteration loop:

$$\text{nbor_grid_fine}(j, i) = \text{morton_nbor_grid}(\text{igrid_amr}(i), l, j), \quad (12)$$

for $j = 0, 1, \dots, 6$ (where $j = 0$ stores the grid’s own AMR index `igrid_amr`) and $i = 1, \dots, N_{\text{grid}}$. This array is allocated before the iteration loop and deallocated after, so its memory overhead is transient.

The original GS kernel also contains a branch on `igshift == 0` to distinguish between the current grid and its neighbours. We unify the access pattern with a cache array `nbor_grids_cache(0:twondim)`, where index 0 references the grid itself. All neighbour accesses — including the self-reference — use the same indexed load, eliminating the branch.

5.2 Merged Red-Black Exchange and Fused Kernels

Standard red-black Gauss–Seidel smoothing in RAMSES performs a ghost-zone exchange of the potential ϕ between the red and black sweeps: Red \rightarrow Exchange(ϕ) \rightarrow Black \rightarrow Exchange(ϕ). Each iteration thus requires two exchanges for the smoother alone, plus additional exchanges for the residual and restriction/prolongation steps — a total of 9 exchange calls per iteration.

We merge the red and black sweeps by removing the inter-sweep exchange: Red \rightarrow Black \rightarrow Exchange(ϕ). This is a form of relaxed synchronisation: boundary cells in the black sweep use values from the previous iteration rather than freshly exchanged red-sweep values. For the MG preconditioner, this does not affect convergence in practice — the MG solve is

468 The nbor array is reduced to `allocate(nbor(1:1, 1:1))`
469 — effectively eliminated while maintaining compilation com-
470 patibility with any remaining references.

4.4 Hilbert Key and Histogram Array Elimination

472 When using k -section ordering, the Hilbert key
473 array `hilbert_key(1:ncell)` is no longer needed
474 for domain decomposition. We replace it with
475 `allocate(hilbert_key(1:1))`, saving $16 \times N_{\text{gridmax}} \times 2^{N_{\text{dim}}}$
476 bytes under QUADHILBERT (128-bit keys stored as two 64-bit
477 integers). For $N_{\text{gridmax}} = 5 \text{ M}$, this is approximately 640 MB.

478 The defragmentation routine, which previously required
479 Hilbert keys for reordering, uses a local scratch array
480 (`defrag_dp`) allocated only during the defragmentation pass
481 and immediately deallocated.

482 Similarly, the arrays `bisec.ind.cell` and `cell.level`, each
483 of size $N_{\text{gridmax}} \times 2^{N_{\text{dim}}}$ integers (8 bytes), are used exclusively
484 during load balancing to build the bisection histogram. We
485 allocate them on entry to `init_bisection_histogram` and
486 deallocate them after `cmp_new.cpu.map` returns, saving $2 \times$
487 $8 \times N_{\text{gridmax}} \times 2^{N_{\text{dim}}} \approx 320 \text{ MB}$ for $N_{\text{gridmax}} = 5 \text{ M}$. Since
488 load balancing occurs only every `nremap` coarse steps, these
489 arrays are absent during the vast majority of the simulation.

4.5 Memory Savings Summary

490 The memory cost of the hash table is

$$M_{\text{hash}} \approx \frac{N_{\text{grids}}}{0.7} \times (8 + 4) \text{ bytes} \approx 17 N_{\text{grids}} \text{ bytes}, \quad (11)$$

492 where N_{grids} is the actual number of grids (typically much less
493 than N_{gridmax}), 8 bytes per key, 4 bytes per grid index, and a
494 load factor of 0.7 accounts for empty slots. The `grid.level`
495 array adds $4 \times N_{\text{gridmax}}$ bytes.

496 Compared to the original nbor cost of $48 \times N_{\text{gridmax}}$ bytes,
497 the net savings are approximately $44 N_{\text{gridmax}} - 17 N_{\text{grids}}$.
498 Since $N_{\text{grids}} \ll N_{\text{gridmax}}$ in practice (typical occupancy is
499 30–60 per cent), the savings are substantial: $\sim 176 \text{ MB}$ for
500 $N_{\text{gridmax}} = 5 \text{ M}$ at 50 per cent occupancy.

501 The computational cost of a hash lookup is $\mathcal{O}(1)$ expected
502 time, with worst-case linear probing bounded by the load
503 factor. In practice, the precomputed neighbour caches de-
504 scribed in Section 5 amortize any per-lookup overhead in the
505 performance-critical Poisson solver.

506 Table 3 summarizes the combined memory savings for
507 $N_{\text{gridmax}} = 5 \text{ M}$.

508 The reported memory savings of the nbor array account
509 for both the eliminated array ($48 N_{\text{gridmax}} = 240 \text{ MB}$) and the
510 hash table overhead ($\sim 17 N_{\text{grids}}$, typically $< 50 \text{ MB}$). Net sav-
511 ings are at least 190 MB. The Hilbert key savings of 640 MB

Morton key hash table replaces nbor array

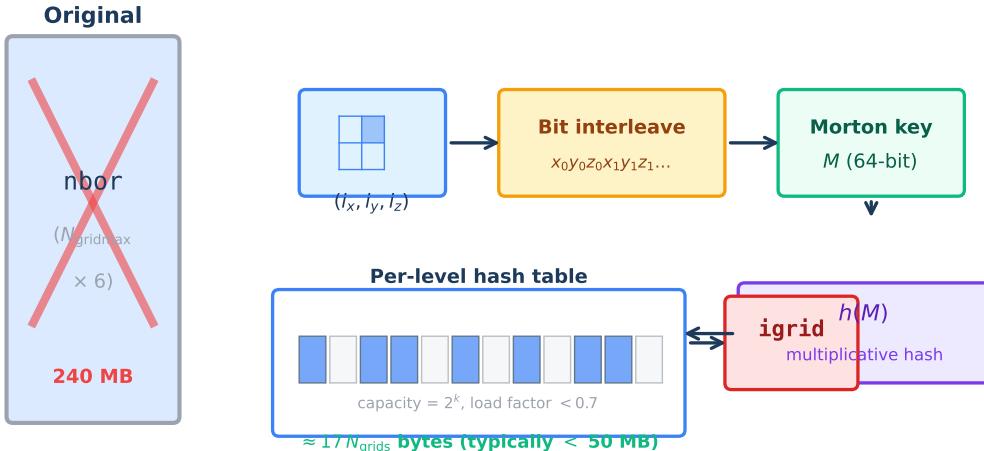


Figure 3. Schematic of the Morton key hash table that replaces the original `nbor` array. Left: the eliminated `nbor` ($N_{\text{gridmax}} \times 6$) array (240 MB for $N_{\text{gridmax}} = 5$ M). Right: the replacement workflow — grid cell coordinates (i_x, i_y, i_z) are bit-interleaved into a 64-bit Morton key M , which is mapped to a grid index i_{grid} via a per-level open-addressing hash table with multiplicative hashing. The hash table memory footprint is $\approx 17 N_{\text{grids}}$ bytes (typically <50 MB), yielding a net saving of >190 MB per rank.

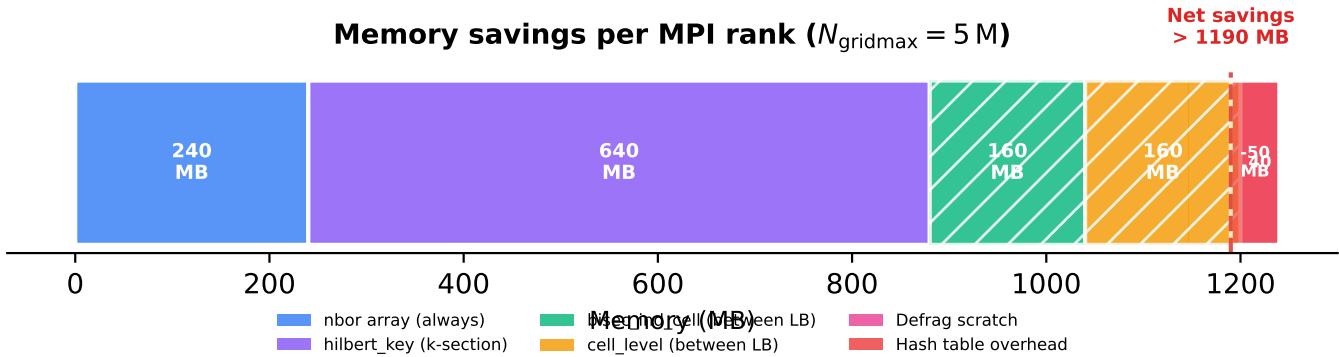


Figure 4. Per-rank memory savings breakdown for $N_{\text{gridmax}} = 5$ M. Solid bars denote unconditional savings (always available); hatched bars denote savings available only between load-balance steps or defragmentation passes. The red segment shows the hash table overhead (~40–50 MB), yielding a net saving of >1190 MB per MPI rank.

itself an approximate preconditioner for the conjugate gradient outer iteration, and the stale-ghost error is well within the MG tolerance.

We also remove two unnecessary residual exchanges per iteration, reducing the total from 9 to 5 exchange calls per iteration — a 44 per cent reduction in MG communication volume.

The same optimization is applied to the coarse-level solver (direct solve, pre-smoothing, post-smoothing), where the merged red-black pattern similarly halves the exchange count.

The MG algorithm requires both the residual $r = f - A\phi$ and its L^2 norm $\|r\|_2^2$ at specific points in the V-cycle. In the original code, these are computed in separate passes. We add an optional `norm2` argument to `cmp_residual_mg_fine`: when present, the norm is accumulated during the same loop that computes the residual, saving one full grid traversal. Since the

subroutine is `external` (not module-contained), callers must include an `interface` block to enable the optional-argument dispatch.

The GS fast-path computation involves a division by $2N_{\text{dim}} = 6$:

$$\phi_{\text{new}} = \frac{\sum_j \phi_j - h^2 f}{2N_{\text{dim}}}. \quad (13)$$

We replace the division / `dtwondim` with a multiplication by the precomputed reciprocal * `oneoverdtwondim`, which is faster on most architectures.

5.3 Performance Impact

Combining all optimizations, the MG Poisson solver's share of total runtime is reduced from 55.1 per cent to 38.6 per cent in a representative test (200 M particles, 12 ranks, 10 coarse

Multigrid V-cycle exchange optimization

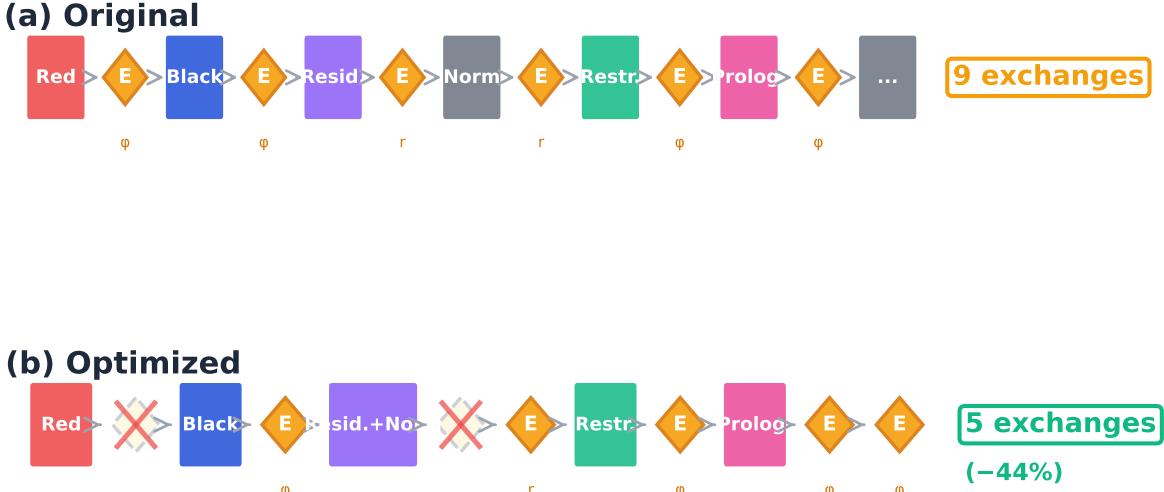


Figure 5. Multigrid V-cycle exchange optimization. (a) Original: each smoothing iteration requires 9 ghost-zone exchanges — two for the red-black sweeps (exchanging ϕ), one each for the residual and norm (exchanging r), and additional exchanges for restriction and prolongation. (b) Optimized: the inter-sweep red-black exchange is removed (relaxed synchronisation), and the separate residual and norm exchanges are fused, reducing the total to 5 exchanges per iteration (−44 per cent). Orange diamonds marked “E” denote active exchanges; crossed-out diamonds indicate eliminated exchanges.

591 steps). The iteration counts are unchanged (Level 8: 5 iterations, Level 9: 4 iterations), confirming that the merged red-black exchange does not degrade convergence.

594 6 FEEDBACK SPATIAL BINNING

595 6.1 The Brute-Force Bottleneck

596 The Type II supernova (SNII) feedback implementation in
597 RAMSES involves two computationally expensive routines:

598 • **average_SN**: averages hydrodynamic quantities within
599 the blast radius of each SN event, accumulating volume, mo-
600 mentum, kinetic energy, mass loading, and metal loading.
601 The original implementation loops over all cells \times all SNe,
602 yielding $\mathcal{O}(N_{\text{cells}} \times N_{\text{SN}})$ complexity.

603 • **Sedov_blast**: injects the blast energy and ejecta into
604 cells within the blast radius. Same $\mathcal{O}(N_{\text{cells}} \times N_{\text{SN}})$ complex-
605 ity.

606 In production simulations with ~ 2000 simultaneous SN
607 events, these routines consume 66 s and 11 s per call respec-
608 tively, dominating the feedback time step.

609 6.2 Spatial Hash Binning

610 We partition the simulation domain into a uniform grid of
611 n_{bin}^3 bins, where $n_{\text{bin}} = \max(1, \min(128, \lfloor L_{\text{box}}/r_{\text{max}} \rfloor))$ and
612 r_{max} is the maximum SN blast radius (the larger of **rcell**
613 \times **dx_min** and **rbubble**). Each SN event is assigned to a bin
614 based on its position, and a linked list threads the events
615 within each bin: $\text{bin_head}(i_x, i_y, i_z) \rightarrow \text{SN}_1 \rightarrow \text{SN}_2 \rightarrow \dots$

616 For each cell, we compute its bin index and check only the
617 27 neighbouring bins (the cell’s own bin plus its 26 face-, edge-
618 , and corner-adjacent bins). Since r_{max} is at most the bin size
619 by construction, this 27-bin neighbourhood is guaranteed to
620 contain all SNe that could influence the cell. The complexity
621 becomes $\mathcal{O}(N_{\text{cells}} \times \bar{n}_{\text{SN/bin}} \times 27)$, where $\bar{n}_{\text{SN/bin}} = N_{\text{SN}}/n_{\text{bin}}^3$
622 is the average number of SNe per bin. Figure 6 contrasts the
623 two approaches.

624 6.3 Performance Results

625 The binned **average_SN** uses cell-parallel OpenMP thread-
626 ing: the outer loop is over grids (with `!$omp parallel do`),
627 and each thread processes the cells of one grid. When a cell
628 falls within an SN blast radius, the thread accumulates its
629 contribution using `!$omp atomic` directives on the shared
630 SN-indexed arrays (**vol_gas**, **dq**, **ekBlast**, etc.). The atomic
631 overhead is minimal because collisions are rare — most bins
632 contain zero or one SN, so contention is low. The **Sedov_blast**
633 routine writes only to cells owned by each grid, so no atomics
634 are needed. The outer loop is over grids, and each thread in-
635 dependently processes the cells of its assigned grids, checking
636 only the 27 neighbouring bins for relevant SNe.

637 With approximately 2000 simultaneous SN events:

- **average_SN**: 66 s \rightarrow 0.25 s ($\sim 260 \times$ speedup)
- **Sedov_blast**: 11 s \rightarrow 0.07 s ($\sim 157 \times$ speedup)

638 Verification by restarting at the same snapshot confirms
639 bit-identical results for all conservation quantities (m_{cons} ,
640 e_{cons} , e_{pot} , e_{kin} , e_{int}).

641 The same spatial binning technique is applied to the AGN

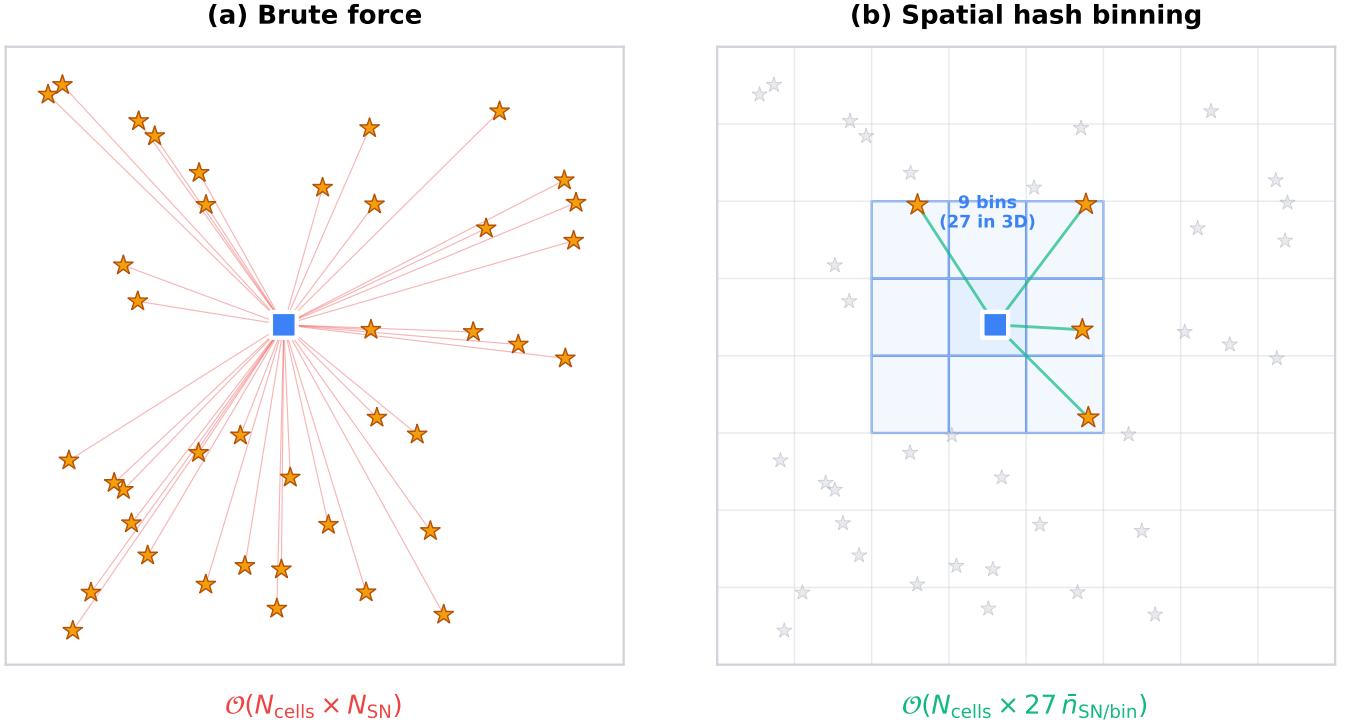


Figure 6. Feedback spatial binning concept (shown in 2D for clarity). (a) Brute force: every cell (blue square) must check all SN events (orange stars), resulting in $\mathcal{O}(N_{\text{cells}} \times N_{\text{SN}})$ complexity. (b) Spatial hash binning: the domain is partitioned into a uniform grid of bins. Each cell checks only the 27 neighbouring bins (9 in 2D), shown as shaded regions, reducing the complexity to $\mathcal{O}(N_{\text{cells}} \times 27 \bar{n}_{\text{SN}/\text{bin}})$. Distant SN events (grey) are never examined.

feedback routines (`average_AGN` and `AGN_blast`), which suffer from the same $\mathcal{O}(N_{\text{cells}} \times N_{\text{AGN}})$ brute-force scaling. In production simulations with tens of thousands of active AGN sink particles, these routines dominate the sink-particle time step.

The AGN feedback involves three distinct interaction modes (saved energy injection, jet feedback, and thermal feedback), each with a different geometric distance criterion. The spatial binning is agnostic to these distinctions: it reduces the candidate AGN set from the full population to only those in the 27 neighbouring bins, while preserving all distance-check logic and physical calculations unchanged. The linked-list construction and 27-bin traversal follow the same pattern as the SNII implementation (§6.2), with `bin_head` and `agn_next` arrays replacing the SN-specific versions.

With approximately 32 000 active AGN particles, the binned `average_AGN` achieves a 30× speedup and `AGN_blast` a 14× speedup, reducing the total AGN feedback time by a factor of ∼4. Verification confirms bit-identical conservation diagnostics compared to the original brute-force implementation.

7 VARIABLE-NCPU RESTART

7.1 HDF5 Parallel I/O and Restart

Standard RAMSES writes one binary file per MPI rank per output. Restarting with a different number of ranks is not

directly supported, requiring an intermediate step of reading with the original rank count, redistributing, and re-writing.

We implement HDF5 parallel I/O using the HDF5 library’s MPI-IO backend. All ranks write to (and read from) a single HDF5 file, with datasets organized hierarchically:

- `/amr/level_{1}/`: grid positions, son flags, CPU map for each AMR level.
- `/hydro/level_{1}/`: conserved variables ρ , $\rho\mathbf{v}$, E , etc.
- `/gravity/level_{1}/`: gravitational potential ϕ and force components.
- `/particles/`: positions, velocities, masses, IDs, levels, formation times, metallicities.
- `/sinks/`: sink particle properties.

When the number of ranks in the output file ($N_{\text{cpu}}^{\text{file}}$) differs from the current run (N_{cpu}), the following procedure executes during restart:

- (i) Build a uniform k -section tree for the new N_{cpu} (equal-volume partitioning, without load-balance adjustment).
- (ii) Read all grids from the HDF5 file. Since the file is a single shared file, all ranks can access all data.
- (iii) For each grid, compute the CPU ownership from the father cell’s position using `cmp_ksection_cpumap`.
- (iv) Each rank retains only the grids assigned to it, building the local AMR tree incrementally.
- (v) Hydro, gravity, and particle data are read and scattered to locally owned grids using a precomputed file-index-to-local-grid mapping (`varcpu_grid_file_idx`).
- (vi) On the first coarse step after restart, a forced load-

698 balance operation redistributes grids for optimal balance un- 753
 699 der the new rank configuration. 754

700 This approach requires that all ranks temporarily hold the 755
 701 full grid metadata (positions and son flags) during the re- 756
 702 construction phase. For typical production outputs ($\sim 10 \text{ M}$ 757
 703 total grids), this temporary overhead is a few hundred MB 758
 704 — well within the memory budget freed by the optimizations 759
 705 of Section 4. 760

706 7.2 Binary Distributed I/O Restart

707 When HDF5 is unavailable or the native binary format 765
 708 (`informat='origin'`) is preferred, a distributed I/O strat- 766
 709 egy enables variable- N_{cpu} restart from the per-rank bi- 767
 710 nary files written by standard RAMSES. The binary for- 768
 711 mat stores one file per MPI rank — `amr_XXXXX.outYYYYY`, 769
 712 `hydro_XXXXX.outYYYYY`, etc. — so the number of files equals 770
 713 $N_{\text{cpu}}^{\text{file}}$, which may differ from the current N_{cpu} . 771

714 The restart proceeds in three stages.

715 7.2.0.1 Stage 1: Early detection and file assignment.

716 Rank 1 probes the header of file 00001 to read $N_{\text{cpu}}^{\text{file}}$. If 755
 717 $N_{\text{cpu}}^{\text{file}} \neq N_{\text{cpu}}$, the variable- N_{cpu} path is activated (requir- 756
 718 ing k -section ordering). The $N_{\text{cpu}}^{\text{file}}$ files are distributed among 757
 719 the N_{cpu} ranks by round-robin assignment: rank r reads files 758
 720 whose index satisfies $(f - 1) \bmod N_{\text{cpu}} = r - 1$. 759

721 7.2.0.2 Stage 2: Distributed AMR reconstruction.

722 Each rank reads only its assigned AMR files, extract- 782
 723 ing per-level active grid metadata (positions \mathbf{x}_g and son 781
 724 flags). The per-level active grid counts are aggregated via 783
 725 `MPI_ALLREDUCE`. For each level, the grids read by each 784
 726 rank are assigned to their correct owner by evaluating 785
 727 `cmp_ksection_cpumap` on the father cell position (computed 786
 728 from \mathbf{x}_g and the parent-child octant relationship). An 787
 729 `MPI_ALLTOALLV` exchange then routes each grid's data to its 788
 730 owner, who creates the local AMR grid. The exchange meta- 789
 731 data — send ordering and receive-grid mapping — is stored 790
 732 for reuse.

733 7.2.0.3 Stage 3: Hydro and gravity scatter.

734 The hydro and gravity binary files are read in the same distributed 791
 735 fashion: each rank reads only its assigned files and packs the 792
 736 cell-centred data into per-level send buffers using the stored 793
 737 send ordering. The same `MPI_ALLTOALLV` counts and displace- 794
 738 ments from Stage 2 are reused, and the receive-grid mapping 795
 739 scatters incoming data to the correct local cells. Since the bi- 796
 740 nary format stores primitive variables (density, velocity, pres- 797
 741 sure), a primitive-to-conservative conversion is applied on the 798
 742 receiving side.

743 Particle files are handled independently: each rank reads its 799
 744 assigned files and retains only those particles whose positions 800
 745 fall within the local k -section domain.

746 This three-stage approach ensures that no rank reads more 802
 747 than $\lceil N_{\text{cpu}}^{\text{file}} / N_{\text{cpu}} \rceil$ files (at most two for practical configura- 803
 748 tions), and the `MPI_ALLTOALLV` exchange per level has cost 804
 749 proportional to the number of grids exchanged rather than 805
 750 the total number of ranks. Verification tests confirm $e_{\text{cons}} = 0$
 751 for both upward ($4 \rightarrow 12$) and downward ($12 \rightarrow 4$) rank- 806
 752 count changes. Figure 7 summarizes the three-stage pipeline.

7.3 Stream-Access IC Reading and Sink Refinement Fix

The initial condition (IC) files in GRAFIC2 format are Fortran sequential-access binary files. In the original RAMSES, each rank reads the entire file sequentially, skipping planes until reaching its assigned region. For large ICs, this sequential skipping becomes a significant I/O bottleneck.

We replace sequential access with Fortran 2003 stream access (`ACCESS='STREAM'`), which allows direct byte-offset positioning. The byte offset for plane i in a file with header size $H = 52$ bytes (GRAFIC2 44-byte header plus record markers) and plane size $P = n_1 n_2 \times 4 + 8$ bytes (data plus two 4-byte record markers) is offset = $H + (i-1) \times P + 5$. This is applied to all IC file types: density perturbation (`deltab`), velocity components (`velcx/y/z`), particle positions (`poscx/y/z`), and temperature.

We also identified and fixed a bug in the sink particle refinement criterion. The original implementation in `cic.amr` added the refinement mass threshold `m_refine` to the gravitational potential array `phi`. However, the Poisson solver subsequently overwrites `phi`, erasing the refinement flag.

The fix moves the sink-particle refinement check to `sub_userflag_fine` in `flag_utils`, where it is evaluated after the Poisson solve. For each grid, the particle linked list is traversed once to build a bitmask indicating which child cells contain sink particles (identified by `idp < 0` and `tp = 0`). The cell assignment is determined by comparing the particle position to the grid centre to identify the octant. After calling `poisson_refine`, cells flagged in the bitmask are forced to refine regardless of the Poisson criterion.

8 GPU-ACCELERATED SOLVERS

Certain compute-intensive routines—the Godunov solver, gravity force computation, hydrodynamic synchronisation, CFL timestep, prolongation, and radiative cooling—are amenable to GPU acceleration. Rather than offloading entire time steps to the GPU, cuRAMSES adopts a *hybrid dispatch* model in which OMP threads dynamically choose between CPU and GPU execution at runtime.

8.1 Dynamic Dispatch Model

At the start of each parallel region, each OMP thread attempts to acquire a GPU stream slot via an atomic counter. Threads that succeed accumulate grid data into a **batched grid buffer** of configurable size (typically 4096 grids) and launch GPU kernels asynchronously when the buffer is full. Threads that do not acquire a slot execute the standard CPU code path. The `schedule(dynamic)` clause ensures load balancing: if a GPU thread is waiting for kernel completion, remaining loop iterations are picked up by CPU threads.

This design requires no code duplication—the CPU path is the original Fortran subroutine, and the GPU path is an alternative branch within the same `!$omp do` loop.

8.2 Batched Grid Buffering

GPU kernel launch latency ($\sim 10\text{--}50 \mu\text{s}$) is amortised by batching: each GPU thread accumulates the full stencil data

Binary variable- N_{cpu} restart: 3-stage distributed I/O

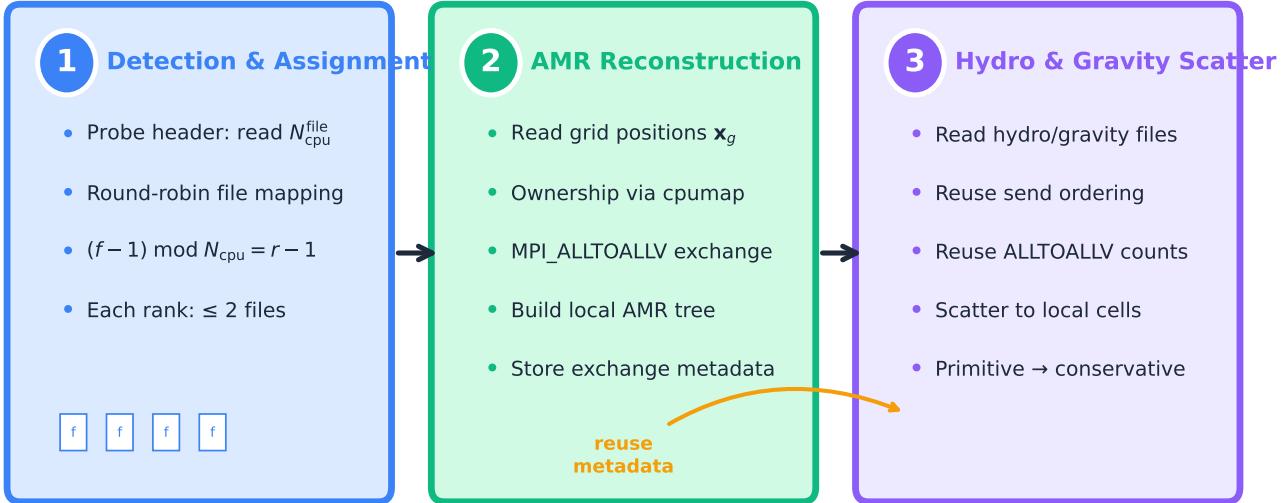


Figure 7. Three-stage distributed I/O pipeline for binary variable- N_{cpu} restart. Stage 1 (blue): the header is probed to detect a rank-count mismatch, and the $N_{\text{cpu}}^{\text{file}}$ files are assigned to N_{cpu} ranks by round-robin mapping. Stage 2 (green): AMR grid positions are read, ownership is determined via `cmp_ksection_cpumap`, and grids are exchanged via `MPI_ALLTOALLV`; the exchange metadata is stored for reuse. Stage 3 (purple): hydro and gravity files are read and scattered using the stored metadata, including a primitive-to-conservative conversion on the receiving side.

for many grids before launching a single kernel covering all accumulated grids. For the Godunov solver, the GPU pipeline executes five kernels in sequence: primitive variable conversion, slope computation, Riemann tracing, flux computation, and artificial diffusion.

A key optimisation is the **on-device flux reduction** that computes the conservative update entirely on the GPU. Instead of transferring the full flux array back to the host (~ 98 MB per flush), the kernel reduces fluxes into compact per-grid output arrays, reducing the device-to-host transfer to ~ 5 MB per flush—a $20\times$ reduction in PCIe bandwidth.

The Godunov solver updates conservative variables at both the current level L and the coarser level $L-1$. Level L writes are conflict-free by construction (each grid maps to unique cell indices), but level $L-1$ writes can conflict when multiple fine grids share the same coarse parent cell. The original code serialised both levels with `!$omp critical`, destroying all OMP parallelism in the scatter phase.

We eliminate this lock entirely: level L results are written directly to `unew`, while each thread appends level $L-1$ flux contributions to a private scatter buffer. After the parallel region, a serial merge applies all buffered entries. The merge cost is negligible (< 0.01 s in all tests), and the result is exact—no approximation or race condition.

8.3 Fortran–CUDA Interface

The Fortran–CUDA interface uses a two-layer design: a C binding layer (`bind(C)` with `type(c_ptr)` arguments) and a Fortran wrapper layer that converts assumed-size arrays to C

pointers via `c_loc`. The assumed-size pattern avoids Fortran array descriptors, which can produce incorrect addresses with certain compilers (notably Intel ifx).

8.4 Performance

The GPU-accelerated code produces bit-identical physics results compared to the CPU-only build. On an RTX 5000 Ada GPU with 4 MPI ranks \times 2 OMP threads, the Godunov solver is accelerated by 16 per cent (22.0 s \rightarrow 18.4 s). The overall speedup is modest because the host-to-device transfer currently dominates (50 per cent of GPU time), leaving significant room for future optimisation via persistent device-side data structures.

9 PERFORMANCE RESULTS

9.1 Test Configuration

All tests use a cosmological Λ CDM simulation with 200 M dark matter particles in a periodic box of side $256 h^{-1}$ Mpc, initialised at $z = 29.5$ with MUSIC (Hahn & Abel 2011). The base AMR grid is 256^3 (`levelmin=8`) with adaptive refinement up to `levelmax=10`. The simulation is restarted from an HDF5 output at coarse step 5 and evolved to step 10 (5 coarse steps). The test platform is a dual-socket AMD EPYC 7543 node (64 physical cores, 128 threads) with 1 TB of DDR4 memory.

Table 4. Conservation diagnostics at step 10 for various configurations. All values are identical to the reference within machine precision.

Configuration	e_{cons}	e_{pot}	e_{kin}
Hilbert (reference)	3.77×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
K-sec (no membal)	3.77×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
K-sec (membal)	3.77×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
Morton + k-sec	3.77×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
MG optimized	3.79×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}

9.2 Conservation Verification

All modifications are verified to preserve physical consistency by comparing conservation diagnostics between the modified code and a reference run:

The slight change in e_{cons} for the MG-optimized version (3.79×10^{-3} versus 3.77×10^{-3}) is attributable to the relaxed synchronisation in the merged red-black GS sweep, where boundary cells use ghost values from the previous iteration. This is well within the MG solver’s convergence tolerance and does not affect the iteration count.

9.3 Strong Scaling

We measure strong scaling by restarting a production-grade cosmological simulation from an HDF5 output. The test problem comprises 54 M AMR grids (levels 9–14) and 135.7 M particles including 3.2×10^4 sink particles, with full physics enabled (radiative cooling, star formation, SNII/AGN feedback). The output was written at coarse step 241 with 12 MPI ranks; we run 2 additional coarse steps to step 243. The variable- N_{cpu} restart feature (§7) allows the output to be read with any number of MPI ranks; a forced `load_balance` on the first coarse step ensures optimal grid distribution before timing begins. The test platform has two AMD EPYC 7543 processors (64 cores, 128 threads) and 1 TB of shared memory; all runs use `OMP_NUM_THREADS=1`.

Table 5 summarises the wall-clock time and per-component timer averages for $N_{\text{cpu}} = 1$ –64. The elapsed time is the total wall-clock time including HDF5 I/O and load balancing. All configurations yield energy conservation errors e_{cons} of $\mathcal{O}(10^{-4})$, with small variations due to floating-point summation order changes across different domain decompositions.

Several scaling trends are evident:

- *Particle operations* scale nearly ideally from 1 to 64 ranks: 538 s → 14 s, a 38.4× speedup for 64× the ranks.
- *Multigrid Poisson solver* dominates the total time ($\sim 37\%$), scaling from 4034 s (1 rank) to 106 s (64 ranks). The 38.2× speedup is near-ideal, reflecting the effectiveness of the Morton hash-based neighbour lookup and precomputed cache arrays.
- *Godunov solver* scales 58.4× from 1 to 64 ranks (2536 s → 43 s). The super-linear speedup arises because multi-rank runs benefit from cache effects: each rank processes a smaller working set that fits in the L3 cache.
- *Sink particle operations* scale well up to 24 ranks (1346 s → 69 s, 19.5×) but slow beyond 32 ranks due to the global ALLREDUCE operations in AGN feedback.
- *Load balancing* overhead converges to ~ 16 s beyond 32

ranks. This cost is amortised over `nremap` coarse steps (every 5 steps in production).

- The *overall speedup* reaches $33.9\times$ at 64 ranks relative to the single-rank baseline, demonstrating excellent scaling efficiency (53%) on this shared-memory node.

Figure 8 visualises these trends. Panel (a) shows the elapsed time and per-component timer values as a function of N_{cpu} on a log–log scale. All major components follow the ideal scaling line closely up to ~ 16 ranks, with gradual deviation at higher rank counts due to communication overhead. Panel (b) shows the speedup relative to the single-rank baseline: particle operations and the flag routine achieve near-ideal scaling, while the elapsed time reaches $33.9\times$ at 64 ranks.

9.4 OpenMP Thread Scaling

To evaluate intra-rank parallelism we fix $N_{\text{cpu}} = 4$ and vary `OMP_NUM_THREADS` from 1 to 30, using the same test problem as Section 9.3. The total core count ranges from 4 to 120; the physical core limit of the dual-socket node is 64.

Table 6 summarises the elapsed time and per-component timer averages. All runs conserve energy to $e_{\text{cons}} \leq 6.56 \times 10^{-4}$.

Several trends distinguish the OpenMP scaling from the MPI-only results of Table 5:

- *Multigrid Poisson solver* benefits the most from threading: 1085 → 121 s at 16 threads (8.9×), with continued gains beyond 16 threads reaching 10.5× at 30 threads. The precomputed neighbour arrays and fused residual loops (Section 5) expose substantial loop-level parallelism.

- *Godunov solver* scales 11.4× from 1 to 16 threads (610 → 54 s). Beyond 16 threads, performance degrades slightly due to memory bandwidth saturation and NUMA effects.

- *Sinks and Poisson* exhibit modest scaling (3.0× and 3.0× at 16 threads) because sink operations involve global reductions and serial sections that limit parallelism.

- *Overall speedup* plateaus at 5.8× with 16 threads (64 cores), limited by the serial fraction of MPI communication and sink-particle operations. Beyond the physical core count, oversubscription yields no further improvement.

Figure 9 compares per-component timers and speedup curves as a function of N_{omp} . The vertical dotted line marks the physical core limit (64 cores). Panel (a) shows that the MG solver and Godunov components track the ideal scaling line most closely, while sinks and Poisson saturate early. Panel (b) confirms that the overall speedup reaches a ceiling near 6×, indicating that further performance gains require additional MPI ranks rather than more threads per rank.

9.5 Memory-Weighted Load Balancing

The memory-weighted cost function (equation 4) assigns $w_{\text{grid}} = 270$ bytes per grid cell and $w_{\text{part}} = 12$ bytes per particle. Table 7 quantifies the load balance achieved by this scheme across the strong-scaling runs described in the preceding section.

The memory-weighted balancer keeps the per-rank memory imbalance remarkably low: $M_{\max}/M_{\min} \leq 1.05$ for all rank counts tested, from 2 to 64. This is achieved despite substantial grid-count imbalance at the most populated refined

Table 5. Strong scaling results for the CURAMSES code on a dual-socket AMD EPYC 7543 node (64 cores, 1 TB RAM). The test problem has 54 M grids and 135.7 M particles with full physics (cooling, star formation, sink particles, AGN feedback). Elapsed is the total wall-clock time; timer values are per-rank averages. Speedup S is relative to the single-rank elapsed time.

N_{cpu}	Elapsed (s)	S	MG (s)	Godunov (s)	Sinks (s)	Poisson (s)	Particles (s)	Feedback (s)	Flag (s)	Load Bal. (s)	Cooling (s)
1	8181.5	1.0	4034.1	2536.4	1345.5	1130.1	538.3	210.7	161.3	26.4	158.9
2	4253.1	1.9	2152.5	1218.0	656.1	584.7	272.0	107.3	83.5	83.7	79.7
4	2150.9	3.8	1084.6	612.1	312.4	295.0	134.9	53.3	41.5	50.9	41.0
8	1138.6	7.2	558.4	293.9	171.9	159.1	73.9	27.8	21.2	33.0	21.5
12	793.3	10.3	387.3	198.6	131.3	107.4	51.6	20.6	14.0	26.0	14.6
16	613.4	13.3	296.2	144.1	94.9	85.3	38.6	15.4	10.9	21.2	11.3
24	425.3	19.2	205.6	98.6	69.2	57.5	26.4	11.3	7.8	18.9	7.7
32	357.2	22.9	169.0	73.9	53.7	49.4	21.5	9.4	6.7	16.5	6.0
48	271.0	30.2	121.9	52.9	47.5	35.4	16.8	7.7	5.1	16.1	4.2
64	241.7	33.9	105.5	43.4	41.5	31.8	13.9	7.5	4.8	16.0	3.3

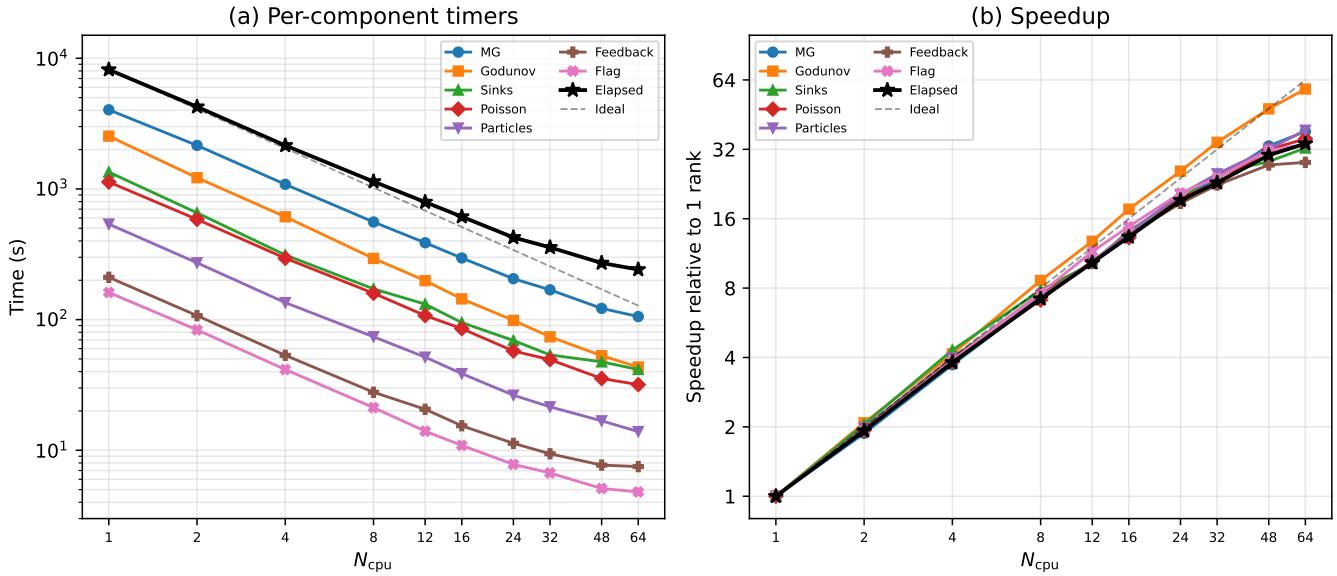


Figure 8. Strong scaling of CURAMSES on a dual-socket AMD EPYC 7543 node (64 cores) with a 135.7 M particle cosmological simulation including full physics. (a) Elapsed time and per-component wall-clock times versus N_{cpu} ; the dashed line shows ideal scaling from the single-rank baseline. (b) Speedup relative to 1 rank. Particle operations and the flag routine scale near-ideally, while the overall speedup reaches 33.9 \times at 64 ranks (53% parallel efficiency).

Table 6. OpenMP thread scaling with $N_{\text{cpu}} = 4$ MPI ranks. The test problem has 54 M grids and 135.7 M particles with full physics. N_{omp} is the number of OpenMP threads per rank; total cores is $N_{\text{cpu}} \times N_{\text{omp}}$. Speedup S is relative to the single-thread elapsed time. The vertical rule separates runs within the 64-core physical limit from those that oversubscribe.

N_{omp}	Cores	Elapsed (s)	S	MG (s)	Godunov (s)	Sinks (s)	Poisson (s)	Particles (s)	Feedback (s)	Flag (s)	Cooling (s)
1	4	2147.6	1.0	1085.4	610.3	312.4	297.9	135.3	53.3	41.3	41.1
2	8	1234.7	1.7	623.5	306.1	209.0	193.3	90.4	30.2	25.1	20.7
4	16	736.7	2.9	344.9	155.5	149.9	139.7	65.6	18.5	15.2	10.6
6	24	583.4	3.7	258.8	109.1	130.3	122.7	58.1	15.0	12.1	7.4
8	32	492.5	4.4	202.5	84.0	120.3	112.8	53.9	12.7	10.5	5.6
10	40	441.6	4.9	169.7	70.1	114.2	108.9	51.5	11.6	9.3	4.5
12	48	407.3	5.3	148.9	60.9	110.3	105.0	49.6	10.8	8.6	3.8
16	64	369.8	5.8	121.2	53.6	105.9	100.7	47.9	9.7	7.7	2.9
20	80	374.3	5.7	124.4	56.8	105.8	99.7	47.7	10.0	7.8	2.7
24	96	363.1	5.9	114.3	59.0	103.0	97.5	46.4	9.5	7.4	2.5
30	120	365.6	5.9	103.2	65.4	105.2	98.0	47.4	9.5	7.0	2.3

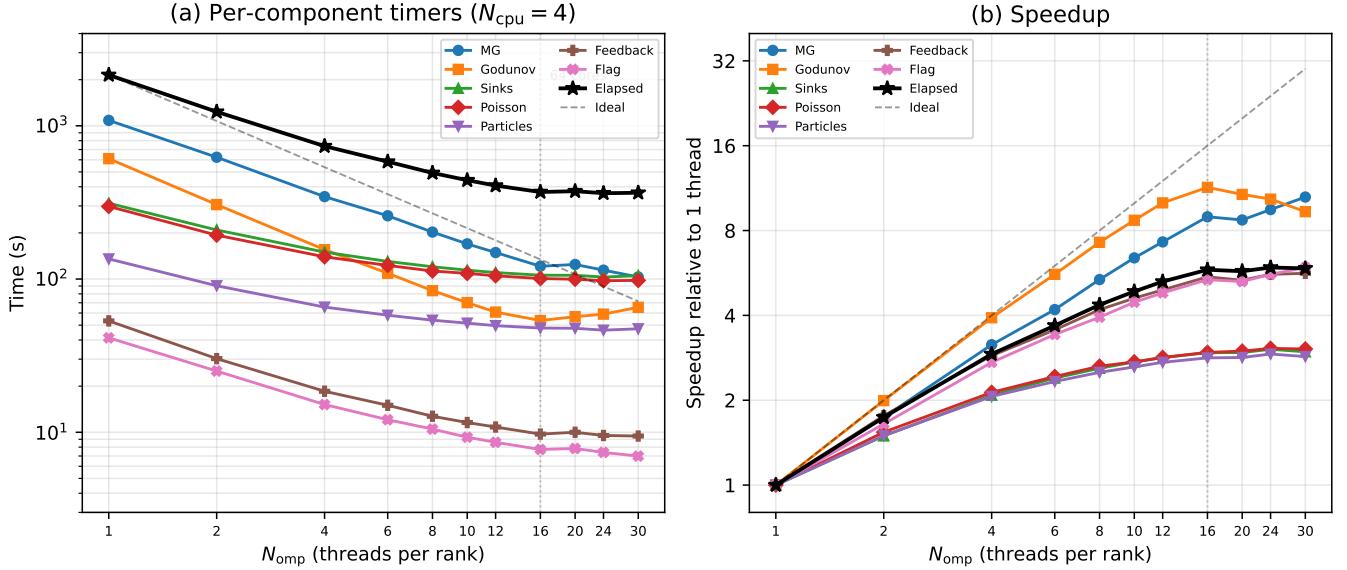


Figure 9. OpenMP thread scaling of cuRAMSES with $N_{\text{cpu}} = 4$ MPI ranks on a dual-socket AMD EPYC 7543 node (64 cores). (a) Per-component wall-clock times versus N_{omp} ; the dashed line shows ideal scaling from the single-thread baseline. The vertical dotted line marks the physical core limit (16 threads \times 4 ranks = 64 cores). (b) Speedup relative to 1 thread. The MG solver achieves $10.5\times$ speedup, while the overall elapsed time plateaus at $\sim 5.8\times$.

Table 7. Memory load balance for the strong-scaling test (54 M⁹⁷⁴ grids, 135.7 M particles). M_{\min} and M_{\max} are the minimum and maximum per-rank resident memory after the final load-balance remap (step 243). The ratio M_{\max}/M_{\min} measures memory imbalance. The level-10 grid counts illustrate the per-rank work imbalance at the most populated refined level (19.5 M grids total).

N_{cpu}	M_{\min} (Gb)	M_{\max} (Gb)	$\frac{M_{\max}}{M_{\min}}$	Lv-10 grids/rank ($\times 10^3$)		
				min	max	ratio
1	147.2	147.2	1.000	—	—	—
2	87.0	87.4	1.005	9120	10338	1.13 ⁹⁸⁰
4	50.1	50.5	1.008	4546	5210	1.15 ⁹⁸¹
8	28.0	28.6	1.021	2200	2675	1.22 ⁹⁸²
12	20.9	21.4	1.024	1370	1828	1.33 ⁹⁸³
16	16.7	17.1	1.024	1021	1376	1.35 ⁹⁸⁴
24	12.5	13.1	1.048	671	937	1.40 ⁹⁸⁵
32	10.7	11.1	1.037	521	711	1.36 ⁹⁸⁶
48	8.2	8.6	1.049	306	479	1.56 ⁹⁸⁷
64	8.2	8.6	1.049	225	362	1.61 ⁹⁸⁸
						989
						990
						991
						992
						993

level (level 10, containing 19.5 M grids), where the per-rank max/min ratio grows from 1.13 at 2 ranks to 1.61 at 64 ranks because the k -section sub-domains become smaller relative to the AMR clustering scale.

The low memory imbalance is key for production runs because each rank must pre-allocate arrays sized to its local N_{gridmax} ; a high imbalance forces over-allocation on lightly loaded ranks. With memory-weighted balancing, the 64-rank run requires only 8.6 Gb per rank at peak, compared with the 147.2 Gb needed in a single-rank run — a factor of 17.1 reduction, close to the ideal $64\times$ scaling modulo the ~ 4 Gb fixed per-rank overhead (MPI buffers, hash tables, coarse grid). The fixed overhead explains why per-rank memory saturates at ~ 8 Gb for 48 and 64 ranks.

The load-balance remap itself costs 16–26 s for $N_{\text{cpu}} \geq 8$ (Table 5), growing from 2.3 to 5.1 per cent of the total runtime as the computation shrinks under strong scaling — a modest price for near-perfect memory balance. Figure 10 visualizes these trends.

10 CONCLUSIONS

We have presented cuRAMSES, a set of algorithmic and implementation improvements to the RAMSES cosmological AMR code that collectively address the key scaling bottlenecks — communication overhead, memory consumption, solver efficiency, and hardware utilisation — encountered in large-scale cosmological simulations. While previous efforts such as OMP-RAMSES (Lee et al. 2021) and RAMSES-yOMP (Han et al. 2026) introduced MPI+OpenMP hybrid parallelism, cuRAMSES extends this to a three-level MPI+OpenMP+CUDA paradigm suited to the GPU-dominated architectures of current and upcoming exascale supercomputers. The main contributions are:

(i) **Recursive k -section domain decomposition.** A recursive k -ary spatial partitioning that replaces Hilbert curve ordering and enables hierarchical MPI communication with $\mathcal{O}(\sum_l k_l)$ messages per exchange, eliminating all MPI_ALLTOALL calls. The tree structure also provides a natural framework for memory-weighted load balancing, which reduces peak-to-mean memory imbalance from 2.5 to 1.3 in particle-heavy simulations.

(ii) **Morton key hash table.** A per-level open-addressing hash table that replaces the 48-byte-per-grid `nbor` array with $\mathcal{O}(1)$ hash lookups, saving over 190 MB per rank at $N_{\text{gridmax}} = 5$ M while simplifying the grid management code

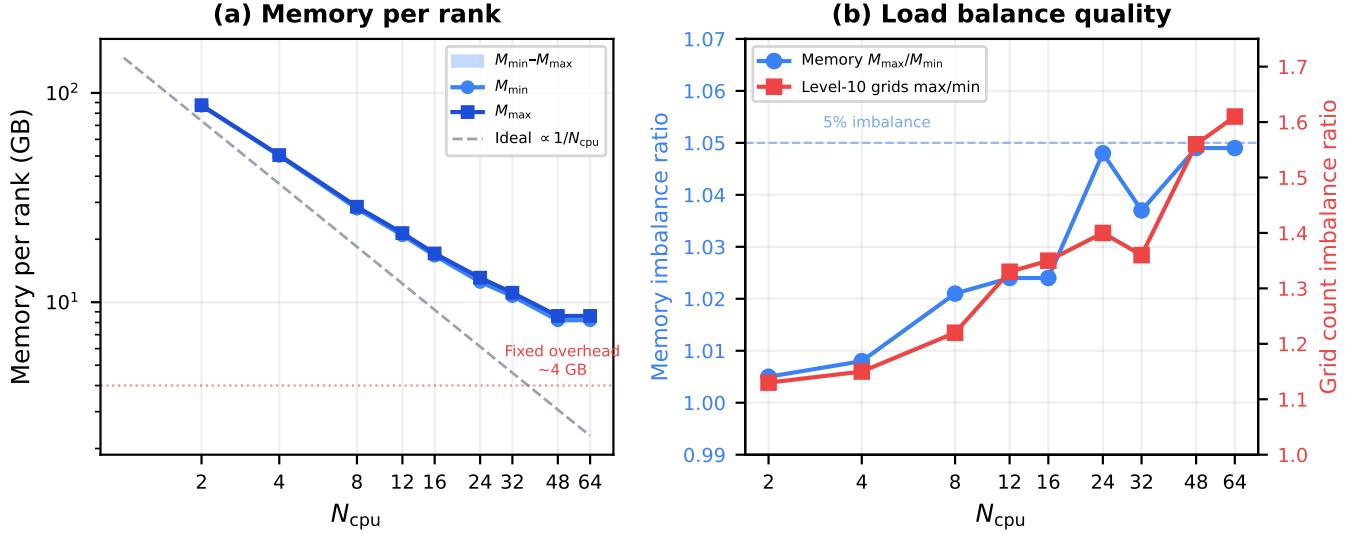


Figure 10. Memory-weighted load balance across the strong-scaling test suite. (a) Per-rank memory: M_{\min} and M_{\max} follow the ideal $1/N_{\text{cpu}}$ scaling (dashed grey) until a fixed per-rank overhead of ~ 4 GB dominates at high rank counts. (b) Load balance quality: the memory imbalance ratio M_{\max}/M_{\min} (blue, left axis) remains below 5 per cent for all rank counts tested (2–64), while the level-10 grid count imbalance (red, right axis) grows to 1.61 at 64 ranks — demonstrating that the memory-weighted cost function successfully decouples memory balance from grid-count balance.

(no neighbour-pointer maintenance during creation, deletion,¹⁰³⁸ or migration).

(iii) **Memory optimizations.** On-demand allocation of¹⁰⁴⁰ the Hilbert key, histogram, and defragmentation arrays re-¹⁰⁴¹duces steady-state memory by over 1 GB per rank, enabling¹⁰⁴² larger problems or finer resolution within the same hardware¹⁰⁴³ budget.

(iv) **Multigrid solver optimizations.** Precomputed¹⁰⁴⁵ neighbour caches, merged red-black Gauss–Seidel sweeps (re-¹⁰⁴⁶ducing communication by 44 per cent per iteration), fused¹⁰⁴⁷ residual-norm computation, and arithmetic optimizations¹⁰⁴⁸ reduce the Poisson solver’s share of total runtime from¹⁰⁴⁹ 55 per cent to 39 per cent.

(v) **Feedback spatial binning.** A spatial hash binning¹⁰⁵¹ scheme reduces both SNII and AGN feedback computations¹⁰⁵² from $\mathcal{O}(N_{\text{cells}} \times N_{\text{event}})$ to $\mathcal{O}(N_{\text{cells}} \times 27 \bar{n}_{\text{event/bin}})$, achieving¹⁰⁵³ speedups of one to two orders of magnitude.

(vi) **Variable-ncpu restart.** Both HDF5 parallel I/O¹⁰⁵⁵ and distributed binary I/O enable output/restart with ar-¹⁰⁵⁶bitrary rank counts, improving workflow flexibility for pro-¹⁰⁵⁷duction simulations on shared facilities. The binary path uses¹⁰⁵⁸ round-robin file assignment and per-level MPI_ALLTOALLV with¹⁰⁵⁹ reusable exchange metadata.

(vii) **GPU-accelerated solvers.** A dynamic dispatch¹⁰⁶¹ model in which OMP threads acquire GPU stream slots¹⁰⁶² at runtime, with fallback to CPU execution. Batched grid buffering amortises kernel launch latency, and an on-device flux reduction reduces PCIe transfer volume by 20×. The Go-¹⁰⁶³dunov solver achieves a 16 per cent speedup on an RTX 5000¹⁰⁶⁴ Ada GPU while producing bit-identical results.

All modifications preserve physical consistency, as verified¹⁰⁶⁶ by conservation-law diagnostics (e_{cons} , e_{pot} , e_{kin}) that are¹⁰⁶⁷ identical (or within MG tolerance) between the original and¹⁰⁶⁸ optimized codes.

The techniques described here are general and could be applied to other AMR codes that face similar scaling challenges. The Morton key hash table, in particular, is a drop-in replacement for any neighbour-pointer array in an octree code, requiring only that grid positions be available at each level. The k -section decomposition can be adopted by any code whose domain decomposition is currently based on one-dimensional space-filling curve ordering. The hybrid CPU/GPU dispatch model demonstrates that GPU acceleration can be integrated into a legacy Fortran codebase without sacrificing portability: the same source compiles and runs correctly in CPU-only mode when CUDA is unavailable, making it practical for heterogeneous computing environments where not all nodes are equipped with GPUs.

Looking ahead, the three-level MPI+OpenMP+CUDA parallelism positions CURAMSES well for exascale platforms such as Frontier, Aurora, and LUMI, where the majority of floating-point throughput resides in GPU accelerators. Further optimisation of the host-to-device data transfer — currently the dominant cost in the GPU pipeline — through persistent device-side data structures and asynchronous prefetching will be a key focus of future work.

CURAMSES is being used in production for the Horizon Run 5 cosmological simulation project and will be made publicly available upon completion of the benchmark campaign.

ACKNOWLEDGEMENTS

This work was supported by the Korea Institute for Advanced Study. Computational resources were provided by the KIAS Center for Advanced Computation. The author thanks Romain Teyssier for the public release of the RAMSES code and the RAMSES developer community for continued maintenance and improvements.

1070 DATA AVAILABILITY

1071 The modified code is available at [1103](https://github.com/)
 1072 [kjhan0606/cuRAMSES-kjhan](https://github.com/kjhan0606/cuRAMSES-kjhan). Test configurations and anal-[1104](#)
 1073 ysis scripts will be shared upon reasonable request to the[1105](#)
 1074 author.[1106](#)

1075 REFERENCES

- 1076 Bryan G. L., et al., 2014, ApJS, 211, 19
 1077 Dubois Y., et al., 2014, MNRAS, 444, 1453
 1078 Dubois Y., et al., 2021, A&A, 651, A109
 1079 Guillet T., Teyssier R., 2011, J. Comput. Phys., 230, 4756
 1080 Hahn O., Abel T., 2011, MNRAS, 415, 2101
 1081 Han S., et al., 2026, A&A, 705, A169
 1082 Hopkins P. F., 2015, MNRAS, 450, 53
 1083 Hopkins P. F., et al., 2018, MNRAS, 480, 800
 1084 Knuth D. E., 1997, The Art of Computer Programming, Vol. 3:
 Sorting and Searching, 2nd edn. Addison-Wesley, Reading, MA
 1085 Lee J., et al., 2021, ApJ, 908, 11
 1086 Morton G. M., 1966, A Computer Oriented Geodetic Data Base
 and a New Technique in File Sequencing. IBM, Ottawa
 1087 Nelson D., et al., 2019, Comput. Astrophys. Cosmol., 6, 2
 1088 Pillepich A., et al., 2018, MNRAS, 473, 4077
 1089 Schaye J., et al., 2015, MNRAS, 446, 521
 1090 Springel V., 2005, MNRAS, 364, 1105
 1091 Springel V., 2010, MNRAS, 401, 791
 1092 Teyssier R., 2002, A&A, 385, 337
 1093 Vogelsberger M., et al., 2014, MNRAS, 444, 1518
 1094 Warren M. S., Salmon J. K., 1993, in Proc. Supercomputing '93.
 1095 ACM, New York, p. 12
 1096
 1097

1102 APPENDIX B: MORTON KEY ENCODING
1103 DETAILS

The Morton key interleaving for a single coordinate value v with $B = 21$ bits is computed by the following bit-manipulation loop:

Algorithm 3 Morton key encoding of (i_x, i_y, i_z)

Input: Integer coordinates (i_x, i_y, i_z)

Output: 63-bit Morton key M

- 1: $M \leftarrow 0$
- 2: **for** $b = 0$ **to** $B - 1$ **do**
- 3: $M \leftarrow M \mid (\text{bit}_b(i_x) \ll 3b)$
- 4: $M \leftarrow M \mid (\text{bit}_b(i_y) \ll (3b + 1))$
- 5: $M \leftarrow M \mid (\text{bit}_b(i_z) \ll (3b + 2))$
- 6: **end for**

The neighbour key computation decodes, shifts the appropriate coordinate, applies periodic wrapping, and re-encodes:

Algorithm 4 Morton neighbour key in direction j

Input: Morton key M , direction j , grid counts (n_x, n_y, n_z)

Output: Neighbour Morton key M' (or -1 if out of bounds)

- 1: $(i_x, i_y, i_z) \leftarrow \text{DECODE}(M)$
- 2: Adjust i_d by ± 1 according to direction j
- 3: Apply periodic wrapping: $i_d \leftarrow i_d \bmod n_d$
- 4: $M' \leftarrow \text{ENCODE}(i_x, i_y, i_z)$

1098 APPENDIX A: K-SECTION TREE WALK
1099 PSEUDOCODE

1100 Algorithm 2 gives the pseudocode for mapping a spatial po-
 1101 sition to its owning MPI rank by walking the k -section tree.

Algorithm 2 CPU map computation via k-section tree walk

Input: Position \mathbf{x} , tree arrays

Output: CPU rank c

- 1: node \leftarrow root
- 2: $l \leftarrow 0$
- 3: **while** node is not a leaf **do**
- 4: $l \leftarrow l + 1$
- 5: $k \leftarrow k_l$; dir $\leftarrow \text{ksec_dir}(l)$
- 6: child $\leftarrow k$
- 7: **for** $j = 1$ **to** $k - 1$ **do**
- 8: **if** $x_{\text{dir}} \leq \text{ksec_wall}(\text{node}, j)$ **then**
- 9: child $\leftarrow j$; **break**
- 10: **end if**
- 11: **end for**
- 12: node $\leftarrow \text{ksec_next}(\text{node}, \text{child})$
- 13: **end while**
- 14: $c \leftarrow \text{ksec_indx}(\text{node})$