MNRAS 000, 1–15 (2026) Preprint 19 February 2026 Compiled using MNRAS LATEX style file v3.2

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

Juhan Kim[1]⋆
[1]Center 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-$N_{\rm cpu}$ 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 ΛCDM 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 $N_{\rm cpu}$ 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 $N_{\rm cpu}$. 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 $\mathcal{O}(N_{\rm cpu}^2)$ message complexity and $\mathcal{O}(N_{\rm cpu})$ per-rank buffer memory — a severe bottleneck when $N_{\rm cpu}$ 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

particles in a dense halo is far more expensive in memory than a void cell with zero particles, yet the standard load balancer treats them equally (Springel 2005). The problem is particularly acute in cosmological zoom-in simulations (Dubois et al. 2021), where the high-resolution region occupies a small fraction of the total volume: the Hilbert curve concentrates nearly all refinement on a few ranks while the remaining ranks are left with low-resolution void cells, leading to severe load imbalance that worsens with increasing zoom factor.

Beyond the communication and load-balancing challenges, several other bottlenecks arise. Large per-rank arrays scale linearly with $N_{\rm gridmax}$: the neighbour-pointer array `nbor(`$N_{\rm gridmax}$`, 6)` alone consumes $48\,N_{\rm gridmax}$ bytes (Teyssier 2002), and the Hilbert key array adds another $16\,N_{\rm gridmax}$ bytes (when compiled with `QUADHILBERT`), together approaching 1 GB per rank for production configurations with $N_{\rm gridmax} \sim 5$ M. The multigrid Poisson solver (Guillet & Teyssier 2011) typically dominates runtime, its per-iteration cost driven by frequent ghost-zone exchanges and repeated hash table lookups for neighbour grids. Finally, RAMSES writes one file per MPI rank, so restarting with a different rank count is practically infeasible: the AMR tree structure — parent–child links, neighbour pointers, and per-rank communication tables — is tightly coupled to the original domain decomposition and cannot be reconstructed without re-reading and redistributing every grid from scratch.

The original RAMSES code relies exclusively on MPI for parallelism, assigning one MPI rank per processor core. To exploit the shared-memory bandwidth of modern multi-core nodes, hybrid MPI+OpenMP implementations have been developed: the Horizon Run 5 simulation (Lee et al. 2021) employs OMP-RAMSES, and the NewCluster zoom-in simulation (Han et al. 2026) employs RAMSES-yOMP, both adding OpenMP threading within each MPI rank for improved intra-node scalability. However, MPI+OpenMP alone still leaves the GPU compute capability of modern heterogeneous nodes untapped. With the emergence of exascale supercomputers whose floating-point throughput is dominated by GPU accelerators, a three-level MPI+OpenMP+CUDA parallelism is essential to fully utilise the available hardware.

In this paper we describe cuRAMSES, a comprehensive set of modifications to RAMSES that addresses each of these challenges. We introduce a recursive k-section domain decomposition (Section 2) that replaces Hilbert ordering with a hierarchical multi-way spatial partitioning, enabling MPI exchange with $\mathcal{O}(\sum_l k_l)$ messages per operation. A Morton key hash table (Morton 1966) (Section 4) eliminates the `nbor` array entirely, saving ∼240 MB per rank, and on-demand allocation of redundant large arrays (Section 5) yields over 1 GB of additional savings. Algorithmic optimizations to the multigrid Poisson solver (Section 6) reduce its share of total runtime from 55 per cent to 39 per cent, while a spatial hash binning scheme (Section 7) accelerates Type II supernova and AGN feedback by orders of magnitude. We also implement variable-$N_{\rm cpu}$ restart for both HDF5 and binary formats (Section 8), along with miscellaneous improvements described in Section 9. A hybrid CPU/GPU dispatch model (Section 10) dynamically offloads compute-intensive routines to GPU streams at runtime. Performance benchmarks are presented in Section 11, and we conclude in Section 12.

Throughout this paper, we use the notation of Teyssier (2002): $N_{\rm levelmax}$ is the maximum AMR level, $N_{\rm gridmax}$ is the maximum number of grids per rank, `twotondim` $= 2^{N_{\rm dim}} = 8$ is the number of cells per oct in three dimensions, and $N_{\rm cpu}$ 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 $N_{\rm cpu}$ 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 $\mathcal{O}(N_{\rm cpu}^2)$ 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 ∼640 MB at $N_{\rm gridmax} = 5 \times 10^6$.

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 $N_{\rm cpu}$ MPI ranks, we first compute the prime factorization as

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

The splitting sequence is then

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

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 $N_{\rm cpu}$ 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, $N_{\rm cpu} = 12 = 3 \times 2 \times 2$ 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 $N_{\rm cpu} = 12$, from the undivided domain through three successive levels of splitting, with the corresponding k-section tree shown beneath each panel.

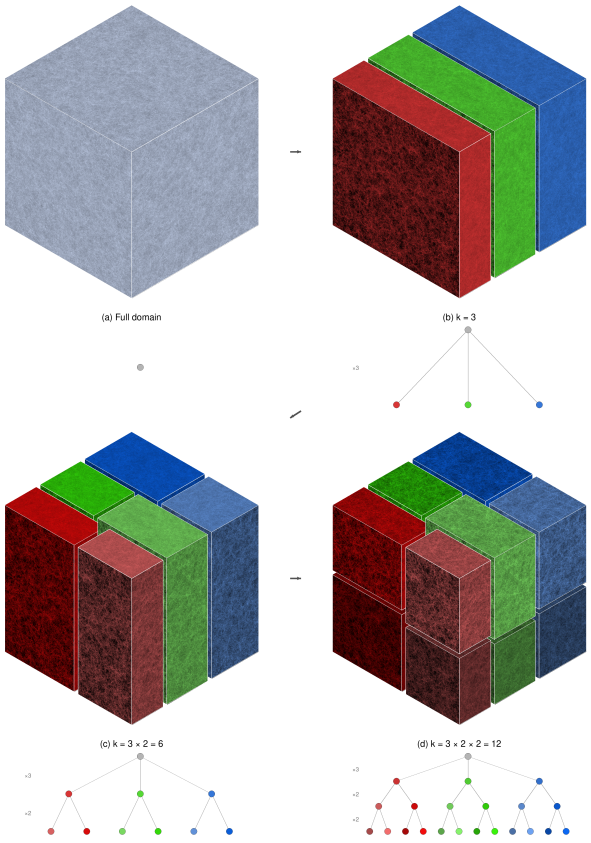

Figure 1. Progressive recursive k-section decomposition for $N_{\rm cpu} = 12 = 3 \times 2 \times 2$. (a) The undivided simulation domain. (b) First split into $k = 3$ slabs along the longest axis. (c) Each slab bisected along the second axis ($3 \times 2 = 6$ sub-domains). (d) Final bisection along 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 ∼10–30 %, reflecting load-balanced wall placement.

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 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 is

$$N_{\rm nodes} = 1 + \sum_{l=1}^{L} \prod_{i=1}^{l} k_i \leq 1 + L \cdot k_{\rm max}^{L}, \tag{3}$$

which for practical values ($N_{\rm cpu} \leq 10^5$) is at most a few hundred, negligible overhead.

2.3 Load-Balanced Wall Placement

When the tree is updated during load balancing (every `nremap` coarse steps), the wall positions are adjusted by iterative dichotomy 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 $\Delta x_{\rm hist}$.

(ii) For each of the $k_l - 1$ walls within each node, a binary search (dichotomy) 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 $[i_{\rm min}, i_{\rm max}]$ with total count $n = i_{\rm max} - i_{\rm min} + 1$ is

$$f_j = n^{-1} \sum_{m=1}^{j} n_m, \tag{4}$$

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 $N_{\rm cpu}$ ranks to obtain the global cumulative load at each wall position. The dichotomy converges when the relative load imbalance $|\hat{L}_j - L_j^{\rm target}|/L_j^{\rm target}$ falls below a tolerance $\epsilon_{\rm tol}$ (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:

$$C_{\rm cell} = 2^{-N_{\rm dim}} \left(w_{\rm grid} + n_{\rm part}({\rm igrid}) \cdot w_{\rm part} \right), \tag{5}$$

where $w_{\rm grid}$ is the memory cost per grid slot (default 270 bytes, accounting for hydro, gravity, and AMR bookkeeping arrays), $w_{\rm part}$ is the memory cost per particle slot (default 12 bytes for position, velocity, mass, and linked-list pointers), and $n_{\rm part}({\rm igrid})$ is the number of particles attached to grid `igrid`. The division by $2^{N_{\rm dim}}$ 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 $e_{\rm cons}$, $e_{\rm pot}$, $e_{\rm kin}$ to machine precision).

3 HIERARCHICAL MPI COMMUNICATION

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

3.1 Exclusive Exchange

In the exclusive exchange, each item has a unique destination rank. The algorithm walks the k-section tree from root to leaf, where each *node* represents a contiguous group of MPI ranks at a given tree level. The *root* node encompasses all $N_{\rm cpu}$ ranks and represents the entire computational domain. At each level l, a node is partitioned into k_l child nodes; the leaf nodes at the bottom of the tree correspond to individual 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

W ← **S**; **D** ← **d**; node ← root
for $l = 1$ **to** L **do**
  $k \leftarrow k_l$; my_child ← `ksec_cpu_path`(myid, l)
  Classify items in **W** by child index (counting sort on **D**)
  Identify $k-1$ correspondent ranks (one per sibling child)
  for each correspondent p **do**
    Exchange count: `MPI_ISEND/IRECV` (tag $100+l$)
    Exchange data: `MPI_ISEND/IRECV` (tags $200+l$, $300+l$)
  end for
  `MPI_WAITALL`
  W ← merge(my_child items, received items)
  node ← `ksec_next`(node, my_child)
end for
R ← **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

$$N_{\rm msg} = \sum_{l=1}^{L}(k_l - 1) = \sum_i m_i(p_i - 1), \tag{6}$$

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

Figure 2 illustrates the communication pattern for $N_{\rm cpu} = 12\,(= 3\times2\times2)$. 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 ↔ rank 5), while step 2 simultaneously pairs children 1 and 3 (dark red arcs, e.g. rank 1 ↔ rank 9) and children 2 and 3 (orange arcs, e.g. rank 5 ↔ 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 ↔ rank 3). Finally, at level 3 ($k_3 = 2$), only adjacent pairs communicate (e.g. rank 1 ↔ 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 $N_{\rm msg} = 2+1+1 = 4$ messages per exchange, independent of $N_{\rm cpu}$.

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 ($s = 1,\ldots,p-1$), child $s+1$ is paired with each of the children $1,\ldots,s$: each rank in child $s+1$ exchanges data with its correspondent in child c for $c = 1,\ldots,s$, 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 $N_{\rm cpu} = 12$ 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 $L \times V_{\rm local}$, where $V_{\rm local}$ 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 $\mathcal{O}(N_{\rm cpu})$ to $\mathcal{O}(\sum_l k_l)$, more than compensates for the modest increase in aggregate volume, particularly at large $N_{\rm cpu}$ 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.

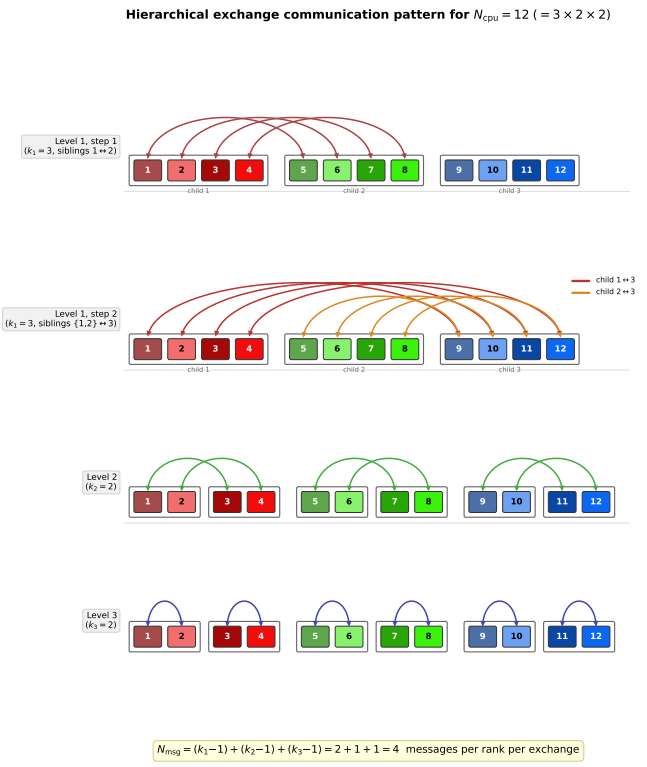

Figure 2. Hierarchical exchange communication pattern for $N_{\mathrm{cpu}} = 12\,(= 3 \times 2 \times 2)$. Coloured rectangles represent MPI ranks numbered 1–12, using the same colour scheme as Figure 1. Arcs denote bidirectional point-to-point exchanges. At level 1 ($k_1 = 3$), two steps connect each rank with correspondents in the two sibling subtrees; the two colours in step 2 distinguish the children 1↔3 (dark red) and children 2↔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 $N_{\mathrm{msg}} = (3{-}1) + (2{-}1) + (2{-}1) = 4$.

The data packing format for each ghost grid is:

$$\mathrm{sendbuf}(1 : 2^{N_{\mathrm{dim}}}{+}2,\, i) = \{u_1, \ldots, u_{2^{N_{\mathrm{dim}}}},\, \mathrm{sender\_id},\, \mathrm{index}\}, \tag{7}$$

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:

$$\mathrm{sendbuf}\big((v-1)2^{N_{\mathrm{dim}}} + j,\, i\big) = \mathrm{xx}(v,\, \mathrm{cell}_{i,j}), \tag{8}$$

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

Table 1. 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	$\mathcal{O}(N_{\mathrm{cpu}})$	$\mathcal{O}(\sum_l k_l)$
Buffer memory	$\mathcal{O}(N_{\mathrm{cpu}} \cdot N_{\mathrm{ghost}})$	$\mathcal{O}(k_{\max} \cdot N_{\mathrm{ghost}})$
`MPI_ALLTOALL` calls	≥ 1 per exchange	0

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.

3.4 Complexity Analysis

Table 1 summarizes the communication complexity of the original RAMSES and cuRAMSES.

4 MORTON KEY OCTREE FOR NEIGHBOUR LOOKUP

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

$$M_{\mathrm{nbor}} = 6 \times N_{\mathrm{gridmax}} \times 8\,\mathrm{bytes} = 48\,N_{\mathrm{gridmax}}\,\mathrm{bytes}. \tag{9}$$

For $N_{\mathrm{gridmax}} = 5\,\mathrm{M}$, this is 240 MB per rank. 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

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:

$$M(i_x, i_y, i_z) = \sum_{b=0}^{B-1} \Big[\mathrm{bit}_b(i_x){\cdot}2^{3b} + \mathrm{bit}_b(i_y){\cdot}2^{3b+1} + \mathrm{bit}_b(i_z){\cdot}2^{3b+2}\Big], \tag{10}$$

where $B = 21$ bits per coordinate and $\mathrm{bit}_b(n)$ 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 $2^{21} - 1 = 2\,097\,151$, 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.

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

$$i_d = \lfloor x_{g,d} \cdot 2^{l-1} \rfloor, \quad d \in \{x, y, z\}, \tag{11}$$

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 4.4) 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.

4.3 Neighbour Finding via Morton Arithmetic

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): $M_{\text{child}} = (M \ll 3) \,|\, i_{\text{child}}$.

4.4 Per-Level Hash Table

We maintain one open-addressing hash table per AMR level, mapping Morton keys to grid indices:

$$\text{mort\_table}(l) : M \mapsto \text{igrid}. \tag{12}$$

The hash function uses multiplicative hashing with Knuth's golden ratio constant and an additional mixing step:

$$h(M) = \left[\left((M \times \phi_1) \oplus (M \gg 16) \right) \times \phi_2 \right] \oplus (h \gg 13), \tag{13}$$

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 rehashing (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.

4.5 Replacement Functions

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.

The `nbor` array is reduced to `allocate(nbor(1:1, 1:1))` — effectively eliminated while maintaining compilation compatibility with any remaining references.

4.6 Memory and Performance Analysis

The memory cost of the hash table is

$$M_{\text{hash}} \approx \frac{N_{\text{grids}}}{0.7} \times (8 + 4) \text{ bytes} \approx 17\, N_{\text{grids}} \text{ bytes}, \tag{14}$$

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

Compared to the original `nbor` cost of $48 \times N_{\text{gridmax}}$ bytes, the net savings are

$$\Delta M = 48\, N_{\text{gridmax}} - 4\, N_{\text{gridmax}} - 17\, N_{\text{grids}} \approx 44\, N_{\text{gridmax}} - 17\, N_{\text{grids}}. \tag{15}$$

Since $N_{\text{grids}} \ll N_{\text{gridmax}}$ in practice (typical occupancy is 30–60 per cent), the savings are substantial: $\sim$176 MB for $N_{\text{gridmax}} = 5$ M at 50 per cent occupancy.

The computational cost of a hash lookup is $\mathcal{O}(1)$ expected time, with worst-case linear probing bounded by the load factor. In practice, the precomputed neighbour caches described in Section 6 amortize any per-lookup overhead in the performance-critical Poisson solver.

5 MEMORY OPTIMIZATIONS

Beyond the Morton key hash table, several additional optimizations reduce the steady-state memory footprint.

5.1 Hilbert Key Elimination

When using k-section ordering, the Hilbert key array `hilbert_key(1:ncell)` is no longer needed for domain decomposition. We replace it with `allocate(hilbert_key(1:1))`, saving

$$\Delta M_{\text{hilbert}} = 16 \times N_{\text{gridmax}} \times 2^{N_{\text{dim}}} \text{ bytes} \tag{16}$$

under `QUADHILBERT` (128-bit keys stored as two 64-bit integers). For $N_{\text{gridmax}} = 5$ M, this is approximately 640 MB.

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

Table 2. Memory savings per MPI rank for $N_{\rm gridmax} = 5\,\mathrm{M}$. Savings marked with * are conditional on using k-section ordering.

Optimization	Savings (MB)	Availability
`nbor` elimination (Morton hash)	240	Always
`hilbert_key` elimination*	640	Steady state
On-demand `bisec_ind_cell`*	160	Between LB steps
On-demand `cell_level`*	160	Between LB steps
Defrag scratch (local)	40	Between defrag
Total	**>1200**	

5.2 On-Demand Histogram Arrays

The arrays `bisec_ind_cell` and `cell_level`, each of size $N_{\rm gridmax} \times 2^{N_{\rm dim}}$ integers (8 bytes), are used exclusively during load balancing to build the bisection histogram. We allocate them on entry to `init_bisection_histogram` and deallocate them after `cmp_new_cpu_map` returns. The savings are

$$\Delta M_{\rm hist} = 2 \times 8 \times N_{\rm gridmax} \times 2^{N_{\rm dim}} \approx 320\,\mathrm{MB} \tag{17}$$

for $N_{\rm gridmax} = 5\,\mathrm{M}$. Since load balancing occurs only every `nremap` coarse steps, these arrays are absent during the vast majority of the simulation.

5.3 Memory Savings Summary

Table 2 summarizes the memory savings for $N_{\rm gridmax} = 5\,\mathrm{M}$.

The reported memory savings of the `nbor` array account for both the eliminated array ($48\,N_{\rm gridmax} = 240\,\mathrm{MB}$) and the hash table overhead ($\sim 17\,N_{\rm grids}$, typically <50 MB). Net savings are at least 190 MB. The Hilbert key savings of 640 MB assume `QUADHILBERT`; for standard 64-bit keys the savings would be 320 MB.

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.

6 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.

6.1 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:

$$\texttt{nbor\_grid\_fine}(j,\, i) = \texttt{morton\_nbor\_grid}(\texttt{igrid\_amr}(i),\, l,\, j), \tag{18}$$

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

6.2 Branch-Free Neighbour Access

The original GS kernel 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.

6.3 Merged Red-Black Exchange

Standard red-black Gauss–Seidel smoothing in RAMSES performs a ghost-zone exchange of the potential ϕ between the red and black sweeps:

$$\text{Red} \to \text{Exchange}(\phi) \to \text{Black} \to \text{Exchange}(\phi). \tag{19}$$

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:

$$\text{Red} \to \text{Black} \to \text{Exchange}(\phi). \tag{20}$$

This is a form of *chaotic relaxation*: boundary cells in the black sweep use slightly stale ghost 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 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.

6.4 Fused Residual and Norm Computation

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.

6.5 Arithmetic Optimization

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

$$\phi_{\rm new} = \frac{\sum_j \phi_j - h^2 f}{2N_{\rm dim}}. \quad (21)$$

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

6.6 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 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.

7 FEEDBACK SPATIAL BINNING

7.1 The Brute-Force Bottleneck

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

- `average_SN`: averages hydrodynamic quantities within the blast radius of each SN event, accumulating volume, momentum, kinetic energy, mass loading, and metal loading. The original implementation loops over all cells × all SNe, yielding $\mathcal{O}(N_{\rm cells} \times N_{\rm SN})$ complexity.
- `Sedov_blast`: injects the blast energy and ejecta into cells within the blast radius. Same $\mathcal{O}(N_{\rm cells} \times N_{\rm SN})$ complexity.

In production simulations with ∼2000 simultaneous SN events, these routines consume 66 s and 11 s per call respectively, dominating the feedback time step.

7.2 Spatial Hash Binning

We partition the simulation domain into a uniform grid of $n_{\rm bin}^3$ bins, where

$$n_{\rm bin} = \max\big(1,\, \min(128,\, \lfloor L_{\rm box}/r_{\rm max} \rfloor)\big), \quad (22)$$

and $r_{\rm max}$ is the maximum SN blast radius (the larger of `rcell` × `dx_min` and `rbubble`). Each SN event is assigned to a bin based on its position, and a linked list threads the events within each bin:

$$\text{bin\_head}(i_x, i_y, i_z) \to \text{SN}_1 \to \text{SN}_2 \to \cdots \quad (23)$$

For each cell, we compute its bin index and check only the 27 neighbouring bins (the cell's own bin plus its 26 face-, edge-, and corner-adjacent bins). Since $r_{\rm max}$ is at most the bin size by construction, this 27-bin neighbourhood is guaranteed to contain all SNe that could influence the cell. The complexity becomes

$$\mathcal{O}(N_{\rm cells} \times \bar{n}_{\rm SN/bin} \times 27), \quad (24)$$

where $\bar{n}_{\rm SN/bin} = N_{\rm SN}/n_{\rm bin}^3$ is the average number of SNe per bin.

7.3 Parallelization

7.3.1 Cell-parallel average_SN

The binned `average_SN` uses cell-parallel OpenMP threading: the outer loop is over grids (with `!$omp parallel do`), and each thread processes the cells of one grid. When a cell falls within an SN blast radius, the thread accumulates its contribution using `!$omp atomic` directives on the shared SN-indexed arrays (`vol_gas`, `dq`, `ekBlast`, etc.). The atomic overhead is minimal because collisions are rare — most bins contain zero or one SN, so contention is low.

7.3.2 Grid-parallel Sedov_blast

The `Sedov_blast` routine writes only to cells owned by each grid, so no atomics are needed. The outer loop is over grids, and each thread independently processes the cells of its assigned grids, checking only the 27 neighbouring bins for relevant SNe.

7.4 Performance Results

With approximately 2000 simultaneous SN events:

- `average_SN`: 66 s → 0.25 s (∼260× speedup)
- `Sedov_blast`: 11 s → 0.07 s (∼157× speedup)

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

7.5 AGN Feedback Spatial Binning

The same spatial binning technique is applied to the AGN feedback routines (`average_AGN` and `AGN_blast`), which suffer from the same $\mathcal{O}(N_{\rm cells} \times N_{\rm AGN})$ 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 (§7.2), 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.

8 VARIABLE-NCPU RESTART AND OTHER IMPROVEMENTS

8.1 HDF5 Parallel I/O

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

8.2 Variable-Ncpu Restart Algorithm

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

(i) Build a uniform k-section tree for the new $N_{\rm cpu}$ (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-balance operation redistributes grids for optimal balance under the new rank configuration.

This approach requires that all ranks temporarily hold the full grid metadata (positions and son flags) during the reconstruction phase. For typical production outputs (∼10 M total grids), this temporary overhead is a few hundred MB — well within the memory budget freed by the optimizations of Sections 4–5.

8.3 Binary Distributed I/O Restart

When HDF5 is unavailable or the native binary format (`informat='origin'`) is preferred, a distributed I/O strategy enables variable-$N_{\rm cpu}$ restart from the per-rank binary files written by standard RAMSES. The binary format stores one file per MPI rank — `amr_XXXXX.outYYYYY`, `hydro_XXXXX.outYYYYY`, etc. — so the number of files equals $N_{\rm cpu}^{\rm file}$, which may differ from the current $N_{\rm cpu}$.

The restart proceeds in three stages.

8.3.0.1 Stage 1: Early detection and file assignment. Rank 1 probes the header of file 00001 to read $N_{\rm cpu}^{\rm file}$. If $N_{\rm cpu}^{\rm file} \neq N_{\rm cpu}$, the variable-$N_{\rm cpu}$ path is activated (requiring k-section ordering). The $N_{\rm cpu}^{\rm file}$ files are distributed among the $N_{\rm cpu}$ ranks by round-robin assignment: rank r reads files whose index satisfies $(f-1) \bmod N_{\rm cpu} = r-1$.

8.3.0.2 Stage 2: Distributed AMR reconstruction. Each rank reads only its assigned AMR files, extracting per-level active grid metadata (positions $\mathbf{x}_g$ and son flags). The per-level active grid counts are aggregated via `MPI_ALLREDUCE`. For each level, the grids read by each rank are assigned to their correct owner by evaluating `cmp_ksection_cpumap` on the father cell position (computed from $\mathbf{x}_g$ and the parent–child octant relationship). An `MPI_ALLTOALLV` exchange then routes each grid's data to its owner, who creates the local AMR grid. The exchange metadata — send ordering and receive-grid mapping — is stored for reuse.

8.3.0.3 Stage 3: Hydro and gravity scatter. The hydro and gravity binary files are read in the same distributed fashion: each rank reads only its assigned files and packs the cell-centred data into per-level send buffers using the stored send ordering. The same `MPI_ALLTOALLV` counts and displacements from Stage 2 are reused, and the receive-grid mapping scatters incoming data to the correct local cells. Since the binary format stores primitive variables (density, velocity, pressure), a primitive-to-conservative conversion is applied on the receiving side.

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

This three-stage approach ensures that no rank reads more than $\lceil N_{\rm cpu}^{\rm file}/N_{\rm cpu} \rceil$ files (at most two for practical configurations), and the `MPI_ALLTOALLV` exchange per level has cost proportional to the number of grids exchanged rather than the total number of ranks. Verification tests confirm $e_{\rm cons} = 0$ for both upward ($4 \to 12$) and downward ($12 \to 4$) rank-count changes.

8.4 Stream-Access IC Reading

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

$$\text{offset} = H + (i-1) \times P + 5. \tag{25}$$

This is applied to all IC file types: density perturbation (`deltab`), velocity components (`velcx/y/z`), particle positions (`poscx/y/z`), and temperature.

8.5 Sink Particle Refinement Fix

We identified and fixed a bug in the sink particle refinement criterion. The original implementation in `cic_amr` added the

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

`nremap`	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%

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.

9 ADDITIONAL IMPLEMENTATION DETAILS

9.1 Nremap Tuning

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

The optimal value `nremap = 5` balances the cost of rebalancing against the growing imbalance that accumulates between rebalancing steps. Higher values (`nremap` $= 10$) reduce LB overhead further but allow imbalance to grow enough to slow other operations, resulting in a net increase in total runtime.

9.2 Load-Balance Profiling

To identify bottlenecks in the load-balancing procedure, we added internal timing instrumentation that reports the wall-clock time of each phase:

(i) `numbp_sync`: MPI synchronization of particle counts for virtual grids.
(ii) `cmp_new_cpu_map`: histogram construction and wall finding.
(iii) `expand_pass`: ghost-zone expansion after grid migration.
(iv) `grid_migration`: actual grid transfer between ranks.
(v) `allreduce+cpumap_update`: global reduction and CPU map reconstruction.
(vi) `shrink_pass`: removal of migrated grids from source rank.

Profiling reveals that `allreduce+cpumap_update` dominates, consuming approximately 50 per cent of the total load-balance time. This motivates the `nremap = 5` default, as reducing the frequency of these expensive global operations has a disproportionate impact on total runtime.

9.3 Pre-Allocated Buffer Pools

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 semantics: 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.

10 HYBRID CPU/GPU DISPATCH

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.

10.1 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 **superbatch 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.

10.2 Superbatch Buffering and Scatter-Reduce

GPU kernel launch latency (∼10–50 μs) is amortised by batching: each GPU thread accumulates the full stencil data 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 **scatter-reduce** kernel 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× reduction in PCIe bandwidth.

10.3 Lock-Free Level $L-1$ Update

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.

10.4 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).

10.5 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 × 2 OMP threads, the Godunov solver is accelerated by 16 per cent (22.0 s → 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.

11 PERFORMANCE RESULTS

11.1 Test Configuration

All tests use a cosmological ΛCDM 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.

11.2 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 $e_{\rm cons}$ for the MG-optimized version (3.79×10^{-3} versus 3.77×10^{-3}) is attributable to the chaotic relaxation 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.

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

Configuration	$e_{\rm cons}$	$e_{\rm pot}$	$e_{\rm kin}$
Reference (Hilbert)	3.77×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
K-section (no membal)	3.77×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
K-section (membal)	3.77×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
Morton hash + ksection	3.77×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
MG optimizations	3.79×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}

11.3 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-$N_{\rm cpu}$ restart feature (§8) 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 $N_{\rm cpu} = 1$–64. The elapsed time is the total wall-clock time including HDF5 I/O and load balancing. All configurations yield energy conservation errors $e_{\rm cons}$ 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 (∼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× at 64 ranks relative to the single-rank baseline, demonstrating excellent scaling efficiency (53%) on this shared-memory node.

Figure 3 visualises these trends. Panel (a) shows the elapsed time and per-component timer values as a function of $N_{\rm cpu}$ 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× at 64 ranks.

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.

$N_{\rm cpu}$	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

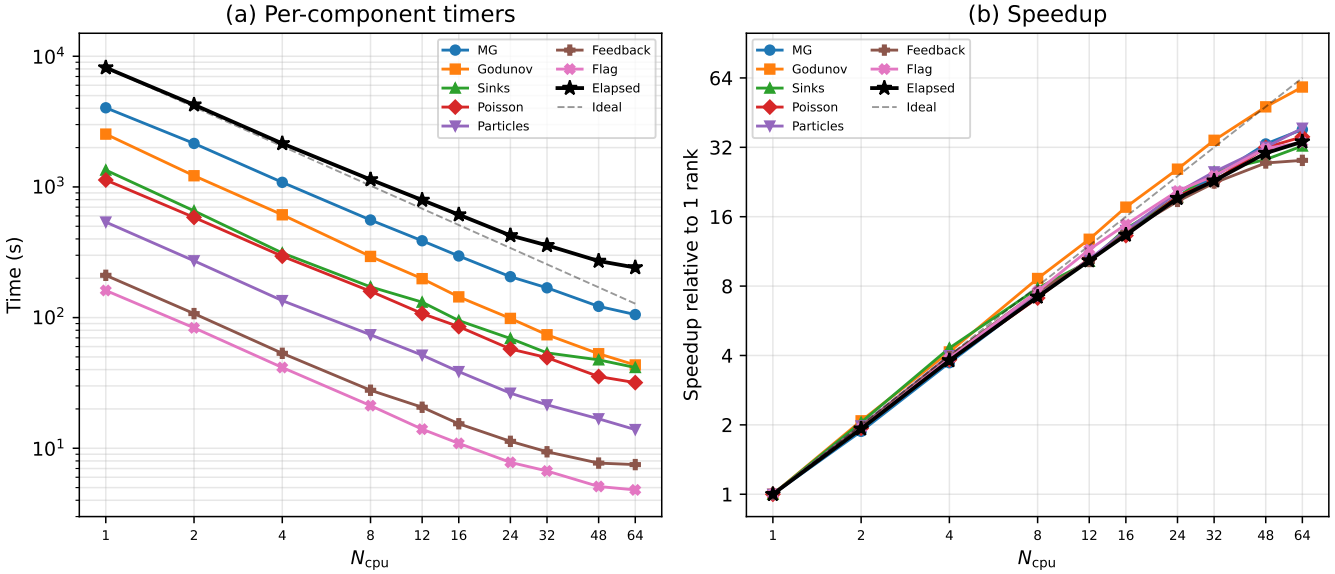

Figure 3. Strong scaling of cuRAMSES on a dual-socket AMD EPYC 7543 node (64 cores) with a 135.7 M particle cosmological simulation including full physics. (a) Elapsed time and per-component wall-clock times versus $N_{\rm cpu}$; the dashed line shows ideal scaling from the single-rank baseline. (b) Speedup relative to 1 rank. Particle operations and the flag routine scale near-ideally, while the overall speedup reaches 33.9× at 64 ranks (53% parallel efficiency).

11.4 OpenMP Thread Scaling

To evaluate intra-rank parallelism we fix $N_{\rm cpu} = 4$ and vary `OMP_NUM_THREADS` from 1 to 30, using the same test problem as Section 11.3. 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 $e_{\rm cons} \leq 6.56 \times 10^{-4}$.

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 **??**) 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 4 compares per-component timers and speedup curves as a function of $N_{\rm omp}$. 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.

Table 6. OpenMP thread scaling with $N_{\rm cpu} = 4$ MPI ranks. The test problem has 54 M grids and 135.7 M particles with full physics. $N_{\rm omp}$ is the number of OpenMP threads per rank; total cores is $N_{\rm cpu} \times N_{\rm omp}$. 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.

$N_{\rm omp}$	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

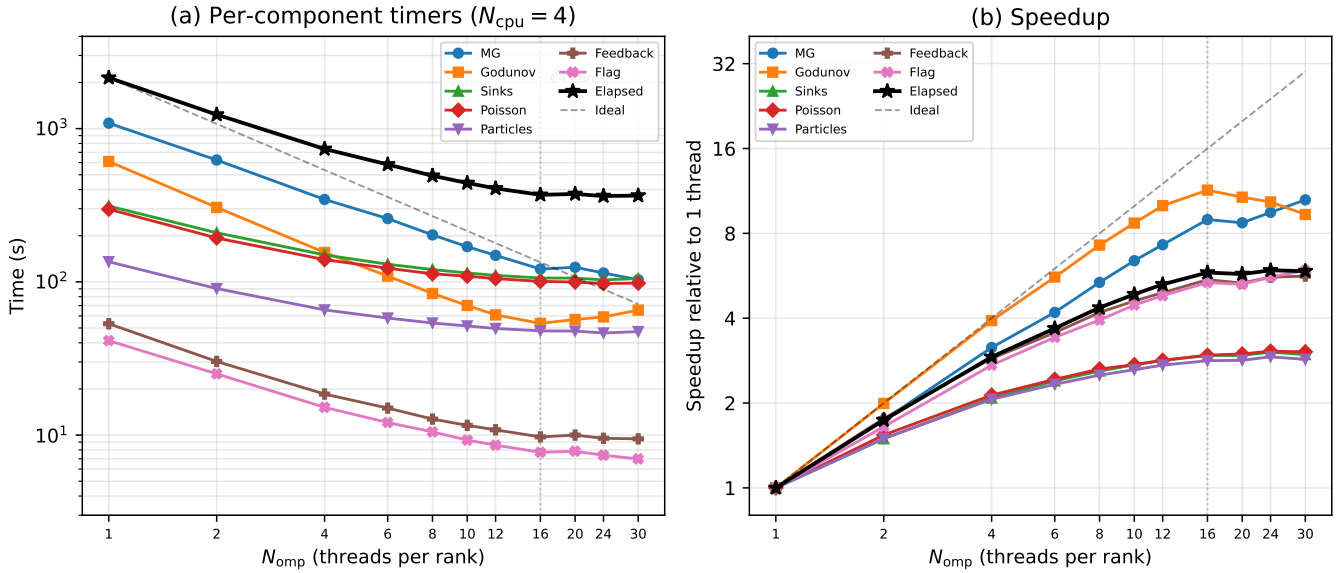

Figure 4. OpenMP thread scaling of CUDA-RAMSES with $N_{\rm cpu} = 4$ MPI ranks on a dual-socket AMD EPYC 7543 node (64 cores). (a) Per-component wall-clock times versus $N_{\rm omp}$; the dashed line shows ideal scaling from the single-thread baseline. The vertical dotted line marks the physical core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

11.5 Memory-Weighted Load Balancing

The memory-weighted cost function (equation 5) assigns $w_{\rm grid} = 270$ bytes per grid cell and $w_{\rm part} = 12$ 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: $M_{\rm max}/M_{\rm min} \leq 1.05$ for all rank counts tested, from 2 to 64. This is achieved despite substantial grid-count imbalance at the most populated refined 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 $N_{\rm gridmax}$; 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× 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 $N_{\rm cpu} \geq 8$ (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.

Table 7. Memory load balance for the strong-scaling test (54 M 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).

$N_{\rm cpu}$	$M_{\min}$ (Gb)	$M_{\max}$ (Gb)	$M_{\max}/M_{\min}$	Level-10 grids /rank min	Level-10 grids /rank max	max/min
1	147.2	147.2	1.000		—	
2	87.0	87.4	1.005	9 119 832	10 337 771	1.13
4	50.1	50.5	1.008	4 545 882	5 209 834	1.15
8	28.0	28.6	1.021	2 199 595	2 675 391	1.22
12	20.9	21.4	1.024	1 369 950	1 827 914	1.33
16	16.7	17.1	1.024	1 021 171	1 375 737	1.35
24	12.5	13.1	1.048	671 418	937 060	1.40
32	10.7	11.1	1.037	521 392	710 999	1.36
48	8.2	8.6	1.049	306 342	478 566	1.56
64	8.2	8.6	1.049	224 830	362 435	1.61

12 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 $N_{\rm gridmax} = 5$ M while simplifying the grid management code (no neighbour-pointer maintenance during creation, deletion, or migration).

(iii) **Memory optimizations.** On-demand allocation of the Hilbert key, histogram, and defragmentation arrays reduces steady-state memory by over 1 GB per rank, enabling larger problems or finer resolution within the same hardware budget.

(iv) **Multigrid solver optimizations.** Precomputed neighbour caches, merged red-black Gauss–Seidel sweeps (reducing communication by 44 per cent per iteration), fused residual-norm computation, and arithmetic optimizations reduce the Poisson solver's share of total runtime from 55 per cent to 39 per cent.

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

(vi) **Variable-ncpu restart.** Both HDF5 parallel I/O and distributed binary I/O enable output/restart with arbitrary rank counts, improving workflow flexibility for production simulations on shared facilities. The binary path uses round-robin file assignment and per-level `MPI_ALLTOALLV` with reusable exchange metadata.

(vii) **Hybrid CPU/GPU dispatch.** A dynamic dispatch model in which OMP threads acquire GPU stream slots at runtime, with fallback to CPU execution. Superbatch buffering amortises kernel launch latency, and an on-device scatter-reduce kernel reduces PCIe transfer volume by 20×. The Godunov solver achieves a 16 per cent speedup on an RTX 5000 Ada GPU while producing bit-identical results.

All modifications preserve physical consistency, as verified by conservation-law diagnostics ($e_{\rm cons}$, $e_{\rm pot}$, $e_{\rm kin}$) that are identical (or within MG tolerance) between the original and 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.

DATA AVAILABILITY

The modified code is available at https://github.com/kjhan0606/cuRAMSES-kjhan. Test configurations and analysis scripts will be shared upon reasonable request to the author.

REFERENCES

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

APPENDIX A: K-SECTION TREE WALK PSEUDOCODE

Algorithm 2 gives the pseudocode for mapping a spatial position 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

node ← root
$l \leftarrow 0$
while node is not a leaf **do**
  $l \leftarrow l + 1$
  $k \leftarrow k_l$; dir ← `ksec_dir`(l)
  child ← k
  for $j = 1$ **to** $k - 1$ **do**
    if $x_{\text{dir}} \leq$ `ksec_wall`(node, j) **then**
      child ← j; **break**
    end if
  end for
  node ← `ksec_next`(node, child)
end while
$c \leftarrow$ `ksec_indx`(node)

APPENDIX B: MORTON KEY ENCODING 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

$M \leftarrow 0$
for $b = 0$ **to** $B - 1$ **do**
  $M \leftarrow M \,|\, (\text{bit}_b(i_x) \ll 3b)$
  $M \leftarrow M \,|\, (\text{bit}_b(i_y) \ll (3b + 1))$
  $M \leftarrow M \,|\, (\text{bit}_b(i_z) \ll (3b + 2))$
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)

$(i_x, i_y, i_z) \leftarrow$ Decode(M)
Adjust i_d by ± 1 according to direction j
Apply periodic wrapping: $i_d \leftarrow i_d \bmod n_d$
$M' \leftarrow$ Encode(i_x, i_y, i_z)