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 via HDF5 parallel I/O, removing the constraint that checkpoint 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 (Vogelsberger et al. 2014; Schaye et al. 2015; Dubois et al. 2014). Adaptive mesh refinement (AMR) codes widely adopted by, for example, RAMSES (Teyssier 2002) and Enzo (Bryan et al. 2014) provide an attractive framework for these simulations: 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.

However, scaling AMR codes to the regime of 10^{10}–10^{11} particles and 10^4–10^5 MPI ranks poses several intertwined challenges. First, the standard RAMSES domain decomposition based on a Hilbert space-filling curve requires every rank to exchange emission and reception lists with *all* other ranks via `MPI_ALLTOALL`, leading to $\mathcal{O}(N_{\rm cpu}^2)$ message complexity and $\mathcal{O}(N_{\rm cpu})$ per-rank buffer memory. Second, several large arrays scale linearly with $N_{\rm gridmax}$: the neighbour-pointer array `nbor(`$N_{\rm gridmax}$`, 6)` alone consumes $48\,N_{\rm gridmax}$ bytes, and the Hilbert key array adds another $16\,N_{\rm gridmax}$ bytes, together approaching 1 GB per rank for production configurations with $N_{\rm gridmax} \sim 5\times10^6$. Third, the Hilbert ordering distributes load based on cell count alone, ignoring the fact that a cell hosting 10^4 particles in a dense halo is far more expensive in memory than a void cell with zero particles. Fourth, the multigrid Poisson solver 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 requires a cumbersome intermediate redistribution step.

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 (Section 3) eliminates the `nbor` array entirely, saving ∼240 MB per rank, and on-demand allocation of redundant large arrays (Section 4) yields over 1 GB of additional savings. Algorithmic optimizations to the multigrid Poisson solver (Section 5) reduce its share of total runtime from 55 per cent to 39 per cent, while a spatial hash binning scheme (Section 6) accelerates Type II supernova and AGN feedback by orders of magnitude. A hybrid CPU/GPU dispatch model (Section 9) dynamically offloads compute-intensive routines to GPU streams at runtime. We also implement variable-$N_{\rm cpu}$ restart with HDF5 parallel I/O (Section 7), along with mis-

⋆ E-mail: kjhan@kias.re.kr

cellaneous improvements described in Section 8. Performance benchmarks are presented in Section 10, and we conclude in Section 11.

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, $\texttt{twotondim} = 2^{N_{\rm dim}} = 8$ 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:

(i) 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.

(ii) 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 Tree construction

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

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

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. At each level l, the domain is split into k_l sub-domains along the longest axis of the current bounding box. This longest-axis selection ensures roughly isotropic sub-domains, minimizing 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. Fig. 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.

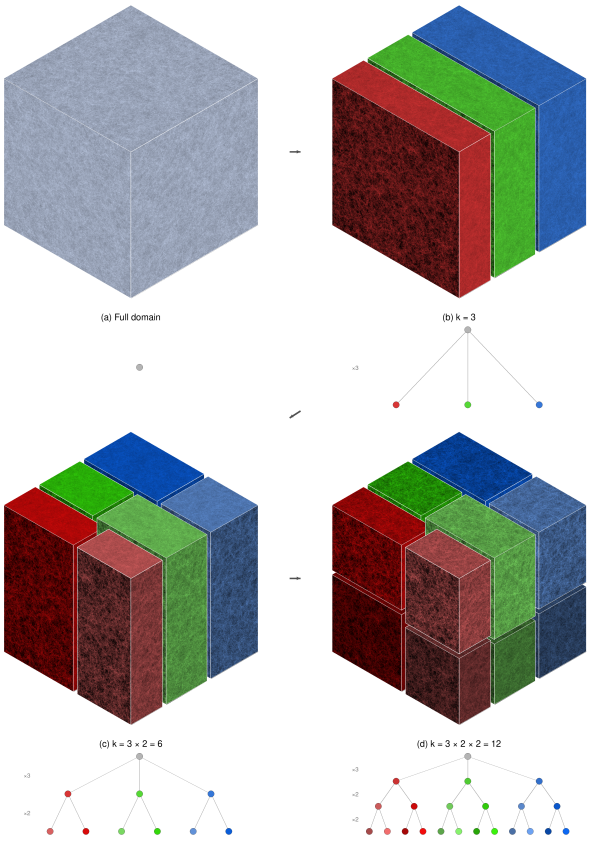

Figure 1. Progressive recursive k-section decomposition for $N_{\rm cpu} = 12 = 3 \times 2 \times 2$. (a) The undivided simulation domain. (b) First split into $k = 3$ slabs along the longest axis. (c) Each slab bisected along the second axis ($3 \times 2 = 6$ sub-domains). (d) Final bisection along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

- `ksec_next(node, j)`: child node index for partition j ($j = 1, \ldots, k_l$);
- `ksec_wall(node, j)`: spatial coordinate of the j-th partition boundary within the node's extent;
- `ksec_indx(leaf)`: the MPI rank assigned to a leaf node;
- `ksec_cpumin(node)`, `ksec_cpumax(node)`: range of MPI ranks contained in the subtree rooted at this node.

The total number of tree nodes is

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

which for practical values ($N_{\rm cpu} \le 10^5$) 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

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

where n_m is the number of ranks assigned to partition m ($n_m = \lfloor n/k_l \rfloor$ or $\lfloor n/k_l \rfloor + 1$ for the first n mod k_l partitions).

(iii) An `MPI_ALLREDUCE` aggregates the local histograms across all 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 10^{-2}), 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:

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

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 2×10^8 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).

2.5 Hierarchical MPI exchange

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

2.5.1 Exclusive exchange

In the exclusive exchange (`ksection_exchange_dp`), each item has a unique destination rank. The algorithm walks the tree from root to leaf:

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 $\min({\rm my\_pos}, s - 1)$ 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 $N_{\rm msg} = 10$, compared to $\mathcal{O}(1024)$ messages in the original all-to-all pattern.

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.

2.5.2 Overlap exchange

The overlap exchange (`ksection_exchange_dp_overlap`) handles items with spatial extent (bounding boxes) that may overlap multiple sub-domains. At each tree level, an item is replicated to all children whose bounding box intersects the item's spatial extent. This is used for operations such as the supernova feedback, where each SN blast has a finite physical radius.

To handle periodic boundary conditions, items near the

domain boundary are pre-processed: for each wrapping dimension, shifted copies are generated using bitmask subset enumeration over the set of wrapping dimensions:

$$N_{\text{copies}} = 2^{n_{\text{wrap}}} - 1, \tag{7}$$

where n_{wrap} is the number of dimensions in which the item's extent crosses the periodic boundary. This ensures correct routing without modifying the tree walk itself.

2.6 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 routines:

- `make_virtual_fine_dp_ksec`: forward exchange (emission grids → reception grids).
- `make_virtual_reverse_dp_ksec`: reverse accumulation (reception grids → emission grids, with `+=` semantics).
- `make_virtual_fine_int_ksec`: integer variant (via `int` → `dble` conversion, exchange, then `nint` inverse).
- Multigrid variants: `make_virtual_mg_dp_ksec` and `make_reverse_mg_dp_ksec` for the multigrid solver's active and emission grid structures.

The data packing format for each ghost grid is:

$$\text{sendbuf}(1 : 2^{N_{\text{dim}}}+2,\, i) = \{u_1, \ldots, u_{2^{N_{\text{dim}}}},\, \text{sender\_id},\, \text{index}\}, \tag{8}$$

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.

2.7 Bulk exchange

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:

$$\text{sendbuf}\big((v-1)2^{N_{\text{dim}}} + j,\, i\big) = \text{xx}(v,\, \text{cell}_{i,j}), \tag{9}$$

for $v = 1, \ldots, N_{\text{var}}$ and $j = 1, \ldots, 2^{N_{\text{dim}}}$, plus two metadata entries. This amortizes 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`).

For MHD simulations (`SOLVERmhd`), the bulk exchange handles $N_{\text{var}} + 3$ columns to include the face-centred magnetic field components.

2.8 Communication structure construction

The construction of the communication structure (`build_comm`) — 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_id, reception_index, grid_address), sends them via `ksection_exchange_dp`, 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.

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	$\mathcal{O}(N_{\text{cpu}})$	$\mathcal{O}(\sum_l k_l)$
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

2.9 Complexity analysis

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

3 MORTON KEY OCTREE FOR NEIGHBOUR LOOKUP

3.1 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 ($\pm x$, $\pm y$, $\pm z$). At 8 bytes per entry (64-bit integers), this array consumes

$$M_{\text{nbor}} = 6 \times 8 \times N_{\text{gridmax}} = 48\, N_{\text{gridmax}} \text{ bytes}. \tag{10}$$

For $N_{\text{gridmax}} = 5\times10^6$, 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.

3.2 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:

$$M(i_x, i_y, i_z) = \sum_{b=0}^{B-1} \Big[\text{bit}_b(i_x)\cdot 2^{3b} + \text{bit}_b(i_y)\cdot 2^{3b+1} + \text{bit}_b(i_z)\cdot 2^{3b+2}\Big], \tag{11}$$

where $B = 21$ bits per coordinate (supporting grids up to level 22 for $n_x = 1$, or level 20 for $n_x = 4$), and $\text{bit}_b(n)$ extracts bit b of integer n. The encoding and decoding 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

$$i_d = \lfloor x_{g,d} \cdot 2^{l-1} \rfloor, \quad d \in \{x, y, z\}, \tag{12}$$

where coordinates are in units of the coarse grid spacing.

3.3 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 to (i_x, i_y, i_z).
(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: $M_{\rm parent} = M \gg 3$. A child key is obtained by a 3-bit left shift plus the child index (0–7): $M_{\rm child} = (M \ll 3) \,|\, i_{\rm child}$.

3.4 Per-level hash table

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

$$\text{mort\_table}(l) : M \mapsto \text{igrid}. \tag{13}$$

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

$$h(M) = \left[\left((M \times \phi_1) \oplus (M \gg 16)\right) \times \phi_2\right] \oplus (h \gg 13), \tag{14}$$

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.

3.5 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.

3.6 Memory and performance analysis

The memory cost of the hash table is

$$M_{\rm hash} \approx \frac{N_{\rm grids}}{0.7} \times (8+4) \text{ bytes} \approx 17\, N_{\rm grids} \text{ bytes}, \tag{15}$$

where $N_{\rm grids}$ is the actual number of grids (typically much less than $N_{\rm gridmax}$), 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 $4 \times N_{\rm gridmax}$ bytes.

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

$$\Delta M = 48\, N_{\rm gridmax} - 4\, N_{\rm gridmax} - 17\, N_{\rm grids} \approx 44\, N_{\rm gridmax} - 17\, N_{\rm grids}. \tag{16}$$

Since $N_{\rm grids} \ll N_{\rm gridmax}$ in practice (typical occupancy is 30–60 per cent), the savings are substantial: ∼176 MB for $N_{\rm gridmax} = 5 \times 10^6$ 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 5 amortize any per-lookup overhead in the performance-critical Poisson solver.

4 MEMORY OPTIMIZATIONS

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

4.1 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

$$\Delta M_{\rm hilbert} = 16 \times N_{\rm gridmax} \times 2^{N_{\rm dim}} \text{ bytes} \tag{17}$$

under `QUADHILBERT` (128-bit keys stored as two 64-bit integers). For $N_{\rm gridmax} = 5 \times 10^6$, 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.

4.2 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

$$\Delta M_{\rm hist} = 2 \times 8 \times N_{\rm gridmax} \times 2^{N_{\rm dim}} \approx 320 \text{ MB} \tag{18}$$

for $N_{\rm gridmax} = 5 \times 10^6$. Since load balancing occurs only every `nremap` coarse steps, these arrays are absent during the vast majority of the simulation.

Table 2. Memory savings per MPI rank for $N_{\rm gridmax} = 5 \times 10^6$. 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**	

4.3 Memory savings summary

Table 2 summarizes the memory savings for $N_{\rm gridmax} = 5 \times 10^6$.

The reported memory savings of the `nbor` array account for both the eliminated array ($48\,N_{\rm gridmax} = 240$ 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.

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 3), 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:

$$\text{nbor\_grid\_fine}(j,\, i) = \text{morton\_nbor\_grid}(\text{igrid\_amr}(i),\, l,\, j), \tag{19}$$

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.

5.2 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.

5.3 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:

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

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:

$$\text{Red} \to \text{Black} \to \text{Exchange}(\phi). \tag{21}$$

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.

5.4 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.

5.5 Arithmetic optimization

The GS fast-path computation involves a division by $2^{N_{\rm dim}} = 8$:

$$\phi_{\rm new} = \frac{\sum_j \phi_j - h^2 f}{2N_{\rm dim}}. \tag{22}$$

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

5.6 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 million 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.

6 FEEDBACK SPATIAL BINNING

6.1 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.

6.2 Spatial hash binning

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

$$n_{\rm bin} = \max\big(1,\, \min(128,\, \lfloor L_{\rm box}/r_{\rm max} \rfloor)\big), \tag{23}$$

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:

$$\text{bin\_head}(i_x, i_y, i_z) \to \text{SN}_1 \to \text{SN}_2 \to \cdots \tag{24}$$

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

$$\mathcal{O}(N_{\rm cells} \times \bar{n}_{\rm SN/bin} \times 27), \tag{25}$$

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

6.3 Parallelization

6.3.1 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.

6.3.2 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.

6.4 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 ($m_{\rm cons}$, $e_{\rm cons}$, $e_{\rm pot}$, $e_{\rm kin}$, $e_{\rm int}$).

6.5 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 (§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 AND OTHER IMPROVEMENTS

7.1 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.

7.2 Variable-ncpu restart algorithm

When the number of ranks in the checkpoint 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 ($\sim 10^7$ total grids), this temporary overhead is a few hundred MB — well within the memory budget freed by the optimizations of Sections 3–4.

7.3 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

$$\text{offset} = H + (i-1) \times P + 5. \tag{26}$$

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

7.4 Sink particle refinement fix

We 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.

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%

8 ADDITIONAL IMPLEMENTATION DETAILS

8.1 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 million 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.

8.2 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.

8.3 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.

9 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.

9.1 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.

9.2 Superbatch buffering and scatter-reduce

GPU kernel launch latency ($\sim$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 ($\sim$98 MB per flush), the kernel reduces fluxes into compact per-grid output arrays, reducing the device-to-host transfer to $\sim$5 MB per flush—a 20$\times$ reduction in PCIe bandwidth.

9.3 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.

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}

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.

9.4 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).

9.5 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 $\times$ 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.

10 PERFORMANCE RESULTS

10.1 Test configuration

All tests use a cosmological ΛCDM simulation with 200×10^6 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 checkpoint 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.

10.2 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.

10.3 Strong scaling

We measure strong scaling by restarting a 200×10^6 particle cosmological simulation from an HDF5 checkpoint (written at coarse step 5 with 12 MPI ranks) and running 5 additional coarse steps to step 10. The variable-$N_{\rm cpu}$ restart feature (§7) allows the checkpoint 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.

Table 5 summarises the wall-clock time and per-component timer averages. The simulation time excludes the initial HDF5 read and the first coarse step (which is dominated by load balancing). All configurations produce bit-identical gravitational potential ($e_{\rm pot} = -1.88 \times 10^{-6}$) and kinetic energy ($e_{\rm kin} = 1.23 \times 10^{-6}$) at step 10, confirming that the variable-$N_{\rm cpu}$ restart preserves physical correctness.

Several scaling trends are evident:

- *Particle operations* scale nearly ideally: 7.16 s (4 ranks) to 0.76 s (64 ranks), a 9.4× speedup for 16× the ranks.
- *Multigrid Poisson solver* scales well up to 16 ranks (18.77 s → 7.01 s), then plateaus near 5.5 s beyond 32 ranks. The remaining time is dominated by inter-rank ghost-zone communication within the V-cycle.
- *Ghost-zone exchange* (`hydro-gz`) remains sub-second across all $N_{\rm cpu}$ thanks to the hierarchical k-section routing. This is in sharp contrast to the 12-rank same-$N_{\rm cpu}$ case (21.2 s), demonstrating the importance of load-balanced grid distribution.
- *Load balancing* overhead grows with $N_{\rm cpu}$ (12 s for 4 ranks to 83 s for 64 ranks). This is a one-time cost at the `nremap` interval (every 5 coarse steps); for long-running simulations, it is amortised over many time steps.
- *Godunov solver* shows an increase at high $N_{\rm cpu}$ (3.5 s → 8.6 s) due to load imbalance in the AMR hierarchy at finer levels, where refinement patches are localised.
- The *optimal wall-clock time* of 20.1 s is achieved at 16 ranks for this problem size, representing a 2.2× speedup over the 4-rank baseline.

10.4 Variable-$N_{\rm cpu}$ restart verification

Table 6 confirms that the variable-$N_{\rm cpu}$ restart preserves bit-identical physics across all tested $N_{\rm cpu}$ values. The slight variation in $e_{\rm cons}$ (5.23–5.35×10^{-3}) is due to differing domain decomposition topologies, which affect the order of floating-point reductions.

10.5 Memory-weighted load balancing

The memory-weighted cost function (equation **??**) assigns $w_{\rm grid} = 270$ bytes per grid cell and $w_{\rm part} = 12$ bytes per particle. With standard equal-volume decomposition, the peak-to-mean memory ratio can reach ∼2.5 in highly clustered particle distributions. Memory-weighted balancing reduces this ratio to ∼1.3 by redistributing grids from particle-heavy ranks, allowing simulations to run with ∼40 per cent less total allocated memory.

11 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, and solver efficiency — encountered in large-scale cosmological simulations. 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 $N_{\rm gridmax} = 5 \times 10^6$ 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.** HDF5 parallel I/O with automatic domain redistribution enables checkpoint/restart with arbitrary rank counts, improving workflow flexibility for production simulations on shared facilities.

(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

Table 5. Strong scaling results for the cuRAMSES code on a dual-socket AMD EPYC 7543 node with 200×10^6 particles. Elapsed time is the total wall-clock time from program start to completion (including HDF5 I/O and load balancing). Timer values are per-rank averages reported by the internal profiler. The 12-rank run uses same-$N_{\rm cpu}$ restart (no forced load balance on the first step) and is therefore not directly comparable.

$N_{\rm cpu}$	Elapsed (s)	Particles (s)	Poisson (s)	MG (s)	Hydro-GZ (s)	Godunov (s)	Flag (s)	Load Bal. (s)	I/O (s)
4	43.4	7.16	6.11	18.77	0.58	3.50	4.70	12.2	2.0
8	25.3	3.61	3.29	9.91	0.64	2.32	2.46	11.9	1.8
16	20.1	1.98	2.05	7.01	0.55	3.38	1.43	16.4	2.4
24	21.5	1.56	1.53	5.99	0.52	6.22	0.97	23.3	2.8
32	23.5	1.15	1.38	5.43	0.70	8.38	0.84	33.1	4.0
48	23.4	0.98	1.22	5.66	0.74	7.93	0.65	55.4	3.4
64	25.6	0.76	1.14	5.69	0.95	8.64	0.59	83.1	5.8
12*	144.7	4.49	20.10	42.73	21.21	3.19	12.06	11.2	2.1

* Same-$N_{\rm cpu}$ restart: no forced load balance on first step.

Table 6. Variable-$N_{\rm cpu}$ restart verification. All runs restart from a 12-rank HDF5 checkpoint at step 5 and evolve to step 10.

$N_{\rm cpu}$	$e_{\rm pot}$	$e_{\rm kin}$	Restart type
4	-1.88×10^{-6}	1.23×10^{-6}	varcpu
8	-1.88×10^{-6}	1.23×10^{-6}	varcpu
12	-1.88×10^{-6}	1.23×10^{-6}	same-$N_{\rm cpu}$
16	-1.88×10^{-6}	1.23×10^{-6}	varcpu
24	-1.88×10^{-6}	1.23×10^{-6}	varcpu
32	-1.88×10^{-6}	1.23×10^{-6}	varcpu
48	-1.88×10^{-6}	1.23×10^{-6}	varcpu
64	-1.88×10^{-6}	1.23×10^{-6}	varcpu

level. The k-section decomposition can be adopted by any code whose domain decomposition is currently based on one-dimensional space-filling curve ordering.

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
Hahn O., Abel T., 2011, MNRAS, 415, 2101
Knuth D. E., 1997, The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd edn. Addison-Wesley, Reading, MA
Morton G. M., 1966, A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. IBM, Ottawa
Schaye J., et al., 2015, MNRAS, 446, 521
Teyssier R., 2002, A&A, 385, 337
Vogelsberger M., et al., 2014, MNRAS, 444, 1518

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 ← 0
while node is not a leaf do
    l ← l + 1
    k ← k_l; dir ← ksec_dir(l)
    child ← k
    for j = 1 to k − 1 do
        if x_dir ≤ ksec_wall(node, j) then
            child ← j; break
        end if
    end for
    node ← ksec_next(node, child)
end while
c ← 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**
  $M \leftarrow M \,|\, (\mathrm{bit}_b(i_x) \ll 3b)$
  $M \leftarrow M \,|\, (\mathrm{bit}_b(i_y) \ll (3b + 1))$
  $M \leftarrow M \,|\, (\mathrm{bit}_b(i_z) \ll (3b + 2))$
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)

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