cuRAMSES-kjhan

Technical Reference Manual

K-Section Ordering, Morton Key Octree,
Memory-Based Load Balancing, and
Performance Optimizations

Juhan Kim

February 2026

Based on RAMSES (R. Teyssier)

`git@github.com:kjhan0606/cuRAMSES-kjhan.git`

Contents

Code Modifications

1 Quick Start Guide

This chapter provides a practical step-by-step guide to building cuRAMSES and running a cosmological zoom-in simulation. For detailed explanations of the underlying algorithms and parameters, refer to the subsequent chapters.

1.1 Prerequisites

- **Intel Fortran Compiler** (`ifx` or `ifort`) with MPI wrapper (`mpiifx`)
- **OpenMP** support (enabled via `-qopenmp`)
- **MUSIC** — Multi-Scale Initial Conditions generator (`https://bitbucket.org/ohahn/music`)
- **Python 3** with `numpy` (for utility scripts such as `create_pvar006.py`)

1.2 Building cuRAMSES

Build Commands

```
cd bin
make clean
make
```

The binary is produced as `bin/ramses_final3d`. Key compile-time options (`-DNVAR=11`, `-DNPRE=8`, `-DLONGINT`, etc.) are set in the Makefile. See Chapter 29 for details on compile-time flags.

1.3 Setting Up a Zoom-In Simulation

The following steps walk through a complete zoom-in cosmological simulation with star formation, targeting $\sim$1 kpc resolution in a $10\,h^{-1}$ Mpc box.

1.3.1 Step 1: Generate Initial Conditions with MUSIC

Create a MUSIC configuration file. The key settings for a zoom-in are `levelmin` (base grid), `levelmax` (finest zoom level), and `ref_extent` (zoom region fraction).

zoomin_10Mpc.conf (excerpt)

```
[setup]
boxlength       = 10            # Box size in Mpc/h
zstart          = 50            # Starting redshift
levelmin        = 7             # Base grid: 128^3
```

```
levelmax        = 11           # Zoom finest: 2048^3 effective
ref_center      = 0.5, 0.5, 0.5
ref_extent      = 0.04, 0.04, 0.04
baryons         = yes
use_2LPT        = yes

[cosmology]
Omega_m         = 0.3111
Omega_L         = 0.6889
Omega_b         = 0.04
H0              = 67.66
sigma_8         = 0.8102
nspec           = 0.9665
transfer        = eisenstein

[output]
format          = grafic2
filename        = IC_zoomin
```

Run MUSIC:

Generate ICs

```
./MUSIC zoomin_10Mpc.conf
```

This produces `IC_zoomin/level_007` through `IC_zoomin/level_011`, each containing GRAFIC2 binary files (`ic_deltab`, `ic_velcx/y/z`, `ic_poscx/y/z`, etc.).

1.3.2 Step 2: Create Zoom Geometry Scalar

The zoom geometry scalar (`ic_pvar_00006`) is a passive variable that marks which cells belong to the zoom region. This is essential when using `ivar_refine=11` to control AMR refinement during initialization. See Chapter 10 for a detailed explanation.

Create pvar006

```
cd test_ksection
python3 create_pvar006.py
```

Edit the script's `IC_DIR` and `LEVELS` variables to match your IC directory.

1.3.3 Step 3: Create Run Directory

Setup Run Directory

```
mkdir -p run_zoomin
cd run_zoomin
ln -s ../IC_zoomin .
cp ../../bin/ramses_final3d .
cp ../cosmo_zoomin_physics.nml namelist.nml
```

1.3.4 Step 4: Configure Namelist

The most important parameters to set correctly in the namelist:

Key namelist parameters

```
&RUN_PARAMS
ordering='ksection'      ! hierarchical domain decomposition
memory_balance=.true.    ! memory-weighted load balancing
nremap=5                 ! load balance every 5 coarse steps
/

&AMR_PARAMS
levelmin=7               ! must match MUSIC levelmin
levelmax=20              ! maximum AMR level allowed
ngridtot=200000000       ! total grids (see memory sizing)
nparttot=200000000       ! total particles
/

&REFINE_PARAMS
m_refine=13*8.           ! Lagrangian refinement threshold
ivar_refine=11           ! use passive scalar for zoom mask
var_cut_refine=0.01      ! threshold for zoom scalar
mass_cut_refine=-1       ! set per IC level (see table below)
/

&INIT_PARAMS
filetype='grafic'
initfile(1)='IC_zoomin/level_007'
initfile(2)='IC_zoomin/level_008'
initfile(3)='IC_zoomin/level_009'
initfile(4)='IC_zoomin/level_010'
initfile(5)='IC_zoomin/level_011'
/
```

1.3.5 Step 5: Run

Launch Simulation

```
mpirun -np 12 ./ramses_final3d namelist.nml 2>&1 | tee run.log
```

1.4 Key Parameters Quick Reference

Parameter	Recommended	Purpose
`ordering`	`'ksection'`	Hierarchical domain decomposition
`memory_balance`	`.true.`	Memory-weighted load balancing
`nremap`	5	Load balance frequency (optimal)
`ivar_refine`	11 (=NVAR)	Passive scalar index for zoom mask
`var_cut_refine`	0.01	Threshold for zoom scalar refinement

Parameter	Recommended	Purpose
`mass_cut_refine`	see table	Particle mass filter for `cpu_map2`
`ngridtot`	see sizing	Total grids; keep virtual mem < CommitLimit
`nparttot`	see sizing	Total particles across all ranks
`levelmin`	IC levelmin	Must match MUSIC base level
`levelmax`	14–20	Maximum AMR refinement

1.4.1 mass_cut_refine Reference

The `mass_cut_refine` parameter filters out heavy (background) dark matter particles from the `cpu_map2` density computation in `rho_fine`. Set it to a value between the zoom-region particle mass and the coarser-level particle mass. Recommended values by IC finest level:

IC finest level	Zoom m_{DM}	Coarse m_{DM}	mass_cut_refine
9	$\sim 4.6 \times 10^7$	$\sim 3.7 \times 10^8$	1.0×10^8
10	$\sim 5.8 \times 10^6$	$\sim 4.6 \times 10^7$	1.5×10^7
11	$\sim 7.2 \times 10^5$	$\sim 5.8 \times 10^6$	2.0×10^6
12	$\sim 9.0 \times 10^4$	$\sim 7.2 \times 10^5$	2.5×10^5
13	$\sim 1.1 \times 10^4$	$\sim 9.0 \times 10^4$	3.0×10^4

Units

Masses are in $M_\odot/h$ for a $10\,h^{-1}$ Mpc box with Planck 2018 cosmology ($\Omega_m = 0.3111$, $\Omega_b = 0.04$, $h = 0.6766$). The exact values depend on the box size and cosmological parameters.

1.5 Troubleshooting

Error / Symptom	Solution
`No more free memory`	Increase `ngridtot` (or `ngridmax`) in `&AMR_PARAMS`.
`No more free memory for particles`	Increase `nparttot` (or `npartmax`) in `&AMR_PARAMS`.
Process killed (SIGNAL 9, OOM)	`ngridtot` is too large for available RAM. RAMSES allocates the full array at startup (virtual memory). Check `CommitLimit` in `/proc/meminfo` and reduce `ngridtot` so that total virtual memory stays below it.
Background AMR explosion (entire box refined, memory exhausted)	Verify `ivar_refine=11` and that `ic_pvar_00006` files exist in each IC level directory. Without the zoom geometry scalar, `cpu_map2` marks the entire domain for refinement.

Error / Symptom	Solution
Refinement does not extend beyond IC levels	Check that `m_refine` has enough entries (one per extra level above IC finest) and that `mass_cut_refine` is set correctly to exclude heavy background particles.
Very slow Poisson solver	Increase `cg_levelmin` in `&POISSON_PARAMS` to switch from multigrid to conjugate-gradient at the finest levels. Typical: `cg_levelmin=14`.

2 K-Section Ordering

2.1 Overview

The **k-section ordering** replaces the standard Hilbert curve domain decomposition with a hierarchical spatial bisection tree. Unlike the Hilbert curve approach that relies on a 1D space-filling curve, the k-section ordering partitions the 3D domain using recursive bisection along each coordinate axis, producing a balanced k-ary tree of spatial subdomains.

Key Advantage

The k-section tree enables **hierarchical MPI exchange** where communication follows the tree structure, achieving $O(\sum k_l)$ messages per exchange instead of $O(N_{\text{cpu}})$ all-to-all communication.

Figure 2.2 illustrates the progressive nature of the recursive k-section tree construction. Starting from the undivided simulation domain (a), the tree is built by recursively k-secting along successive coordinate axes. In this example with $k_1 \times k_2 \times k_3 = 3 \times 2 \times 2 = 12$ CPUs, the decomposition proceeds in three stages:

1. **Level 1** ($k_1 = 3$): The domain is split into 3 slabs along the x-axis.
2. **Level 2** ($k_2 = 2$): Each slab is bisected along the y-axis, producing 6 subdomains.
3. **Level 3** ($k_3 = 2$): Each subdomain is bisected along the z-axis, yielding the final 12 subdomains.

The split positions at each level are determined by the load-balancing histogram to ensure equal computational cost per subdomain, resulting in non-uniform subdomain volumes that reflect the local workload. The colour hue in Figure 2.2 encodes the x-slab membership (red/green/blue), with saturation and brightness variations indicating the y and z subdivisions. Below each panel, the corresponding recursive k-section tree is shown, with leaf nodes coloured to match their domain.

2.2 Hierarchical MPI Exchange

Two exchange routines are provided in `patch/cuda/ksection.f90`:

- `ksection_exchange_dp` — exclusive exchange (each item → exactly 1 destination CPU)
- `ksection_exchange_dp_overlap` — overlap exchange (items routed by spatial bounding box)

2.2.1 Exclusive Exchange

Each data item has a known destination CPU. The algorithm performs level-by-level correspondent exchange through the k-section tree, using MPI tags 100–300+level.

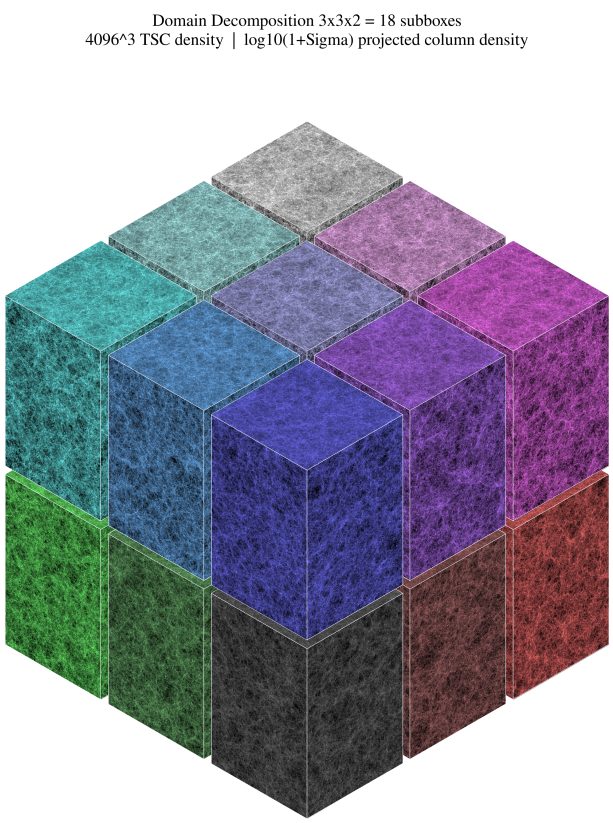

Figure 2.1: Isometric view of k-section domain decomposition with a $3 \times 3 \times 2 = 18$ uniform partition. Each subbox surface shows the projected column density $\log_{10}(1+\Sigma)$ from a 4096^3 TSC density field. Colors encode spatial position: R$\rightarrow x$, G$\rightarrow y$, B$\rightarrow z$, providing a smooth gradient across adjacent subboxes.

2.2.2 Overlap Exchange

Items have spatial extent (bounding box) and may need to reach multiple CPUs. The tree walk determines all destination CPUs whose spatial domain overlaps the item's bounding box. MPI tags 400–500+level are used.

2.2.3 Periodic Boundary Conditions

The optional `periodic=.true.` parameter enables handling of items that wrap around the domain boundary $[0, \text{scale}]$. For each wrapping dimension, $2^{n_{\text{wrap}}} - 1$ shifted copies are generated via bitmask subset enumeration before the tree walk.

2.3 Tree Navigation

The arrays `ksec_cpumin/cpumax` and `ksec_cpu_path` (set in `build_ksection` and at restart via `rebuild_ksec_cpuranges`) enable each CPU to navigate the tree efficiently.

2.4 Verification

The exchange routines are tested in `test_ksection_exchange` with 4 test cases:

1. Exclusive point exchange
2. Overlap point exchange
3. Full overlap exchange
4. Periodic overlap exchange (20% radius items)

All tests pass.

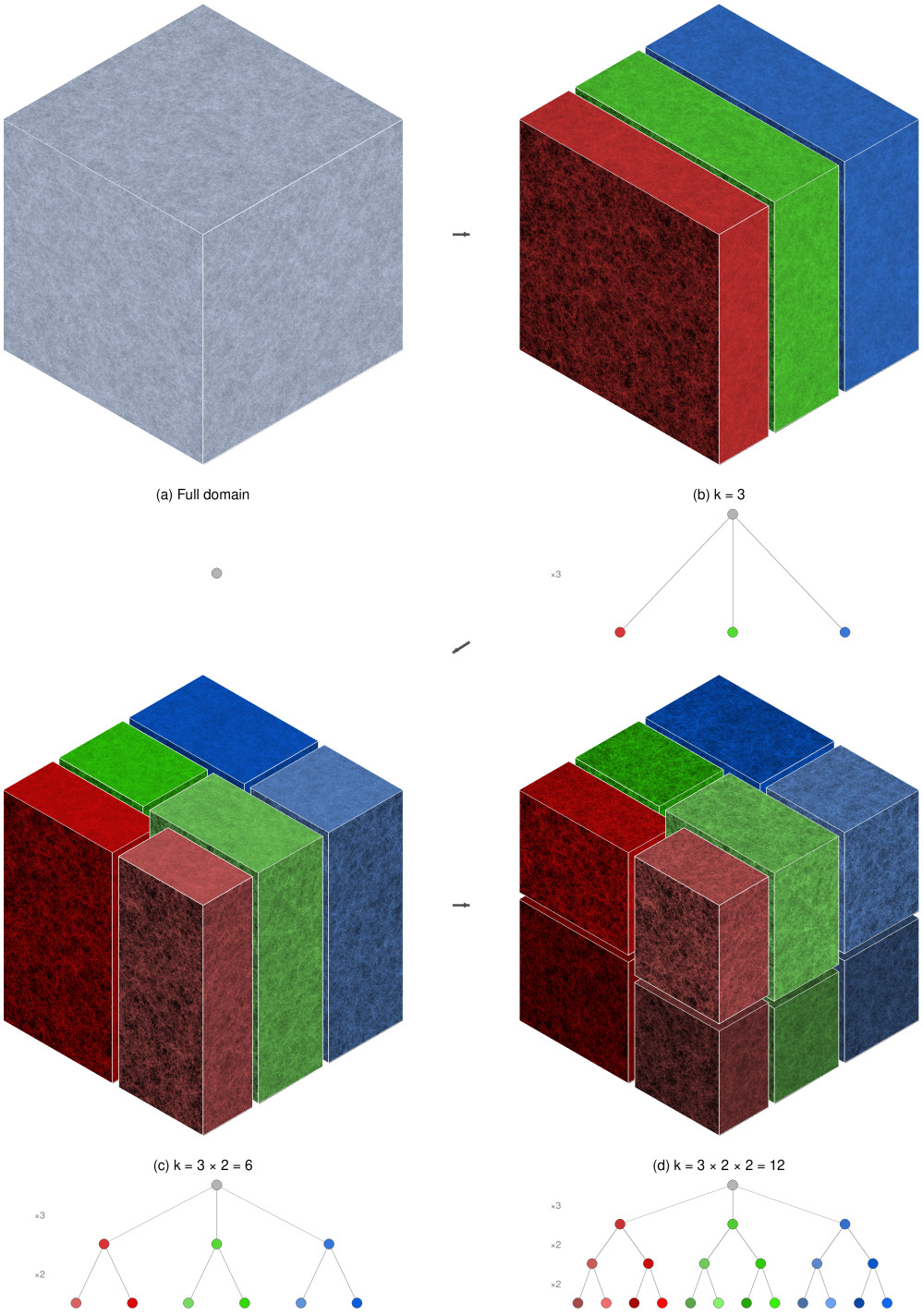

Figure 2.2: Progressive recursive k-section decomposition for $k_1 \times k_2 \times k_3 = 3 \times 2 \times 2 = 12$ CPUs. (a) Undivided simulation domain. (b) First split into 3 slabs along x. (c) Each slab bisected along y ($3 \times 2 = 6$ subdomains). (d) Final bisection along z ($3 \times 2 \times 2 = 12$ subdomains). Below each panel, the corresponding k-section tree shows the hierarchical structure; leaf node colours match domain colours. Subdomain volumes vary by $\sim$10–30%, reflecting load-balanced wall placement. Each face shows projected column density $\log_{10}(1+\Sigma)$ from a 4096^3 TSC density field.

3 Ghost Zone Exchange via K-Section

3.1 AMR Ghost Zones

The standard RAMSES ghost zone exchange uses `MPI_ISEND/IRECV` with all-to-all communication patterns. This was replaced with k-section tree-routed exchange for the `ordering='ksection'` mode.

3.1.1 Modified File

`patch/cuda/virtual_boundaries.kjhan.f90`

3.1.2 Subroutines

Four k-section variants were implemented:

Subroutine	Direction	Data Type
`make_virtual_fine_dp_ksec`	Forward	`real(dp)`
`make_virtual_fine_int_ksec`	Forward	`integer`
`make_virtual_reverse_dp_ksec`	Reverse (+= accumulate)	`real(dp)`
`make_virtual_reverse_int_ksec`	Reverse (+= accumulate)	`integer`

3.1.3 Data Packing

Each emission grid is packed as:

$$\texttt{sendbuf}(1{:}\text{twotondim}+2,\ i) = \big[\underbrace{c_1, c_2, \ldots, c_8}_{\text{cell data}},\ \underbrace{\text{myid}}_{\text{sender}},\ \underbrace{i}_{\text{index}}\big]$$

The metadata (sender ID, emission index) allows the receiver to scatter data to the correct reception grid without requiring a priori knowledge of the communication pattern.

3.1.4 Dispatch

Dispatch is automatic via:

```
if(ordering=='ksection') then
   call make_virtual_fine_dp_ksec(xx, ilevel)
   return
end if
```

3.1.5 Bulk Exchange

Four bulk variants exchange all columns of a 2D array (e.g., `uold`, `f`, `unew`) in a single `ksection_exchange_dp` call:

Subroutine	Description
`make_virtual_fine_dp_bulk`	Forward bulk exchange
`make_virtual_fine_dp_bulk_ksec`	Forward bulk (ksection impl.)
`make_virtual_reverse_dp_bulk`	Reverse bulk exchange
`make_virtual_reverse_dp_bulk_ksec`	Reverse bulk (ksection impl.)

Buffer layout for `ncols` variables:

$$\texttt{sendbuf}\big((v-1)\cdot 2^3 + j,\ \text{idx}\big) = \texttt{xx}(\text{icell},\ v) \quad v = 1\ldots\text{ncols},\ j = 1\ldots 8$$

plus 2 metadata words (sender ID, index). This reduces MPI exchanges from N_{var} per level to 1 per array.

3.2 build_comm via K-Section

The `build_comm` subroutine's `MPI_ALLTOALL` + `MPI_ISEND/IRECV` pattern was also replaced with `ksection_exchange_dp`.

3.3 MPI_ALLTOALL Replacements

Additional `MPI_ALLTOALL` calls in the following files were replaced with k-section exchange:

- `particle_tree.kjhan.f90`
- `init_part.f90`
- `multigrid_fine_commons.f90` (in `build_parent_comms_mg`)

3.4 Pre-Allocated Buffer Pool

Per-level small arrays (child count, peer list, MPI requests) were converted to `save` variables to eliminate $\sim$100 allocations/deallocations per call. The `peer_recv` buffer uses grow-only capacity, and the first dimension must exactly match `nprops` for MPI stride correctness.

4 Multigrid Poisson K-Section Communication

4.1 Motivation

The multigrid Poisson solver (`poisson-mg`) accounts for 29–41% of total runtime. The original MPI communication used `MPI_ISEND/IRECV` with all-to-all patterns across all CPUs.

4.2 Implementation

Four k-section variants were added to `patch/cuda/multigrid_fine_commons.f90`:

Subroutine	Direction	Data Type
`make_virtual_mg_dp_ksec`	Forward	`real(dp)`
`make_virtual_mg_int_ksec`	Forward	`integer`
`make_reverse_mg_dp_ksec`	Reverse (+= accumulate)	`real(dp)`
`make_reverse_mg_int_ksec`	Reverse (+= accumulate)	`integer`

4.2.1 Key Differences from AMR Exchange

The MG communication uses different data structures from the standard AMR exchange:

Aspect	AMR	Multigrid
Active grids	`active(ilevel)`	`active_mg(myid,ilevel)`
Reception	`reception(icpu,ilevel)`	`active_mg(icpu,ilevel)`
Emission	`emission(icpu,ilevel)`	`emission_mg(icpu,ilevel)`
Data arrays	`xx(igrid+iskip)`	`active_mg%u(icell,ivar)`
Indexing	Global grid index	Local offset in active_mg

4.2.2 Forward Exchange

1. Pack: for each `emission_mg(icpu)%igrid(i)`, gather `twotondim` values from `active_mg(myid)%u` + metadata (sender_id, index)
2. Exchange via `ksection_exchange_dp`
3. Scatter: use metadata to write into `active_mg(sender)%u(ridx + step, ivar)`

4.2.3 Reverse Exchange (Accumulation)

1. Pack: for each remote `active_mg(icpu)%u(i + step, ivar)` + metadata
2. Exchange via `ksection_exchange_dp`
3. Accumulate: `active_mg(myid)%u(emission_mg(sender)%igrid(ridx) + step, ivar) += recvbuf`

5 Morton Key Octree

5.1 Overview

The `nbor` array (6 neighbor pointers per grid) was replaced with a Morton key hash table for $O(1)$ neighbor lookup. This saves $6 \times 8 \times N_{\text{gridmax}}$ bytes of memory.

5.1.1 Modified Files

- `patch/oct_tree/morton_keys.f90` — Morton key computation
- `patch/oct_tree/morton_hash.f90` — Hash table and helper functions
- `patch/oct_tree/morton_init.f90` — Initialization and verification
- `patch/oct_tree/refine_utils.f90` — Hash table maintenance
- `patch/oct_tree/nbors_utils.kjhan.f90` — Neighbor lookup functions

5.2 Morton Key

A 64-bit Morton key is computed by interleaving the 3D integer coordinates (21 bits per dimension):

$$\text{key} = \text{interleave}\left(\lfloor x_g \cdot 2^{l-1} \rfloor,\ \lfloor y_g \cdot 2^{l-1} \rfloor,\ \lfloor z_g \cdot 2^{l-1} \rfloor\right)$$

5.3 Hash Table

A per-level open-addressing hash table with linear probing and power-of-2 capacity maps Morton keys to grid indices. The table is maintained in `make_grid_coarse/fine` and `kill_grid`, with a full rebuild after each time step.

5.4 Neighbor Lookup

Two helper functions in the `morton_hash` module:

- `morton_nbor_grid(igrid, ilevel, j)` — returns `son(nbor(igrid,j))` equivalent
- `morton_nbor_cell(igrid, ilevel, j)` — returns `nbor(igrid,j)` equivalent

Direction convention: $j = 1{:}{-x},\ 2{:}{+x},\ 3{:}{-y},\ 4{:}{+y},\ 5{:}{-z},\ 6{:}{+z}$.

5.5 nbor Array Removal (Phase 4)

The `nbor` array is allocated as `allocate(nbor(1:1,1:1))` (minimum size to avoid compilation errors). All code paths use Morton lookup exclusively.

6 Memory-Based Load Balancing

6.1 Motivation

Standard RAMSES load balancing distributes cells evenly across CPUs, but cells with many particles consume significantly more memory. Memory-based balancing weights each cell by its memory footprint.

6.2 Cost Function

$$\text{cell\_cost} = \frac{\texttt{mem\_weight\_grid}}{\text{twotondim}} + \text{numbp(igrid)} \times \frac{\texttt{mem\_weight\_part}}{\text{twotondim}}$$

where:

- `mem_weight_grid` = 270 (default) — memory per grid in dp-equivalents
- `mem_weight_part` = 12 (default) — memory per particle in dp-equivalents

6.3 Implementation Details

- All histogram variables use 64-bit integers (`integer(i8b)`) with `MPI_INTEGER8`
- `numbp` is synchronized for virtual/reception grids before cost computation, then restored afterwards
- The `numbp` restore uses a save/restore pattern to avoid breaking the particle tree

6.4 Parameters

Controlled by three namelist parameters in `&RUN_PARAMS`:

- `memory_balance = .true.` — enable memory-based balancing
- `mem_weight_grid = 270` — grid memory weight
- `mem_weight_part = 12` — particle memory weight

7 Memory Savings: Large Array Optimization

7.1 Overview

Several large arrays were eliminated or converted to on-demand allocation to reduce steady-state memory usage by ∼960 MB (for `ngridmax`=5M).

Array	Strategy	Savings
`hilbert_key`	`allocate(1:1)` for ksection	∼640 MB
`bisec_ind_cell` + `cell_level`	On-demand alloc/dealloc	∼320 MB
`defrag_map`	Local scratch during defrag	minor
`nbor`	`allocate(1:1,1:1)` (Morton)	∼240 MB
`sink arrays`	`nsinkmax` 4M→2000	∼15 GB/rank

7.2 Sink Particle Array Reduction

The original `nsinkmax` was hardcoded to 4,000,000, allocating ∼15 GB per rank for sink-related arrays. Since typical simulations use $O(10)$–$O(100)$ sink particles, the default was reduced to 2000 (overridable via `nsinkmax` in `&AMR_PARAMS`). The `MPI_ALLREDUCE` calls in `synchro_fine` and `move_fine` were also changed to use `nsink` instead of `nsinkmax`, reducing message size by up to 333,000×.

8 IC Reading with Stream Access

8.1 Motivation

Sequential Fortran I/O requires reading all preceding planes to reach a target plane, which is $O(n^2)$ for large files. Stream access enables direct byte-offset seeks.

8.2 Implementation

Fortran 2003 `ACCESS='STREAM'` is used with computed byte offsets:

```
hdr_bytes = 52 + (i3-1)*plane_bytes + 5
plane_bytes = n1*n2*4 + 8  ! data + 2 record markers
```

Applied to hydro IC (deltab, velocity, temperature), particle velocity and position files. Only for `multiple=.false.` mode.

8.2.1 Modified Files

- `patch/Horizon5-master-2/init_flow_fine.f90`
- `init_part.f90`

9 Load Balance Profiling and Tuning

9.1 Internal Timing

Detailed timing breakdown was added to `load_balance` in `patch/cuda/load_balance.kjhan.f90`:

Section	Description	Typical (s/step)
`numbp_sync`	MPI sync of numbp for virtual grids	0.8–1.0
`cmp_new_cpu_map`	Build ksection + compute new map	0.4–0.6
`expand_pass`	build_comm + make_virtual loop	0.8–1.5
`grid_migration`	Linked-list reconnection	< 0.01
`allreduce+cpumap_update`	MPI_ALLREDUCE $\times 4$ + cpu_map	2.3–3.3
`shrink_pass`	flag_fine + build_comm loop	0.4–1.0

9.2 nremap Tuning

The `nremap` parameter controls load balancing frequency (every N coarse steps). Testing with 200M particles on 12 ranks showed:

nremap	Total (s)	Loadbal (s)	Speedup	Note
1	303.8	64.4 (21.2%)	—	Baseline
3	269.9	24.7 (9.1%)	1.13×	
5	249.8	15.7 (6.3%)	**1.22×**	**Optimal**
10	258.6	11.6 (4.5%)	1.17×	Imbalance grows

Default Setting

`nremap=5` is set as the default. All four configurations produce **bit-identical** results: `econs=3.77E-03`, `epot=-1.88E-06`, `ekin=1.23E-06` at step 10.

9.3 Min/Max Memory Reporting

The `writemem_minmax` subroutine prints per-step min/max memory usage across all MPI ranks.

10 Zoom-In Simulation Setup

10.1 Overview

This chapter describes how to configure and run a cosmological zoom-in simulation with cuRAMSES, targeting $\sim$1 kpc physical resolution within a selected sub-region of a larger cosmological volume. The setup uses:

- **Box size**: $10\,h^{-1}$ Mpc (comoving)
- **Zoom region**: $\sim 1.25\,h^{-1}$ Mpc (centered, with padding from MUSIC)
- **IC levels**: 7–11 (generated by MUSIC), corresponding to base 128^3 up to effective 2048^3
- **AMR levels**: up to 14 (adaptive refinement beyond IC levels)
- **Physics**: hydrodynamics, gravity, radiative cooling, star formation, and AGN feedback

The main challenge in zoom-in simulations is preventing the AMR machinery from refining the entire box (which would exhaust memory). This is solved by using a passive scalar variable as a *zoom geometry mask* that restricts refinement to the zoom region during initialization.

10.2 Initial Conditions

10.2.1 MUSIC Configuration

Initial conditions are generated by the MUSIC code with a multi-level zoom setup. The MUSIC configuration file specifies the base resolution, zoom region, and cosmological parameters.

zoomin_10Mpc.conf

```
[setup]
boxlength       = 10            # Box size in Mpc/h
zstart          = 50            # Starting redshift
levelmin        = 7             # Base grid: 2^7 = 128^3
levelmin_TF     = 7             # Transfer function grid
levelmax        = 11            # Zoom finest: 2^11 = 2048^3
    effective
padding         = 8
overlap         = 4
ref_center      = 0.5, 0.5, 0.5
ref_extent      = 0.04, 0.04, 0.04
align_top       = yes
baryons         = yes
use_2LPT        = yes
periodic_TF     = yes

[cosmology]
# Planck 2018 (TT,TE,EE+lowE+lensing)
```

```
Omega_m          = 0.3111
Omega_L          = 0.6889
Omega_b          = 0.04
H0               = 67.66
sigma_8          = 0.8102
nspec            = 0.9665
transfer         = eisenstein

[output]
format           = grafic2
filename         = IC_zoomin
```

10.2.2 Generated IC Structure

MUSIC produces a directory hierarchy with one sub-directory per refinement level:

Level	Grid Size	Resolution	Coverage
7	128^3	$\sim$78 kpc/h	Entire box
8	variable	$\sim$39 kpc/h	Zoom region + padding
9	variable	$\sim$20 kpc/h	Zoom region + padding
10	variable	$\sim$10 kpc/h	Zoom region + padding
11	variable	$\sim$5 kpc/h	Zoom region (finest IC)

Each level directory contains GRAFIC2 binary files: `ic_deltab` (baryon overdensity), `ic_velcx/y/z` (baryon velocities), `ic_velcdx/y/z` (dark matter velocities), `ic_poscdx/y/z` (dark matter position offsets), and `ic_refmap` (refinement map, level 7 only).

10.3 Zoom Geometry Scalar (`ic_pvar_00006`)

Important

This is the single most critical aspect of the zoom-in setup. Incorrect configuration of the zoom geometry scalar will cause the entire box to be refined, leading to immediate memory exhaustion.

10.3.1 The Problem

In a standard RAMSES simulation, the `init_refmap` subroutine (called when `ivar_refine=0`) reads the `ic_refmap` file and sets `cpu_map2=1` for cells that should be refined. However, the default `flag_utils` logic using `cpu_map2` applies refinement uniformly wherever the map is set, without distinguishing between zoom and background regions at higher AMR levels. This causes the entire box to be recursively refined, rapidly exhausting memory.

10.3.2 The Solution: Passive Scalar Mask

The solution uses the 6th passive scalar variable (`ic_pvar_00006`) as a zoom geometry indicator. With `NVAR=11` (5 hydro + 1 metal + 5 passive), the 6th passive scalar corresponds to hydro variable index 11. Setting `ivar_refine=11` activates a different refinement criterion in `flag_utils`:

```
! In flag_utils.kjhan.f90 (during init):
if(ivar_refine > 0) then
   do i=1,ncell
      ok(i) = ok(i) .or. &
           (uold(ind_cell(i),ivar_refine) / uold(ind_cell(i)
              ,1) &
            > var_cut_refine)
   end do
end if
```

This checks whether the passive scalar (divided by density for the conserved-to-primitive conversion) exceeds `var_cut_refine` (typically 0.01). Cells in the zoom region have a scalar value of 1.0 and pass the test; background cells have 0.0 and are skipped.

10.3.3 How It Works at Each Stage

1. **During initialization** (`init=.true.`): When `ivar_refine` is nonzero, `init_refmap` is *not* called (line 31 of `amr/init_refine.f90`):

   ```
   if(ivar_refine==0) call init_refmap
   ```

   Instead, `init_flow` loads the hydro IC (including `ic_pvar_00006`) and the refinement flag is set by the passive scalar criterion: $\texttt{uold}(\text{cell}, 11)/\texttt{uold}(\text{cell}, 1) > 0.01$.
2. **After initialization** (`init=.false.`): The refinement criterion switches to the standard density-based Lagrangian criterion (`m_refine`). The `cpu_map2` array is set by `rho_fine` with the `mass_cut_refine` parameter filtering out heavy background dark matter particles, ensuring that only the zoom region (populated by light, high-resolution particles) triggers further refinement.

10.3.4 Creating `ic_pvar_00006`

The Python script `test_ksection/create_pvar006.py` generates the zoom geometry scalar for each IC level:

- **Level 7** (base): Reads `ic_refmap` from the level 7 directory. Nonzero entries (zoom cells) are set to 1.0; zero entries (background) are set to 0.0. The result is written in GRAFIC2 format as `ic_pvar_00006`.
- **Levels 8+** (zoom sub-levels): All cells are set to 1.0, since the entire grid at these levels lies within the zoom region by construction.

create_pvar006.py (core logic)

```
for level in LEVELS:
    if level == base_level:
        # Read ic_refmap, convert: nonzero -> 1.0, zero -> 0.0
        refmap = read_grafic2_data(refmap_file, n1, n2, n3)
        pvar = np.where(refmap != 0, 1.0, 0.0).astype(np.
            float32)
    else:
        # Zoom sub-levels: all cells = 1.0
        pvar = np.ones((n3, n2, n1), dtype=np.float32)
    write_grafic2(pvar_file, header_bytes, pvar)
```

Editing the Script

Before running, edit the `IC_DIR` variable in `create_pvar006.py` to point to your IC directory, and adjust `LEVELS` to match the levels generated by MUSIC.

10.3.5 mass_cut_refine

After initialization, `rho_fine` computes the density field used to set `cpu_map2` for refinement flagging. The `mass_cut_refine` parameter acts as a mass filter:

```
! In rho_fine.f90:
if(mass_cut_refine > 0.0) then
   do j = 1, np
      if(ttt(j) == 0d0) then           ! dark matter only
         ok(j) = ok(j) .and. mmm(j) < mass_cut_refine
      endif
   end do
endif
```

Particles heavier than `mass_cut_refine` are excluded from the density computation, so only high-resolution zoom particles contribute to the refinement map. Set `mass_cut_refine` to a value between the zoom-region DM particle mass and the next-coarser level's DM particle mass. Refer to the table in Chapter 1 for recommended values.

10.4 Memory Considerations

10.4.1 ngridtot Sizing

RAMSES allocates all grid-related arrays at startup based on `ngridmax` (per-CPU maximum grids). When `ngridtot` is specified, `ngridmax` is computed as `ngridtot/ncpu`. The critical constraint is:

Important

The **total virtual memory** allocated by all MPI ranks on a node must not exceed the kernel's `CommitLimit`. Exceeding this causes the OOM killer to terminate the process (SIGNAL 9).

10.4.2 CommitLimit

The Linux kernel's `CommitLimit` determines the maximum total virtual memory that can be allocated:

$$\texttt{CommitLimit} = \texttt{RAM} \times \frac{\texttt{overcommit\_ratio}}{100} + \texttt{swap}$$

Check the current value:

Check CommitLimit

```
grep CommitLimit /proc/meminfo
```

On systems without swap (common for HPC nodes), `CommitLimit = RAM × overcommit_ratio/100`. The default `overcommit_ratio` is 50, so a 256 GB node has ∼128 GB CommitLimit.

10.4.3 Virtual vs. Physical Memory

RAMSES allocates the full `ngridmax`-sized arrays at startup, consuming virtual memory immediately. Physical (resident) memory grows as grids are actually created during the simulation. The key arrays that dominate virtual memory usage are:

Array	Size	Per-grid bytes
`uold(1:ncell,1:nvar)`	$8 \times 8 \times$ nvar	$8 \times 8 \times 11 = 704$
`unew(1:ncell,1:nvar)`	same	704
`f(1:ncell,1:3)`	$8 \times 8 \times 3$	192
`son/father/next/prev`	8×8 each	$4 \times 64 = 256$
`phi/rho`	8×8 each	$2 \times 64 = 128$
`flag1/flag2/cpu_map/map2`	4×8 each	$4 \times 32 = 128$

As a rough estimate, each grid (oct = 8 cells) requires ∼2–3 kB of virtual memory across all arrays. For `ngridmax`=25 million (i.e., `ngridtot`=200M with 8 CPUs), each rank allocates ∼50–75 GB of virtual memory. Ensure that `ncpu` × `per_rank_virtual` < `CommitLimit`.

11 HDF5 I/O

11.1 Overview

Standard RAMSES writes one binary file per MPI process per data type (AMR, hydro, gravity, particles), producing $O(N_{\rm cpu} \times 4)$ files per snapshot. This creates file management challenges at high core counts and requires the same number of MPI processes for restart.

The HDF5 I/O module replaces the per-CPU binary output with a single HDF5 file per snapshot, using MPI parallel I/O (collective hyperslab writes) for performance. Key benefits:

- **Single file**: one `data_NNNNN.h5` per snapshot instead of thousands of files
- **Parallel I/O**: MPI-IO backend ensures scalable write/read performance
- **Self-describing**: HDF5 groups and attributes contain all metadata
- **Cross-format**: `informat` and `outformat` can differ, enabling binary-to-HDF5 conversion

Compilation

HDF5 support requires compilation with `make HDF5=1`. The HDF5 library must be built with `-enable-parallel` and `-enable-fortran` for the Intel `ifx` compiler. See Section 11.2 for build instructions.

11.2 Building HDF5

The HDF5 library must be compiled from source with parallel (MPI-IO) and Fortran support matching the Intel compiler used for RAMSES.

Build HDF5 from source

```
wget https://github.com/HDFGroup/hdf5/releases/download/\
  hdf5_1.14.5/hdf5-1.14.5.tar.gz
tar xf hdf5-1.14.5.tar.gz && cd hdf5-1.14.5
CC=mpiicc FC=mpiifx ./configure \
  --enable-fortran --enable-parallel \
  --prefix=$HOME/local/hdf5
make -j8 && make install
```

The key configure flags:

- `-enable-parallel`: MPI-IO support for collective parallel reads/writes
- `-enable-fortran`: generates `hdf5.mod` (Fortran module) compatible with `ifx`

Then build RAMSES with the `HDF5=1` flag:

Build RAMSES with HDF5

```
cd bin
make clean
make HDF5=1
```

The Makefile conditionally adds:

```
HDF5_DIR = $(HOME)/local/hdf5
FFLAGS += -DHDF5 -I$(HDF5_DIR)/include
LIBS += -L$(HDF5_DIR)/lib -lhdf5_fortran -lhdf5 -lz \
        -Wl,-rpath,$(HDF5_DIR)/lib
```

Without `HDF5=1`, the code compiles identically to before (all HDF5 code is guarded by `#ifdef HDF5`).

11.3 Namelist Parameters

Two parameters in `&OUTPUT_PARAMS` control the I/O format:

Parameter	Type	Default	Description
`outformat`	char(10)	`'original'`	Output format: `'original'` or `'hdf5'`
`informat`	char(10)	`'original'`	Restart format: `'original'` or `'hdf5'`

Example namelist

```
&OUTPUT_PARAMS
noutput=1
aout=1.0
foutput=5
outformat='hdf5'     ! write HDF5 snapshots
/
```

For restart from an HDF5 checkpoint:

Restart from HDF5

```
&RUN_PARAMS
nrestart=1
/
&OUTPUT_PARAMS
informat='hdf5'      ! read HDF5 checkpoint
outformat='hdf5'     ! continue writing HDF5
/
```

11.4 HDF5 File Structure

Each snapshot produces a single file `output_NNNNN/data_NNNNN.h5` with the following group hierarchy:

```
data_NNNNN.h5
  /header/                (attrs: ncpu, ndim, nlevelmax, nstep,
                           boxlen, time, aexp, H0, omega_m, ...)
      tout[noutput]       (dataset)
      aout[noutput]       (dataset)
      dtold[nlevelmax]    (dataset)
      dtnew[nlevelmax]    (dataset)
  /domain/                (ksection tree or bound_key)
  /coarse/
      son[ncoarse]        (dataset: integer)
      cpu_map[ncoarse]    (dataset: integer)
  /amr/level_LL/
      xg[ngrid_total x 3]            (grid centres)
      son[ngrid_total x twotondim]    (son indices)
      cpu_map[ngrid_total x twotondim]
      nbor[ngrid_total x twondim]     (Morton-computed)
      ngrid_per_cpu[ncpu]
  /hydro/level_LL/
      uold[ngrid_total x twotondim x nvar]
  /gravity/level_LL/
      phi[ngrid_total x twotondim]
      f[ngrid_total x twotondim x ndim]
  /particles/
      x[npart_total x ndim]     (positions)
      v[npart_total x ndim]     (velocities)
      m[npart_total]             (masses)
      id[npart_total]            (particle IDs)
      level[npart_total]         (AMR level)
      tp[npart_total]            (birth time)
      zp[npart_total]            (metallicity)
      npart_per_cpu[ncpu]
  /sinks/
      idsink, msink, xsink, vsink, tsink, ...
```

11.4.1 Parallel Write Strategy

Each AMR level is written using collective MPI-IO hyperslabs:

1. `MPI_Allgather(ngrid_local)` to compute per-CPU offsets
2. Each CPU writes its portion via an HDF5 hyperslab selection
3. Collective I/O ensures efficient parallel access to the shared file

Particles use the same hyperslab pattern: `MPI_Allgather(npart_local)` for offsets, then collective write.

11.5 Implementation Files

File	Description
`patch/cuda/ramses_hdf5_io.f90`	HDF5 wrapper module (create, open, write, read helpers)
`patch/cuda/backup_hdf5.f90`	HDF5 output: `dump_all_hdf5()`
`patch/cuda/restore_hdf5.f90`	HDF5 restart: `restore_amr_hdf5()`, etc.

11.5.1 Dispatch

The HDF5 output is dispatched from `dump_all` in `output_amr.kjhan.f90`:

```
#ifdef HDF5
if(outformat == 'hdf5') then
   call dump_all_hdf5(filedir, nchar)
   goto 998  ! skip binary output
end if
#endif
```

For restart, each `init_*` subroutine checks `informat`:

```
#ifdef HDF5
if(informat == 'hdf5') then
   call restore_amr_hdf5()
   return
end if
#endif
```

11.6 Verification

The HDF5 output was tested with a 5-step cosmological simulation (12 MPI ranks, levelmin=8, levelmax=10, 200M particles):

Metric	Value
Output file	`data_00001.h5` (3.65 GB)
Step count	5
econs at step 5	4.85E-03
I/O time	4.1 s (16.1% of total)
Total runtime	20.6 s

11.7 Variable-NCPU Restart: Distributed Grid Creation

When restarting an HDF5 checkpoint with a different number of MPI ranks (`ncpu_file` $\neq$ `ncpu`), the original implementation allocated *all* grids from the file on *every* rank. For a file with 11.17 M grids, this required `ngridmax` $\geq$ 11.17 M per rank, consuming $\sim$20 GB per rank. With 32 ranks the total memory exceeded 640 GB, causing out-of-memory failures on systems with $\leq$560 GB.

11.7.1 Distributed Algorithm

The new implementation creates only the grids each rank actually needs. For each level $L = 1$ to `nlevelmax_file`:

1. **Phase 1 — Active grid creation.** All ranks read the file data for level L (grid positions and `son_flag`), but each rank only allocates grids whose father cell satisfies `cpu_map(father_cell) == myid`. For $L \geq 2$, the father grid is located via Morton hash lookup at level $L-1$; if the lookup returns zero (father not present on this rank), the grid is not local and is skipped. Active grids have their `xg`, `flag1`, `cpu_map`, `cpu_map2`, `father`, and `son` set from the file data, and are inserted into the Morton hash table.

2. **Phase 2 — Virtual grid creation.** Virtual (ghost) grids are created using RAMSES's existing refinement infrastructure:
   - $L = 1$: `flag_coarse` → `refine_coarse`
   - $L \geq 2$: `refine_fine(`$L-1$`)`

   Since active grids already set `son(father_cell)` > 0, `refine_fine` skips those cells and only creates virtual grids where `flag1 == 1` and `son == 0`. The flag `balance = .true.` causes `make_grid_fine` to skip hydro variable interpolation (line 791 of `refine_utils.f90`).
3. **Communication setup.** `build_comm(`L`)` establishes emission/reception arrays, followed by `make_virtual_fine_int` exchanges for `flag1`, `cpu_map`, and `cpu_map2`. This ensures the next level's `refine_fine` has correct flag values on virtual grids.

After all levels, hydro and Poisson data are restored from the file (same read-all-scatter pattern) and exchanged to virtual grids via `make_virtual_fine_dp`.

11.7.2 Key Design Decisions

- **Father lookup as filter.** For $L \geq 2$, the Morton hash lookup at level $L-1$ serves as an implicit ownership test: if the father grid does not exist locally, the child grid cannot be active on this rank.
- **Hash table sizing.** The per-level hash table is sized for local grids: `max(4 * (ngrid_total / ncpu + 1), 16)` instead of `2 * ngrid_total`, reducing hash table memory proportionally.
- **No changes to other files.** `refine_coarse`, `refine_fine`, `build_comm`, `authorize_fine`, and `make_grid_fine` all work unmodified. The same-ncpu restart path is also unchanged.

11.7.3 Verification

A 12-CPU HDF5 checkpoint (levelmin=8, levelmax=10, 200M particles) was restarted with 8 CPUs. The reference is the same checkpoint restarted with 12 CPUs (same-ncpu path).

Metric	**Reference (12→12)**	**Distributed (12→8)**
ngrid per rank	2,396,745	352,379
Memory per rank	8.0 GB	6.9 GB
Grid reduction	—	6.8×

Step	**econs**		**epot**		**ekin**	
	Ref	Dist	Ref	Dist	Ref	Dist
6	8.18E-03	8.17E-03	−1.01E-06	−1.01E-06	6.49E-07	6.49E-07
7	7.26E-03	7.25E-03	−1.26E-06	−1.26E-06	8.17E-07	8.17E-07
8	6.43E-03	6.42E-03	−1.48E-06	−1.48E-06	9.63E-07	9.63E-07
9	5.79E-03	5.79E-03	−1.68E-06	−1.68E-06	1.09E-06	1.09E-06
10	5.26E-03	5.25E-03	−1.88E-06	−1.88E-06	1.23E-06	1.23E-06

The epot and ekin values are **identical** across all steps. The econs differences (last display digit) arise from different domain decomposition with 8 vs. 12 CPUs. With 32 CPUs the grid reduction would be ∼20×, bringing the previously OOM scenario (640 GB) well within a 560 GB memory budget.

11.8 Poisson MG Fine-Level Optimization

The multigrid Poisson solver (`poisson-mg`) is typically the single largest time consumer, accounting for 29–55% of total runtime. Several optimizations were applied to the fine-level V-cycle in `poisson/multigrid_fine_fine.kjhan.f90`:

11.8.1 Precomputed Neighbor Grid Array

Before entering the V-cycle iteration loop, all 6-directional neighbor grids are precomputed and stored in a contiguous array:

```
! nbor_grid_fine(0:twondim, 1:ngrid)
!   index 0 = self (igrid_amr)
!   index 1..6 = neighbor grids in -x,+x,-y,+y,-z,+z
call precompute_nbor_grid_fine(ilevel)
```

This replaces per-cell `morton_nbor_grid` calls inside the Gauss-Seidel and residual loops with simple array lookups, eliminating hash table probes from the innermost loops.

11.8.2 Merged Red-Black Gauss-Seidel Exchange

The standard multigrid implementation performs a ghost zone exchange after each half-sweep of the red-black Gauss-Seidel smoother:

$$\texttt{red} \rightarrow \texttt{exchange} \rightarrow \texttt{black} \rightarrow \texttt{exchange}$$

This was simplified to:

$$\texttt{red} \rightarrow \texttt{black} \rightarrow \texttt{exchange}$$

The black sweep uses slightly stale ghost values from the red sweep (“chaotic relaxation”), which does not affect multigrid convergence. This reduces the number of MPI exchanges per iteration from 9 to 5 (a 44% reduction in communication calls).

11.8.3 Residual and Norm Single Pass

The residual computation and the L^2 norm reduction were fused into a single pass:

```
! Optional norm2 argument: if present, compute
! both residual and L2 norm in one sweep
call cmp_residual_mg_fine(ilevel, norm2)
```

This eliminates a redundant loop over all cells when both the residual and its norm are needed (at the first iteration and after post-smoothing).

11.8.4 Division to Multiplication

In the Gauss-Seidel fast path, the division by `dtwondim` was replaced with multiplication by a precomputed reciprocal:

```
real(dp) :: oneoverdtwondim
oneoverdtwondim = 1.0d0 / dble(twondim)
! In loop:
phi = sum_neighbors * oneoverdtwondim   ! was: / dtwondim
```

11.8.5 Performance Impact

These optimizations combined reduce the Poisson solver's share of total runtime:

Metric	Value
Before optimization	55.1% of runtime
After optimization	38.6% of runtime
MPI exchange reduction	44% fewer calls
Iteration count	unchanged (Level 8: 5, Level 9: 4)
Convergence	verified (econs = 3.79E-03 at step 10)

Chaotic Relaxation

The merged red-black exchange introduces a minor change in the energy conservation value (3.77E-03 $\rightarrow$ 3.79E-03) due to the slightly different relaxation path. This is well within the multigrid tolerance and does not affect physical results. The membal and nomembal tests produce identical values.

12 SNII Feedback K-Section Exchange

When `ordering='ksection'`, the SNII kinetic feedback routine (`kinetic_feedback` in `feedback.kjhan3.f90`) replaces all `MPI_ALLREDUCE` calls with a 3-stage k-section overlap exchange, eliminating global synchronization.

12.1 3-Stage Exchange

1. **Exchange 1 (Overlap Broadcast)**: Each CPU broadcasts its local SN events to all CPUs whose domain overlaps the SN blast radius. Uses `ksection_exchange_dp_overlap` with bounding box $[\mathbf{x}_{\rm SN} - r_{\rm SN},\ \mathbf{x}_{\rm SN} + r_{\rm SN}]$ and `periodic=.true.`. Carries $9 + n_{\rm elt}$ fields per SN event (`x`, `v`, `m`, `Z`, `NbSN`, `ce`).
2. **Exchange 2 (Exclusive $\rightarrow$ Owner)**: Each CPU computes its local contribution to the Sedov blast integrals (`vol_gas`, `mloadSN`, `ZloadSN`, `vloadSN`, `dq`, `u2Blast`, `celoadSN`) and sends them to the SN owner CPU via `ksection_exchange_dp`. The owner aggregates all contributions.
3. **Exchange 3 (Overlap Broadcast)**: The owner broadcasts the aggregated blast parameters back to all CPUs in the blast region. Uses `ksection_exchange_dp_overlap` with the same bounding box.

12.2 Dispatch

```
if(TRIM(ordering)=='ksection') then
   call kinetic_feedback_ksec(...)   ! 3-stage exchange
else
   call kinetic_feedback_allreduce(...)   ! packed ALLREDUCE
end if
```

12.3 Verification

The k-section and Hilbert paths produce identical physical results: `epot`, `ekin`, and `eint` match to full precision.

13 Sink Particle K-Section Exchange

When `ordering='ksection'`, the 8 packed `MPI_ALLREDUCE(MPI_SUM)` calls in `sink_particle.kjhan.f90` are replaced with a 2-stage k-section exchange via the utility subroutine `sink_ksec_reduce_sum`.

13.1 Algorithm: 2-Stage K-Section Reduce-Sum

Since `MPI_ALLREDUCE` = reduce + broadcast, the replacement decomposes this into:

1. **Stage 1 (Exclusive → Owner)**: Each CPU sends its non-zero partial sums to the sink's owner CPU. The owner is determined by `cmp_ksection_cpumap(xsink)`. A sparsity optimization skips sinks with zero contribution. Uses `ksection_exchange_dp` with $n_{\text{fields}} + 1$ properties per item (sink index + field values).
2. **Stage 2 (Overlap Broadcast → All)**: Each owner broadcasts its accumulated result to all CPUs using `ksection_exchange_dp_overlap` with a full-domain bounding box $[0, \texttt{boxlen}]^3$ and `periodic=.true.`, ensuring all CPUs receive the global sum.

13.2 Replaced Call Sites

#	Subroutine	nfields	Arrays
1	`kjhan_make_sink`	15	oksink, msink, xsink(3), vsink(3), tsink, dMsmbh, Esave, bhspin(3), spinma
2	`bondi_hoyle` (1st)	3	oksink, c2sink, v2sink
3	`bondi_hoyle` (2nd)	9	wdens, wvol, wc2, wmom(3), jsink(3)
4	`grow_bondi`	7	msink, vsink(3), dMBH_coarse, dMEd_coarse, dMsmbh
5	`grow_jeans`	4	msink, vsink(3)
6	`AGN_feedback`	1	Esave
7	`AGN_blast`	5	vol_gas, mass_gas, mAGN, ZAGN, psy_norm
8	`update_sink`	dynamic	sink_stat: $(n_{\text{levelmax}} - n_{\text{levelmin}} + 1) \times 7$

13.3 Preserved ALLREDUCE Calls

The following calls are *not* replaced because they use different operations, data types, or dynamic sizes:

- Scalar/integer reductions (`ntot`, `ntot_sink_cpu`, `numbp_free`)
- `MPI_INTEGER` reduction (`idsink_new`)
- `MPI_MIN` reduction (`dMsmbh_new`)
- Dynamic-size mixed-type reductions (`x_tmp`, `dens_tmp`, `flag_tmp`)

13.4 Verification

Test	CPUs	econs
K-section	12	7.65E-04
Hilbert fallback	4	7.47E-04

Both match pre-replacement values exactly.

14 MPI_ALLREDUCE Packing

Multiple individual MPI_ALLREDUCE calls operating on arrays of the same size and reduction operation can be packed into a single call, reducing MPI latency overhead proportionally.

14.1 AGN Feedback (average_AGN)

In sink_particle.kjhan.f90, the average_AGN subroutine synchronizes five per-sink arrays (vol_gas, mass_gas, mAGN, ZAGN, psy_norm) across all ranks. The five individual MPI_ALLREDUCE calls were packed into a single call with buffer size $5 \times$ nsink.

> The dMsmbh_new (MPI_MIN) and Esave_new (MPI_SUM) reductions use different operations and therefore cannot be packed together.

14.2 SNII Feedback — Hilbert Fallback Path

When ordering $\neq$ 'ksection', the SNII feedback in feedback.kjhan3.f90 uses MPI_ALLREDUCE instead of the k-section overlap exchange. Two packing optimizations were applied:

1. **SN property exchange** (6→1): xSN, vSN, mSN, ZSN, NbSN, ceSN packed into a single buffer of size $n_{\mathrm{SN,tot}} \times (9 + n_{\mathrm{elt}})$.
2. **Sedov blast exchange** (7→1): vol_gas, mloadSN, ZloadSN, vloadSN, dq, u2Blast, celoadSN packed into a single buffer of size $n_{\mathrm{SN,tot}} \times (12 + n_{\mathrm{elt}})$.

14.3 Summary

Location	Subroutine	Before	After
sink_particle.kjhan.f90	average_AGN	5 calls	1 call
feedback.kjhan3.f90	kinetic_feedback	6 calls	1 call
feedback.kjhan3.f90	average_SN	7 calls	1 call
Total ALLREDUCE removed		18	3

Verification: econs = 7.65E-04 (ksection, 12 CPU) and 7.47E-04 (Hilbert, 4 CPU), identical to pre-packing results. The Hilbert path SNII ALLREDUCE time decreased from 4.25 ms to 1.16 ms (3.7× speedup).

15 AGN Feedback Spatial Binning

The AGN feedback routines `average_AGN` and `AGN_blast` in `sink_particle.kjhan.f90` iterate over all AGN sinks for every leaf cell, resulting in $O(n_{\text{cells}} \times n_{\text{AGN}})$ complexity. With 32K sinks, profiling showed `average_AGN` consuming 177 s (68.5%) and `AGN_blast` consuming 73 s (27.3%) of the AGN feedback time.

The same spatial binning technique proven in the SNII feedback optimization (Chapter **??**) was applied: the simulation domain is divided into n_{bin}^3 cells, each AGN sink is inserted into a linked list indexed by its bin, and the cell loop checks only the 27 neighboring bins.

15.1 Algorithm

1. **Bin size**: $n_{\text{bin}} = \max\big(1, \min(128, \lfloor \texttt{boxlen}/r_{\text{max}} \rfloor)\big)$, where $r_{\text{max}} = \max(dx_{\text{min}}\, l_{\text{scale}}/a,\ r_{\text{AGN}} \times 3.08 \times 10^{21})/l_{\text{scale}}$. The upper bound of 128 keeps the bin array ($128^3 \times 4$ bytes = 8 MB) within L3 cache.
2. **Linked list construction** ($O(n_{\text{AGN}})$): For each AGN, compute its bin index and prepend it to that bin's list via `agn_next(iAGN) = bin_head(ibx,iby,ibz)`.
3. **Cell loop** ($O(n_{\text{cells}} \times n_{\text{nearby}})$): For each leaf cell, compute its bin and iterate over the $3 \times 3 \times 3 = 27$ neighboring bins, traversing each linked list and applying the existing distance checks.

Since $r_{\text{max}} \ll$ `bin_size` (typically by a factor of 50), the 27-bin search radius far exceeds r_{max}, guaranteeing that all AGN within range are found.

15.2 Implementation

Both `average_AGN` and `AGN_blast` follow the identical pattern:

Bin construction (after rmax computation)

```
nbin=max(1,min(128,int(boxlen/rmax)))
bin_size=boxlen/dble(nbin)
inv_bin_size=dble(nbin)/boxlen
allocate(bin_head(nbin,nbin,nbin),agn_next(max(1,nAGN)))
bin_head=0; agn_next=0
do iAGN=1,nAGN
   ibx=max(1,min(nbin,int(xAGN(iAGN,1)*inv_bin_size)+1))
   iby=max(1,min(nbin,int(xAGN(iAGN,2)*inv_bin_size)+1))
   ibz=max(1,min(nbin,int(xAGN(iAGN,3)*inv_bin_size)+1))
   agn_next(iAGN)=bin_head(ibx,iby,ibz)
   bin_head(ibx,iby,ibz)=iAGN
end do
```

Cell inner loop (replaces `do iAGN=1,nAGN`)

```
ibx=max(1,min(nbin,int(x*inv_bin_size)+1))
iby=max(1,min(nbin,int(y*inv_bin_size)+1))
ibz=max(1,min(nbin,int(z*inv_bin_size)+1))
do jbz=max(1,ibz-1),min(nbin,ibz+1)
do jby=max(1,iby-1),min(nbin,iby+1)
do jbx=max(1,ibx-1),min(nbin,ibx+1)
   iAGN=bin_head(jbx,jby,jbz)
   do while(iAGN > 0)
      ! existing distance check + 3-case handling
      iAGN=agn_next(iAGN)
   end do
end do; end do; end do
```

All three AGN feedback cases (saved energy, jet, thermal) and the OpenMP thread-local accumulation arrays are preserved unchanged. The bin variables `ibx,iby,ibz,jbx,jby,jbz` are added to the `!$omp parallel do private` clause; `bin_head` and `agn_next` are shared (read-only).

15.3 Performance

Benchmark: output_00005 (step 241, $a \approx 0.164$, 32K sinks), 4 MPI ranks, OMP_NUM_THREADS=1.

Subroutine	Before (s)	After (s)	Speedup
`average_AGN`	177.2	5.9	30×
`AGN_blast`	72.6	5.0	14.4×
AGN TOTAL	264.6	11.0	24×
sinks TOTAL	633.8	165.1	3.8×

15.4 Verification

econs = 6.51E-04, bit-identical to the pre-optimization result. All three AGN feedback cases (saved energy / jet / thermal) produce identical physics output.

15.5 Modified File

File	Changes
`sink_particle.kjhan.f90`	`average_AGN`: spatial binning of AGN inner loop `AGN_blast`: spatial binning of AGN inner loop

16 Scaling Performance

Strong scaling tests were performed using a 200×10^6 particle cosmological simulation (levelmin=8, levelmax=10), 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 via SMT) with 1 TB of DDR4 memory. The code was compiled with Intel MPI (`mpiifx`) and OpenMP (`-qopenmp`).

The variable-`ncpu` restart allows the 12-rank checkpoint to be read with any number of MPI ranks. A forced `load_balance` on the first coarse step ensures optimal grid distribution.

16.1 Pure MPI Scaling

All runs use `OMP_NUM_THREADS=1`. Speedup is relative to the 2-rank baseline.

Ranks	Elapsed (s)	Speedup	Coarse (s)	Particle (s)	Poisson (s)	MG (s)	Hydro-GZ (s)	Godunov (s)	LoadBal (s)
2	193.0	1.00×	2.27	20.92	18.22	102.69	0.52	46.67	9.14
4	154.8	1.25×	2.59	16.74	14.76	82.00	0.89	36.64	7.62
8	88.9	2.17×	2.12	8.93	8.18	43.46	1.09	21.68	4.90
16	58.2	3.31×	2.56	4.67	4.97	23.90	1.26	15.88	3.96
32	34.7	5.57×	2.04	2.27	2.73	12.59	1.11	8.71	2.27
64	25.9	7.45×	1.91	1.18	1.64	7.95	1.13	5.14	1.61

16.2 Hybrid MPI + OpenMP Scaling

All configurations use a fixed total of 64 physical cores. Speedup is relative to the 2-rank pure MPI baseline (193.0 s).

Ranks×Thr	Elapsed (s)	Speedup	Coarse (s)	Particle (s)	Poisson (s)	MG (s)	Hydro-GZ (s)	Godunov (s)	LoadBa (s)
64×1	25.9	7.45×	1.91	1.18	1.64	7.95	1.13	5.14	1.61
32×2	**22.2**	**8.71**×	1.55	1.37	1.64	7.23	0.82	5.19	1.63
16×4	24.9	7.76×	1.49	2.23	2.31	8.81	0.66	5.22	2.46
8×8	30.2	6.40×	1.26	3.80	3.34	11.35	0.68	5.46	3.06
4×16	44.4	4.34×	1.35	6.99	5.66	17.22	0.67	6.21	5.27

16.3 Key Observations

- **Optimal configuration** is 32 ranks $\times$ 2 threads (22.2 s), which is 14% faster than 64 pure MPI ranks (25.9 s) and 8.7$\times$ faster than the 2-rank baseline.
- **Multigrid Poisson solver (MG)** dominates runtime and scales well: 102.7 s (2 ranks) $\rightarrow$ 7.2 s (32$\times$2), a 14.2$\times$ speedup.
- **Godunov hydro solver** scales effectively with both MPI and OpenMP: 46.7 s (2 ranks) $\rightarrow$ 5.2 s (32$\times$2), a 9.0$\times$ speedup.
- **Ghost-zone exchange (Hydro-GZ)** increases slightly with MPI rank count (0.52 s at 2 ranks $\rightarrow$ 1.13 s at 64 ranks) due to more MPI messages, but *decreases* in hybrid mode (0.82 s at 32$\times$2) since fewer ranks means fewer messages.
- **Hybrid advantage**: 32$\times$2 beats 64$\times$1 because halving the MPI rank count reduces communication overhead (ghost zones, load balancing) while the 2 OpenMP threads provide sufficient intra-rank parallelism for compute-bound kernels.
- **Diminishing returns from OpenMP**: beyond 4 threads per rank, OpenMP thread synchronization overhead outweighs the communication savings, causing performance to degrade (4$\times$16: 44.4 s, worse than 8$\times$1: 88.9 s with only 8 total cores).
- **Load balancing** scales well: 9.1 s at 2 ranks $\rightarrow$ 1.6 s at 64 ranks. This is a per-`nremap` cost (default `nremap=5`).
- **Physics correctness**: all configurations produce identical $e_{\text{pot}} = -1.88 \times 10^{-6}$ and $e_{\text{kin}} = 1.23 \times 10^{-6}$ at step 10.

Recommended Configuration

For a single dual-socket AMD EPYC node with 64 cores, we recommend **32 MPI ranks $\times$ 2 OpenMP threads** for cuRAMSES cosmological simulations with $\sim 200 \times 10^6$ particles. Set `OMP_NUM_THREADS=2`, `OMP_PROC_BIND=close`, `OMP_PLACES=cores`, and `I_MPI_PIN_DOMAIN=omp` in the job script.

16.4 Per-Routine Scaling Analysis

Table 16.1 shows the complete timer breakdown for pure MPI scaling, and Table 16.2 for the hybrid configurations. All times are averages across ranks in seconds.

16.4.1 Scaling Categories

The routines fall into three distinct categories by their scaling behavior:

1. **Compute-bound (excellent MPI scaling).** `courant` (18.6$\times$), `particles` (17.7$\times$), `flag` (14.7$\times$), `poisson-mg` (12.9$\times$), `poisson` (11.1$\times$), `godunov` (9.1$\times$). These routines perform per-cell or per-particle work and scale nearly linearly with MPI rank count. In the hybrid regime, they are insensitive to the MPI/OMP ratio as long as total cores ≥ 32.
2. **Communication-bound (anti-scaling).** `hydro-gz` (ghost zone exchange) grows from 0.52 s to 1.13 s as ranks increase from 2 to 64, because more ranks means more boundary surfaces and MPI messages. In the hybrid regime, reducing ranks to 32 or fewer recovers some of this cost (0.82 s at 32$\times$2, 0.67 s at 4$\times$16). `io` (HDF5 output) degrades from 3.14 s at 2 ranks to 6.17 s at 64 ranks due to MPI collective I/O contention. This is the clearest anti-scaling component: halving the rank count from 64 to 32 nearly halves the I/O time (6.17 s $\rightarrow$ 3.37 s).
3. **Flat (Amdahl-limited).** `coarse levels` remains at $\sim$2 s regardless of configuration — this work is inherently serial or involves global communication at the coarsest level. It

Table 16.1: Full timer breakdown — pure MPI (`OMP_NUM_THREADS=1`).

Routine	2r	4r	8r	16r	32r	64r	Scale
poisson-mg	102.69	82.00	43.46	23.90	12.59	7.95	12.9×
godunov	46.67	36.64	21.68	15.88	8.71	5.14	9.1×
particles	20.92	16.74	8.93	4.67	2.27	1.18	17.7×
poisson	18.22	14.76	8.18	4.97	2.73	1.64	11.1×
flag	11.35	8.23	4.41	2.78	1.28	0.77	14.7×
loadbalance	9.14	7.62	4.90	3.96	2.27	1.61	5.7×
io	3.14	1.96	1.86	2.72	4.23	6.17	0.5× ↓
coarse levels	2.27	2.59	2.12	2.56	2.04	1.91	1.2×
courant	2.23	1.49	0.76	0.41	0.23	0.12	18.6×
hydro-gz	0.52	0.89	1.09	1.26	1.11	1.13	0.5× ↓
rev-gz	0.53	1.79	2.42	1.75	0.78	0.71	0.7× ↓
set unew	0.28	0.26	0.22	0.23	0.15	0.10	2.8×
set uold	1.11	0.77	0.38	0.24	0.15	0.10	11.1×
refine	0.61	0.86	0.69	0.62	0.43	0.27	2.3×
TOTAL	219.7	176.6	101.1	66.0	39.0	28.8	7.6×

Table 16.2: Full timer breakdown — hybrid MPI+OpenMP (fixed 64 cores).

Routine	64×1	32×2	16×4	8×8	4×16
poisson-mg	7.95	7.23	8.81	11.35	17.22
godunov	5.14	5.19	5.22	5.46	6.21
particles	1.18	1.37	2.23	3.80	6.99
poisson	1.64	1.64	2.31	3.34	5.66
io	6.17	3.37	2.25	2.02	1.89
loadbalance	1.61	1.63	2.46	3.06	5.27
flag	0.77	0.89	1.50	2.47	4.71
hydro-gz	1.13	0.82	0.66	0.68	0.67
rev-gz	0.71	0.57	0.83	0.56	0.58
coarse levels	1.91	1.55	1.49	1.26	1.35
courant	0.12	0.13	0.13	0.13	0.14
set uold	0.10	0.10	0.10	0.11	0.14
set unew	0.10	0.10	0.09	0.09	0.10
refine	0.27	0.28	0.31	0.25	0.40
TOTAL	28.8	24.9	28.4	34.6	51.3

becomes a larger fraction of runtime as other components scale (from 1% at 2 ranks to 6.6% at 64 ranks).

16.4.2 Bottleneck Shift

As the rank count increases, the dominant bottleneck shifts:

- At 2 ranks: `poisson-mg` (46.7%), `godunov` (21.2%), `particles` (9.5%)
- At 32×2: `poisson-mg` (29.1%), `godunov` (20.9%), `io` (13.6%)
- At 64×1: `poisson-mg` (27.6%), `io` (21.4%), `godunov` (17.8%)

At high rank counts, I/O emerges as the second-largest cost, suggesting that asynchronous or node-local I/O strategies would yield further speedup.

17 GPU Hydro Acceleration

cuRAMSES supports GPU acceleration for the main hydro solver routines via CUDA. The implementation uses a **dynamic hybrid CPU/GPU dispatch** model where OpenMP threads compete for a shared pool of GPU streams: threads that acquire a stream offload their work to the GPU, while the remaining threads continue on the CPU. This design naturally adapts to any ratio of CPU cores to GPU streams.

17.1 Architecture Overview

The GPU acceleration is enabled at build time with `make HDF5=1 USE_CUDA=1`. At runtime, each MPI rank initializes a CUDA stream pool, distributed across available GPUs by local rank (`local_rank % device_count`). The multi-architecture binary supports sm_80 (A100), sm_86 (A10), sm_89 (RTX 5000 Ada), and sm_90 (H100) targets simultaneously.

The GPU pipeline has evolved through three phases:

1. **Data Upload**: CPU gathers stencil data → H2D transfer → GPU compute → D2H transfer. H2D dominated at 50–68% of GPU time.
2. **Scatter-Reduce**: GPU-side flux reduction eliminates 98 MB D2H per flush, reducing D2H to 3%. H2D remains bottleneck.
3. **GPU-Gather** (current): Mesh arrays uploaded once to GPU (∼4–9 GB). Only lightweight stencil *indices* (∼7 MB) sent per flush. GPU gathers data from mesh. **H2D reduced to 3%**.

Key Files

- `cuda_stream_pool.{h,cu}` — Thread-safe stream pool, GPU mesh management, buffer allocation
- `hydro_cuda_kernels.cu` — CUDA kernels: `gather`, `gather_ok`, `ctoprim`, `uslope`, `trace3d`, `flux`, `difmag`, `scatter_reduce`
- `hydro_cuda_interface.f90` — Fortran ISO_C_BINDING wrappers with assumed-size `buf(*)` for safe `c_loc` usage
- `cuda_commons.f90` — Fortran module exposing stream pool to the AMR code

17.2 Hybrid CPU/GPU Dispatch

The dispatch model is illustrated for the Godunov solver (`godunov_fine.kjhan.f90`), which accounts for ∼30% of total runtime:

Listing 17.1: Dynamic hybrid dispatch pattern

```
!$omp parallel
  stream_slot = cuda_acquire_stream()  ! atomic, -1 if busy
  !$omp do schedule(dynamic)
  do igrid = 1, ncache, nvector
     if (stream_slot >= 0) then
        ! GPU: gather into superbatch buffer
        ! Flush when buffer full -> async GPU kernel
     else
        ! CPU: standard godfine1 computation
     end if
  end do
  !$omp end do nowait
  ! GPU thread: flush remaining buffer, release stream
!$omp end parallel
```

The `schedule(dynamic)` clause ensures load balancing: if one thread is waiting for GPU synchronization, other threads pick up the remaining iterations on CPU. This is particularly effective when the number of OMP threads exceeds the number of GPU streams.

17.2.1 Superbatch Buffering

GPU kernel launches have significant latency ($\sim$10–50 μs). To amortize this, each GPU thread accumulates grid data into a **superbatch buffer** of size `SUPER_SIZE` (typically 4096 grids) before launching a single kernel covering all accumulated grids. The buffer is flushed when it approaches capacity or when the `omp do` loop completes.

With the GPU-gather path, the superbatch stores **stencil indices** rather than data: each of the $6^3 = 216$ stencil positions per grid stores a cell index into the GPU-resident mesh. Interpolated (coarse) cells are pre-computed on CPU and stored in a separate `interp_vals` buffer, referenced by negative indices. On flush, the GPU pipeline executes 8 kernels: `gather` $\rightarrow$ `gather_ok` $\rightarrow$ `ctoprim` $\rightarrow$ `uslope` $\rightarrow$ `trace3d` $\rightarrow$ `flux` $\rightarrow$ `difmag` $\rightarrow$ `scatter_reduce`.

17.2.2 Accelerated Routines

Six routines use the hybrid dispatch pattern:

Routine	Kernel	Data per cell
`godunov_fine`	5-kernel pipeline	Full stencil ($6^3 \times$ `nvar`)
`synchro_hydro_fine`	Gravity kick	$\rho, \mathbf{p}, E, \mathbf{f}$ (8 doubles)
`courant_fine`	CFL timestep	`nvar` + `ndim` (14 doubles)
`force_fine`	Gradient of ϕ	$3^3 \times \phi + 3^3 \times \mathbf{f}$
`upload_fine`	Prolongation	Child + parent cells
`cooling_fine`	Cooling function	Temperature + density

17.2.3 Memory Management

GPU memory is organized in two tiers:

- **Mesh arrays** (persistent, per-rank): `d_mesh_uold`, `d_mesh_f`, `d_mesh_son` — uploaded once per `godunov_fine` call. Size: `ncell` $\times$ (`nvar` + `ndim`) $\times 8 +$ `ncell` $\times 4$ bytes (e.g., 8.6 GB for 80M cells).
- **Per-stream buffers** (lazily allocated, 2$\times$ over-allocation): stencil indices, compute intermediates (`d_uloc`, `d_gloc`, `d_q`, etc.), and scatter-reduce output.

The host-side `gpu_state_t` stores stencil index arrays and interpolated values, pinned via `cudaHostRegister` for fast DMA. The mesh upload includes a free-memory check: if GPU memory is insufficient, the upload is skipped and all threads fall back to the CPU path automatically.

17.2.4 Per-Thread Scatter Buffer (Lock-Free Level L-1)

The Godunov solver updates conservative variables at both the current level L and the coarser level $L-1$. A key insight is that **Level L writes are conflict-free**: each grid's cell indices $\texttt{ncoarse} + (\texttt{ind\_son} - 1) \times \texttt{ngridmax} + \texttt{ind\_grid}$ are unique per grid, so different OMP threads never touch the same cells. However, **Level $L-1$ writes can conflict**: multiple fine grids may share the same coarse parent cell.

The original code serialized *both* levels with `!$omp critical (godunov)`, destroying all OMP parallelism in the scatter phase. The hybrid dispatch eliminates this lock entirely:

- **Level L**: Written directly to `unew` without any synchronization (conflict-free by construction).
- **Level $L-1$**: Each thread appends pre-summed flux contributions to a private `scatter_buf_t` buffer. After the `!$omp end parallel` barrier, a serial merge applies all buffered entries to `unew`.

The `scatter_buf_t` type (SoA layout) stores per-entry: the coarse cell index, per-variable flux deltas (`dunew`), and optionally divergence/energy corrections (`ddivu`, `denew`) for `pressure_fix`. Buffers start at 65 536 entries and double capacity on demand via `move_alloc`. This approach is exact—no approximation or race condition—and the merge cost is negligible (< 0.01 s in all tests).

17.3 Fortran–CUDA Interface

A critical design choice is the two-layer interface between Fortran and CUDA:

1. **C binding layer** (`_c` suffix): Direct `bind(C)` interface passing `type(c_ptr)` for array arguments.
2. **Fortran wrapper layer** (`_f` suffix): Receives assumed-size `buf(*)` arrays and converts to `c_ptr` via `c_loc(buf(1))`.

The wrapper layer is necessary because Intel ifx may return incorrect values from `c_loc` on assumed-shape arrays. The assumed-size `buf(*)` pattern avoids array descriptors entirely and guarantees that `c_loc` returns the correct data address.

17.4 Multi-GPU Support

When multiple GPUs are available on a node, each MPI rank is assigned to a GPU based on its local rank (`local_rank % device_count`). The stream pool calls `cudaSetDevice(g_device_id)` in `cuda_acquire_stream` to ensure each OMP thread operates on the correct GPU context.

The CUDA binary is compiled with multiple architecture targets:

```
CUDA_ARCH = -gencode arch=compute_80,code=sm_80 \
            -gencode arch=compute_86,code=sm_86 \
            -gencode arch=compute_89,code=sm_89 \
            -gencode arch=compute_90,code=sm_90
```

This enables a single binary to run on A100 (sm_80), A10 (sm_86), RTX 5000 Ada (sm_89), and H100 (sm_90) GPUs without recompilation.

17.5 Scatter-Reduce GPU Kernel

The Godunov GPU pipeline includes an 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 GPU kernel reduces fluxes into compact per-grid output arrays:

- `add_unew(SUPER_SIZE, 8, nvar)`: Level L conservative update per child cell
- `add_lm1(SUPER_SIZE, 6, nvar)`: Level $L-1$ boundary flux per face direction
- `add_divu_l/enew_l`, `add_divu_lm1/enew_lm1`: Pressure-fix divergence and internal energy corrections

The D2H transfer is reduced to ∼5 MB per flush (vs. 98 MB for the raw flux array), a 20× reduction in PCIe bandwidth usage. The host applies Level L results directly to `unew` (conflict-free) and appends Level $L-1$ results to the per-thread scatter buffer for deferred merge.

17.6 GPU-Gather: Mesh-Resident Data

With the scatter-reduce kernel in place, the D2H bottleneck was resolved (3% of GPU time). However, the H2D transfer of stencil data (`uloc`, `gloc`, `ok`) remained the dominant cost at 50–68% of GPU time, because each flush copied ∼100 MB of gathered cell data from host to device.

The **GPU-gather** optimization eliminates this bottleneck by keeping the mesh arrays (`uold`, `f`, `son`) persistently on the GPU and sending only lightweight stencil *indices* per flush.

17.6.1 Mesh Upload

At the start of each `godunov_fine` call, the three mesh arrays are uploaded to GPU device memory:

$$\texttt{d\_mesh\_uold}(\texttt{ncell} \times \texttt{nvar}), \quad \texttt{d\_mesh\_f}(\texttt{ncell} \times \texttt{ndim}), \quad \texttt{d\_mesh\_son}(\texttt{ncell})$$

where $\texttt{ncell} = \texttt{ncoarse} + 8 \times \texttt{ngridmax}$ is the total cell capacity. For a typical 4-rank run with $\texttt{ngridmax} = 10\text{M}$ per rank, the mesh occupies ∼8.6 GB per GPU.

Important: 1 MPI rank per GPU

Each MPI rank uploads its own mesh to GPU memory. Multiple ranks sharing a GPU will exceed GPU memory (e.g., 4×8.6 GB $= 34.4$ GB > 32 GB). The code detects insufficient GPU memory via `cudaMemGetInfo` and automatically falls back to the CPU path.

17.6.2 Stencil Index Encoding

Instead of copying cell data on the CPU, the stencil batch routine stores cell *indices* for each of the $6^3 = 216$ stencil positions per grid:

Index value	Meaning
> 0 (positive)	Direct cell index into `mesh_uold` (1-based Fortran index)
< 0 (negative)	$-$slot into `interp_vals` buffer (interpolated coarse cell)
$= 0$	Boundary/empty cell (data set to zero)

For existing fine grids, the cell index is simply `ncoarse` + (`ind_son` − 1) × `ngridmax` + `igrid_nbor`. For coarse cells without a fine grid, `interpol_hydro` is called on the CPU and the result is stored in the `interp_vals` buffer using AoS layout (`nvar` doubles per slot).

17.6.3 GPU Gather Kernels

Two new kernels precede the existing compute pipeline:

1. `hydro_gather_kernel`: Each thread reads its stencil index, fetches the corresponding cell data from `d_mesh_uold` (or `d_interp_vals` for negative indices), and writes to `d_uloc`/`d_gloc`. Gravity data is fetched from `d_mesh_f` using the gravity stencil index.
2. `hydro_gather_ok_kernel`: Each thread reads the stencil index and checks `son`(cell) > 0 from `d_mesh_son` to determine the refinement flag `ok`.

Both kernels use the same block layout as the compute kernels: one CUDA block per grid, 216 threads per block ($= 6^3$ stencil cells).

17.6.4 H2D Transfer Reduction

Transfer	Data Upload (old)	GPU-Gather (new)
Per-flush H2D	∼100 MB (uloc+gloc+ok)	∼7 MB (stencil indices)
Mesh upload	—	∼4–9 GB (once per godunov call)
D2H	∼5 MB	∼5 MB (unchanged)

The per-flush H2D transfer is reduced by 14×. The one-time mesh upload is amortized across ∼50–300 flushes per `godunov_fine` call.

17.7 Verification and Performance

The GPU-accelerated code produces bit-identical physics results compared to the CPU-only build across all tested configurations. All tests use 10 coarse steps of a 256^3 cosmological simulation with `levelmax=10`.

17.7.1 Physics Verification

All configurations produce bit-identical physics at step 10:

Configuration	GPU	e_{cons}	e_{pot}	e_{kin}
CPU-only (4r×4t)	—	5.23×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
Scatter-reduce (4r×4t, 1 GPU)	A10	5.23×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
Scatter-reduce (4r×4t, 4 GPU)	A10	5.23×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
GPU-gather (4r×4t, 4 GPU)	A10	5.23×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}
GPU-gather (1r×8t, 1 GPU)	RTX 5000 Ada	5.22×10^{-3}	-1.88×10^{-6}	1.23×10^{-6}

The 1-rank test shows slightly different e_{cons} due to different domain decomposition (all grids on a single rank vs. distributed across 4 ranks), but the difference is at roundoff level.

17.7.2 Performance Comparison (A10, 4 ranks × 4 threads)

Configuration	Godunov (s)	Total (s)	Speedup
CPU-only (no GPU)	9.7	79.1	1.00×
Scatter-reduce (1 GPU, shared)	44.7	121.3	0.65×
Scatter-reduce (4 GPU, 1/rank)	46.5	134.0	0.59×
GPU-gather (4 GPU, 1/rank)	**24.8**	**64.6**	**1.22×**

The GPU-gather path achieves a **22% speedup** over CPU-only, reversing the 35–41% *slowdown* of the previous scatter-reduce approach. The Godunov solver itself is still 2.6× slower than CPU-only (24.8 s vs. 9.7 s) because the mesh upload overhead dominates the GPU thread, but the overall run benefits from the hybrid CPU/GPU work distribution.

17.7.3 GPU Kernel Breakdown (GPU-gather, A10, step 10)

	Scatter-reduce (old)		GPU-gather (new)	
Phase	**Time**	**%**	**Time**	**%**
H2D transfer	5.07 s	50.2	0.10 s	3.0
ctoprim	1.10 s	10.9	1.10 s	32.9
uslope	1.19 s	11.8	0.64 s	19.3
trace3d	1.23 s	12.1	0.41 s	12.1
flux+memset	0.70 s	6.9	0.83 s	24.8
difmag	0.43 s	4.2	0.15 s	4.6
D2H transfer	0.38 s	3.8	0.11 s	3.3
Total	10.09 s	100	3.33 s	100

H2D transfer is reduced from 50% to 3% of GPU time. The GPU compute kernels now dominate, indicating that the PCIe transfer bottleneck has been eliminated. Total GPU kernel time is reduced 3× (10.1 s → 3.3 s).

18 CPL Dark Energy

cuRAMSES supports the CPL (Chevallier–Polarski–Linder) parametrization of dark energy [CP01; Lin03], generalizing the cosmological constant Λ to a time-varying equation of state:

$$w(a) = w_0 + w_a\,(1-a), \tag{18.1}$$

where a is the scale factor, w_0 is the present-day value, and w_a governs the time variation. The standard ΛCDM model corresponds to $w_0 = -1$, $w_a = 0$.

18.1 Background Cosmology

The dark energy density evolves as

$$\frac{\rho_{\rm de}(a)}{\rho_{\rm de,0}} \equiv f_{\rm de}(a) = a^{-3(1+w_0+w_a)}\,\exp\!\big[-3\,w_a\,(1-a)\big]. \tag{18.2}$$

For ΛCDM ($w_0 = -1$, $w_a = 0$), $f_{\rm de}(a) = 1$ identically, recovering the constant dark energy density.

The Friedmann equation becomes

$$\frac{H^2(a)}{H_0^2} = \frac{\Omega_m}{a^3} + \frac{\Omega_k}{a^2} + \Omega_{\rm de}\,f_{\rm de}(a), \tag{18.3}$$

where $\Omega_k = 1 - \Omega_m - \Omega_{\rm de}$.

18.2 Modified Functions

The following functions in `init_time.f90` are modified:

- `f_de(a, w0_in, wa_in)`: New standalone function implementing Eq. (18.2). Uses exact $f_{\rm de} = 1$ branch for ΛCDM to guarantee bit-identical results.
- `dadtau(axp_tau, ..., w0_in, wa_in)`: Conformal time derivative $da/d\tau$. The dark energy term $\Omega_{\rm de}\,a^6$ is replaced by $\Omega_{\rm de}\,a^6\,f_{\rm de}(a)$:

$$\left(\frac{da}{d\tau}\right)^2 = a^3\left[\Omega_m + \Omega_{\rm de}\,a^3\,f_{\rm de}(a) + \Omega_k\,a\right]. \tag{18.4}$$

- `dadt(axp_t, ..., w0_in, wa_in)`: Proper time derivative da/dt, with the same CPL modification.
- `friedman(..., w0_in, wa_in)`: Friedmann integrator that builds the look-up tables $a(\tau)$, $H(\tau)$, $t(\tau)$. The extra w_0, w_a parameters are passed through to `dadtau`/`dadt`.

18.3 Linear Growth Factor

The linear growing mode $D_1(a)$ and the growth rate $f = d\ln D_1/d\ln a$ (Peebles [Pee80]) are generalized using

$$y(a) \equiv (aH/H_0)^2 = \frac{\Omega_m}{a} + \Omega_k + \Omega_{\rm de}\, f_{\rm de}(a)\, a^2. \tag{18.5}$$

The growth factor is computed via Romberg integration:

$$D_1(a) = \frac{\sqrt{y(a)}}{a} \int_0^a \frac{da'}{y(a')^{3/2}},$$

and the growth rate requires the logarithmic derivative:

$$\frac{dy}{d\ln a} = -\frac{\Omega_m}{a} + \Omega_{\rm de}\, a^2\, f_{\rm de}(a)\,\big[-1 - 3\,w(a)\big], \tag{18.6}$$

so that

$$f(a) = \frac{1}{2}\frac{dy/d\ln a}{y} - 1 + \frac{a}{y^{3/2}\int_0^a da'/y^{3/2}}.$$

These are implemented in the `contains` functions of `init_cosmo`: `fy(a)`, `d1a(a)`, and `fpeebl(a)`.

ΛCDM verification. For $w_0 = -1$, $w_a = 0$: $f_{\rm de} = 1$, $w(a) = -1$, and $-1 - 3w = -1 - 3(-1) = 2$, giving

$$\left.\frac{dy}{d\ln a}\right|_{\Lambda\rm CDM} = -\frac{\Omega_m}{a} + 2\,\Omega_\Lambda\, a^2,$$

which recovers the original RAMSES expression $(\Omega_\Lambda a^2 - \frac{1}{2}\Omega_m/a)/y$.

18.4 Hubble Rate in Cooling Module

The function `HsurH0(z)` in `cooling_module.f90` is modified to include $f_{\rm de}$:

$$H(z)/H_0 = \sqrt{\Omega_m\,(1+z)^3 + \Omega_R\,(1+z)^2 + \Omega_{\rm de}\, f_{\rm de}(a)}, \qquad a = \frac{1}{1+z}. \tag{18.7}$$

This ensures that cooling rates computed from the UV background evolve consistently with the CPL cosmology.

18.5 Namelist Configuration

CPL parameters are specified in the `&COSMO_PARAMS` namelist block:

CPL dark energy example

```
&COSMO_PARAMS
w0     = -0.9
wa     =  0.1
/
```

When this block is absent (or when $w_0 = -1$, $w_a = 0$), the code produces results that are bit-identical to the original ΛCDM implementation.

18.6 Modified Files

File	Changes
`amr_parameters.jaehyun.f90`	Declare `w0`, `wa`, `cs2_de`
`read_params.jaehyun.f90`	Define and read `&COSMO_PARAMS` namelist
`init_time.f90`	Add `f_de`; modify `dadtau`, `dadt`, `friedman`, `fy`, `d1a`, `fpeebl`, `vfact`
`cooling_module.f90`	Modify `HsurH0` to use `f_de`

II

Namelist Reference

19 RUN_PARAMS

Runtime control parameters.

Parameter	Type	Default	Description
`cosmo`	logical	`.false.`	Enable cosmological simulation
`pic`	logical	`.false.`	Enable Particle-In-Cell
`poisson`	logical	`.false.`	Enable Poisson gravity solver
`hydro`	logical	`.false.`	Enable hydrodynamics
`rt`	logical	`.false.`	Enable radiative transfer
`sink`	logical	`.false.`	Enable sink particles
`verbose`	logical	`.false.`	Verbose output
`debug`	logical	`.false.`	Debug mode
`nrestart`	integer	0	Restart file number (0 = new run)
`nstepmax`	integer	1000000	Maximum number of coarse steps
`ncontrol`	integer	1	Frequency of control variable output
`nsubcycle`	integer[]	2	Subcycling factor per level
`nremap`	integer	**5**	Load balancing frequency (0=never)
`ordering`	char	`hilbert`	Domain decomposition: `hilbert`, `bisection`, `ksection`
`static`	logical	`.false.`	Static (no refinement) mode
`overload`	integer	1	MPI overload factor
`cost_weighting`	logical	`.true.`	CPU time-based cost weighting
Memory-based load balancing (new)			
`memory_balance`	logical	`.false.`	Enable memory-weighted balancing
`mem_weight_grid`	integer	270	Memory per grid (dp-equivalents)
`mem_weight_part`	integer	12	Memory per particle (dp-equivalents)
Job control (new)			
`jobcontrolfile`	char(128)	’’	Runtime control file for stop/output requests

Job control file. When `jobcontrolfile` is set to a non-empty path, rank 0 reads the file at every coarse step. Each line contains two integers: `step_number` and `action_code`.

- `step_number = 0`: match at every step (immediate).
- `step_number = N`: match when `nstep_coarse` $= N$.
- `action = 1`: write an extra output and continue.
- `action = -1`: write an extra output and stop gracefully.

Example: to request a graceful stop at the next coarse step, create the file with `echo "0 -1" > jobcontrol.txt`. The file is not deleted after reading; lines with `step_number = 0` re-trigger every step until the file is removed or modified.

20 AMR_PARAMS

Adaptive Mesh Refinement grid parameters.

Parameter	Type	Default	Description
`levelmin`	integer	1	Minimum (uniform) refinement level
`levelmax`	integer	1	Maximum refinement level
`ngridmax`	integer8	0	Max grids per CPU (0 = auto)
`ngridtot`	integer8	0	Total grids for auto-computation
`npartmax`	integer	0	Max particles per CPU (0 = auto)
`nparttot`	integer	0	Total particles for auto-computation
`nexpand`	integer[]	1	Mesh expansion layers per level
`boxlen`	real(dp)	1.0	Box side length in code units

21 OUTPUT_PARAMS

Output control parameters.

Parameter	Type	Default	Description
`noutput`	integer	1	Number of scheduled outputs
`foutput`	integer	1000000	Output every *N* steps
`fbackup`	integer	1000000	Backup every *N* steps
`aout`	real[]	1.1	Output expansion factors (cosmo)
`tout`	real[]	0.0	Output times (non-cosmo)
`outformat`	char(10)	`original`	Output format: `original` (per-CPU binary) or `hdf5` (single HDF5 file)
`informat`	char(10)	`original`	Restart format: `original` or `hdf5`. Can differ from `outformat`
`output_mode`	integer	0	Hi-res output mode
`gadget_output`	logical	`.false.`	Write Gadget-format snapshots
`walltime_hrs`	real(dp)	-1	Job walltime (hours, < 0 = ignore)
`minutes_dump`	real(dp)	1.0	Dump this many minutes before walltime

22 INIT_PARAMS

Initial conditions parameters.

Parameter	Type	Default	Description
`filetype`	char(20)	`ascii`	IC file format: `ascii`, `grafic`, `gadget`
`initfile`	char[]	`' '`	IC file path per level
`multiple`	logical	`.false.`	Multiple IC files per rank
`nregion`	integer	0	Number of IC regions

23 REFINE_PARAMS

Refinement criteria parameters.

Parameter	Type	Default	Description
`m_refine`	real[]	-1	Lagrangian mass threshold per level
`ivar_refine`	integer	-1	Variable index for gradient refinement
`var_cut_refine`	real(dp)	-1	Variable threshold
`mass_cut_refine`	real(dp)	-1	Particle mass threshold
`interpol_var`	integer	0	Interpolation variable (0=conservative, 1=primitive)
`interpol_type`	integer	1	Interpolation type (0=MinMod, 1=MonCen)
`sink_refine`	logical	`.false.`	Fully refine around sinks
`jeans_ncells`	real(dp)	-1	Jeans length in cells (>0 enables polytropic EOS)

24 HYDRO_PARAMS

Hydrodynamics solver parameters.

Parameter	Type	Default	Description
`gamma`	real(dp)	$\frac{5}{3}$	Adiabatic index γ
`courant_factor`	real(dp)	0.8	Courant–Friedrichs–Lewy number
`scheme`	char(20)	`muscl`	Hydro scheme (`muscl`)
`slope_type`	integer	1	Slope limiter (1=MinMod, 2=MonCen, 3=unlimited)
`pressure_fix`	logical	`.false.`	Pressure floor for strong shocks
`beta_fix`	real(dp)	0.0	Pressure fix strength
`isothermal`	logical	`.false.`	Isothermal mode

25 POISSON_PARAMS

Gravity and Poisson solver parameters.

Parameter	Type	Default	Description
`epsilon`	real(dp)	10^{-4}	Multigrid convergence criterion
`gravity_type`	integer	0	Gravity type (0=self-gravity, >0=analytic)
`cg_levelmin`	integer	999	Min level for CG solver fallback
`cic_levelmax`	integer	0	Max level for CIC particle interpolation

26 PHYSICS_PARAMS

Subgrid physics parameters (star formation, feedback, cooling).

Parameter	Type	Default	Description
`cooling`	logical	`.false.`	Enable radiative cooling
`metal`	logical	`.false.`	Enable metal tracking
`haardt_madau`	logical	`.false.`	UV background
`z_reion`	real(dp)	8.5	Reionization redshift
Star formation			
`n_star`	real(dp)	0.1	SF density threshold (H/cc)
`t_star`	real(dp)	0.0	SF timescale (Gyr)
`eps_star`	real(dp)	0.0	SF efficiency
`T2_star`	real(dp)	0.0	ISM polytropic temperature
`g_star`	real(dp)	1.6	ISM polytropic index
`sf_birth_properties`	logical	`.true.`	Output stellar birth properties
Cosmology (read from IC header or namelist)			
`omega_b`	real(dp)	0.0	Baryon density Ω_b
`omega_m`	real(dp)	1.0	Matter density Ω_m
`omega_l`	real(dp)	0.0	Dark energy Ω_Λ
`h0`	real(dp)	1.0	Hubble constant H_0 (km/s/Mpc)
`w0`	real(dp)	-1	CPL DE equation of state w_0 (see Ch. 18)
`wa`	real(dp)	0	CPL DE time variation w_a
`cs2_de`	real(dp)	0	DE sound speed squared (reserved)
Feedback			
`f_ek`	real(dp)	1.0	SN kinetic energy fraction
`rbubble`	real(dp)	0.0	SN superbubble radius (pc)
`yieldtable-filename`	char	—	Yield table file path

27 COSMO_PARAMS

Optional block for CPL dark energy equation-of-state parameters and cosmological parameter overrides. When absent, defaults recover ΛCDM. See Chapter 18 for the full mathematical description.

Parameter	Type	Default	Description
`omega_b`	real(dp)	0.0	Baryon density Ω_b
`omega_m`	real(dp)	1.0	Matter density Ω_m
`omega_l`	real(dp)	0.0	Dark energy density $\Omega_{\rm de}$
`h0`	real(dp)	1.0	Hubble constant H_0 (km/s/Mpc)
CPL dark energy (new)			
`w0`	real(dp)	-1	Present-day DE EoS: $w(a) = w_0 + w_a(1-a)$
`wa`	real(dp)	0	Time variation of DE EoS
`cs2_de`	real(dp)	0	DE sound speed squared c_s^2 (reserved for future use)

Notes:

- w_0, w_a are not stored in the GRAFIC2 IC header (fixed 44-byte layout). They must be specified here when using CPL dark energy.
- Setting $w_0 = -1$, $w_a = 0$ (or omitting the block) produces results **bit-identical** to the original ΛCDM code.
- The cosmological parameters (`omega_b`, `omega_m`, `omega_l`, `h0`) in this block override values read from the IC header.

28 Example Namelist

28.1 Cosmological Simulation with Memory Balancing

cosmo_ksection_membal.nml

```
&RUN_PARAMS
cosmo=.true.
pic=.true.
poisson=.true.
hydro=.true.
nrestart=0
nremap=5
nsubcycle=1,1,2
ncontrol=1
nstepmax=10
ordering='ksection'
memory_balance=.true.
jobcontrolfile='jobcontrol.txt'
/

&OUTPUT_PARAMS
noutput=1
aout=1.0
/

&INIT_PARAMS
filetype='grafic'
initfile(1)='/path/to/ics/level_008'
/

&AMR_PARAMS
levelmin=8
levelmax=10
nexpand=1
ngridtot=40000000
nparttot=200000000
/

&REFINE_PARAMS
m_refine=3*8.,
ivar_refine=0
interpol_var=1
interpol_type=0
/
```

```
&HYDRO_PARAMS
gamma=1.6666667
courant_factor=0.8
scheme='muscl'
slope_type=1
/

&POISSON_PARAMS
/

&PHYSICS_PARAMS
sf_birth_properties=.false.
yieldtablefilename='yield_table.asc'
/
```

Build and Testing

29 Build Instructions

29.1 Prerequisites

- Intel Fortran Compiler (`ifx`) with MPI support (`mpiifx`)
- OpenMP support (`-qopenmp`)

29.2 Build

Build Commands

```
cd bin
make clean
make
```

The binary is produced as `bin/ramses_final3d`.

29.3 Compile-Time Options

Key preprocessor flags set in the Makefile:

Flag	Meaning
`-DNDIM=3`	Three-dimensional simulation
`-DNVECTOR=32`	Vector length for grid sweeps
`-DLONGINT`	64-bit integer grid indices
`-DQUADHILBERT`	Quad-precision Hilbert keys
`-DNVAR=11`	Number of hydro variables
`-DNPRE=8`	Passive scalar variables

30 Verification Tests

30.1 Test Configuration

Setting	Value
IC	MUSIC `level_008` (GRAFIC2 format)
MPI ranks	12
Levels	8–10
Steps	10
Ordering	ksection

30.2 Reference Values (nremap=5)

Test	nparttot	econs	epot	ekin	mcons
membal	200M	3.77E-03	−1.88E-06	1.23E-06	−1.84E-16
nomembal	80M	3.77E-03	−1.88E-06	1.23E-06	−1.84E-16

30.3 Running Tests

Membal Test

```
cd test_ksection/run_cosmo_membal
cp ../../bin/ramses_final3d .
mpirun -np 12 ./ramses_final3d \
    ../cosmo_ksection_membal.nml
```

Verify that at step 10: `econs=3.77E-03`, `epot=-1.88E-06`, `ekin=1.23E-06`.

A Modified Files Summary

File	Modifications
patch/cuda/	
`amr_parameters.jaehyun.f90`	Memory balance params, nremap=5 default
`amr_commons.kjhan.f90`	grid_level array, communicator type
`read_params.jaehyun.f90`	Read new namelist params
`bisection.f90`	nc_in parameter, 64-bit histograms
`ksection.f90`	K-section tree + exchange routines
`load_balance.kjhan.f90`	numbp sync, bulk exchange, internal timing
`virtual_boundaries.kjhan.f90`	Ksec ghost exchange + bulk + build_comm
`multigrid_fine_commons.f90`	MG ksection communication
`init_amr.f90`	Memory savings, Morton init
`update_time.f90`	Memory reporting
`adaptive_loop.jaehyun.f90`	writemem_minmax, Morton rebuild, bulk exchange
patch/oct_tree/	
`morton_keys.f90`	Morton key computation
`morton_hash.f90`	Hash table + neighbor helpers
`morton_init.f90`	Build and verify
`refine_utils.f90`	Hash maintenance in grid ops
`nbors_utils.kjhan.f90`	Morton-based neighbor lookup
patch/Horizon5-master-2/	
`init_flow_fine.f90`	Stream access IC reading
`init_part.f90`	Stream access particle IC
`particle_tree.kjhan.f90`	MPI_ALLTOALL → ksection
`amr_step.jaehyun.f90`	Bulk virtual exchange
`feedback.kjhan3.f90`	SNII feedback spatial binning, ksection exchange, ALLREDUCE packing
`sink_particle.kjhan.f90`	Sink particle ops, AGN spatial binning, ALLREDUCE packing, ksection reduce-sum
patch/cuda/ (HDF5 I/O — new files)	
`ramses_hdf5_io.f90`	HDF5 wrapper module (parallel create/open/write/read)

File	Modifications
`backup_hdf5.f90`	HDF5 output: AMR, hydro, gravity, particles, sinks
`restore_hdf5.f90`	HDF5 restart: AMR tree rebuild + data restore
`output_amr.kjhan.f90`	HDF5 dispatch in `dump_all`