I/O Analysis and Optimization for an AMR Cosmology Simulation

Jianwei Li Wei-keng Liao

Alok Choudhary Valerie Taylor

ECE Department

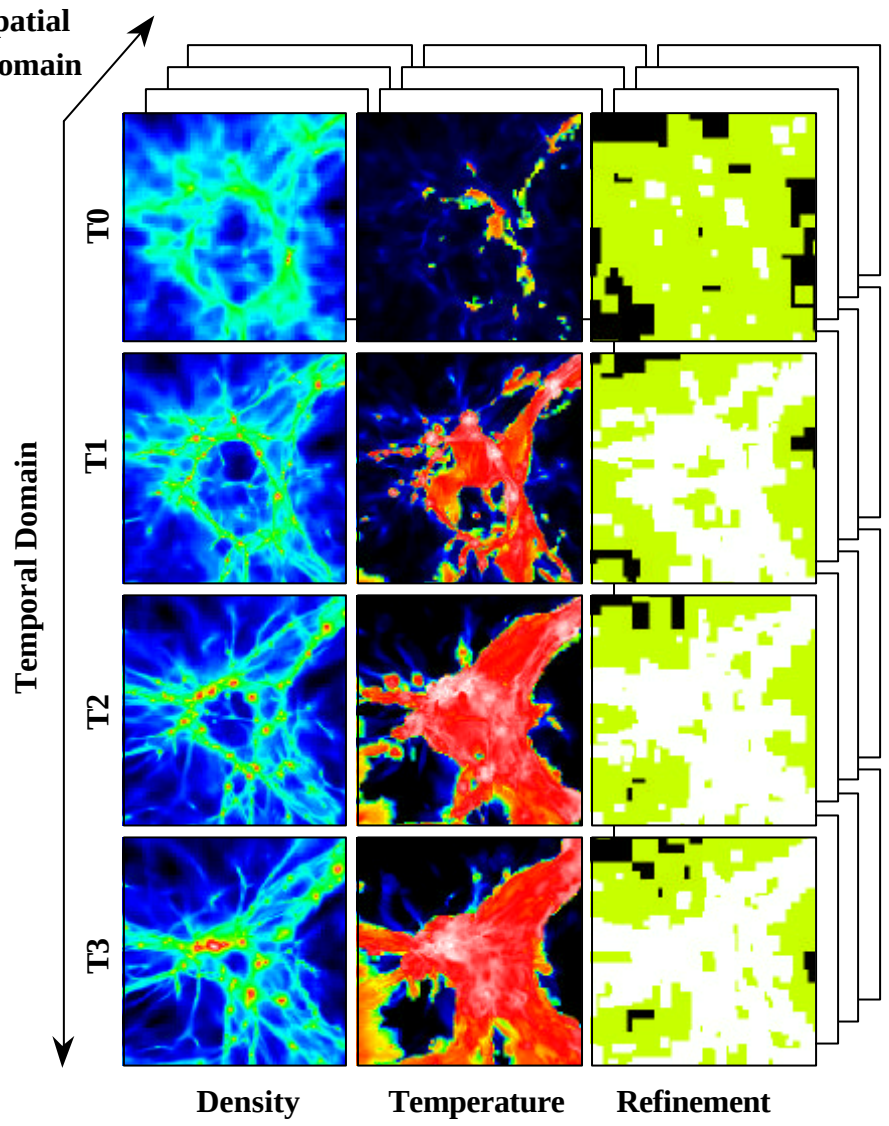
Northwestern University

ENZO Background

- Simulate the formation of a cluster of galaxies (gas and starts) starting near the big bang until the present day
- Used to test theories of how galaxy forms by comparing the results with what is really observed in the sky today

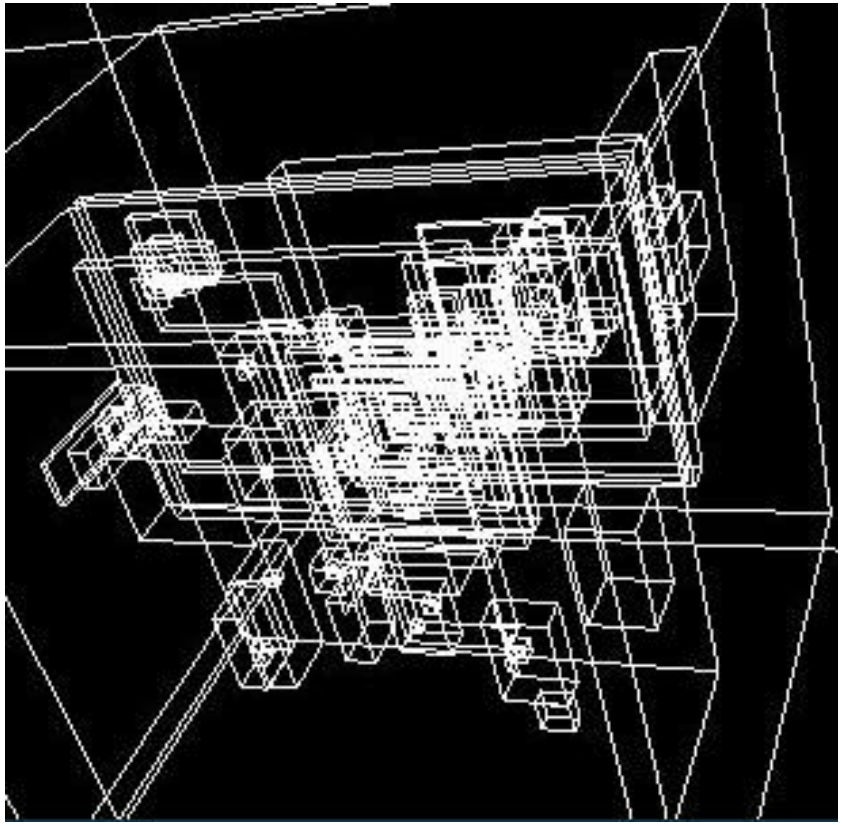
Purpose
Implementation
Visualize
Datasets

- Highly irregular spatial distribution of cosmic objects
- Algorithm: Adaptive Mesh Refinement (AMR)
- Parallelism achieved by domain decomposition of 3-D grids
- Dynamic load balance using MPI



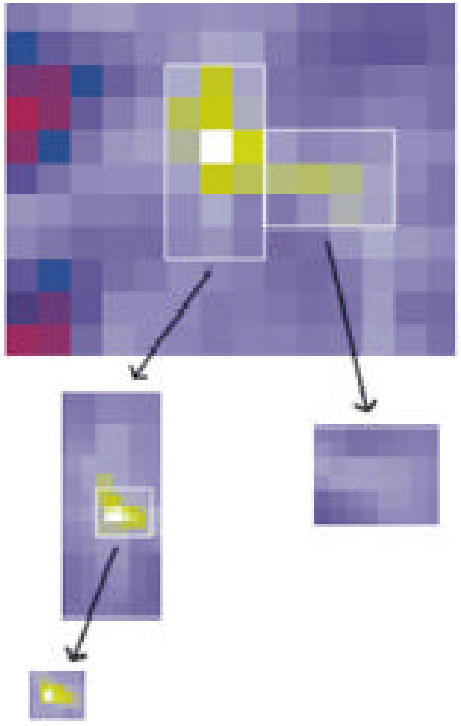
AMR Algorithm

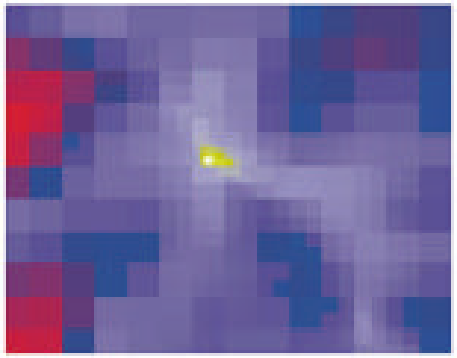
- Multi-scale algorithm that achieves high spatial resolution in localized regions
- Recursively produces a deep, dynamic hierarchy of increasingly refined grid patches



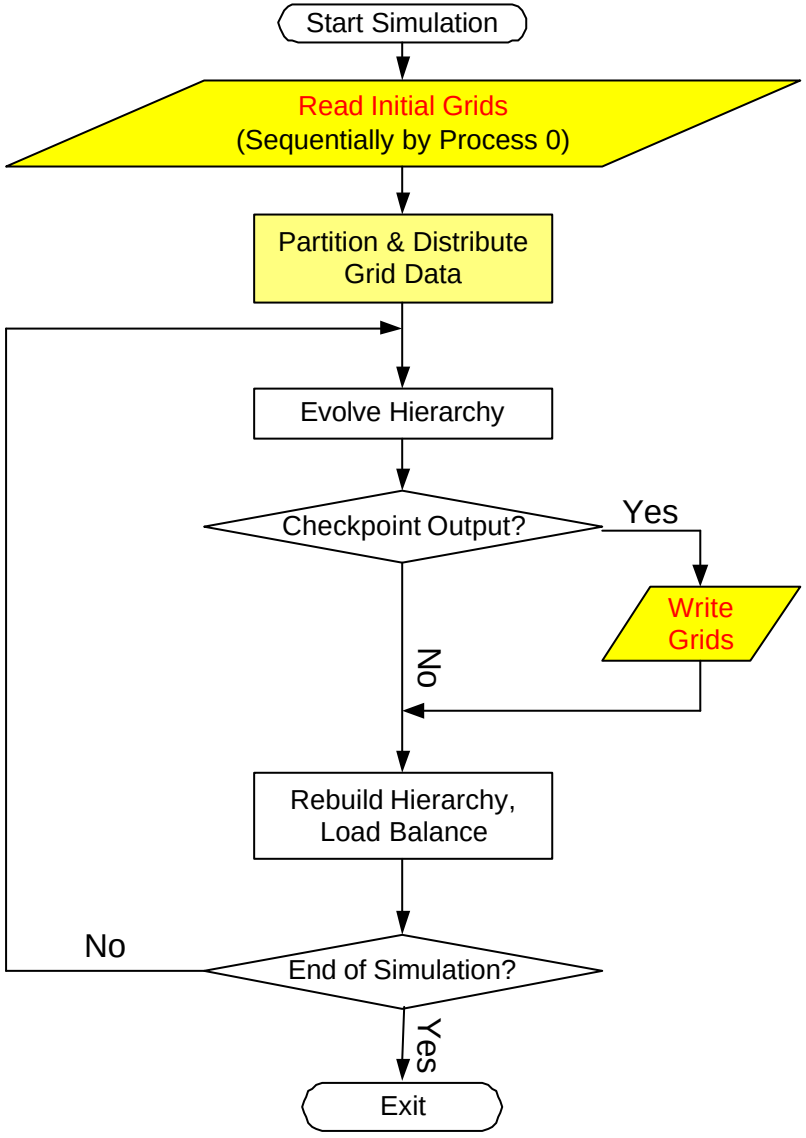
hierarchical AMR data sets as bounding boxes

Refinement Hierarchy





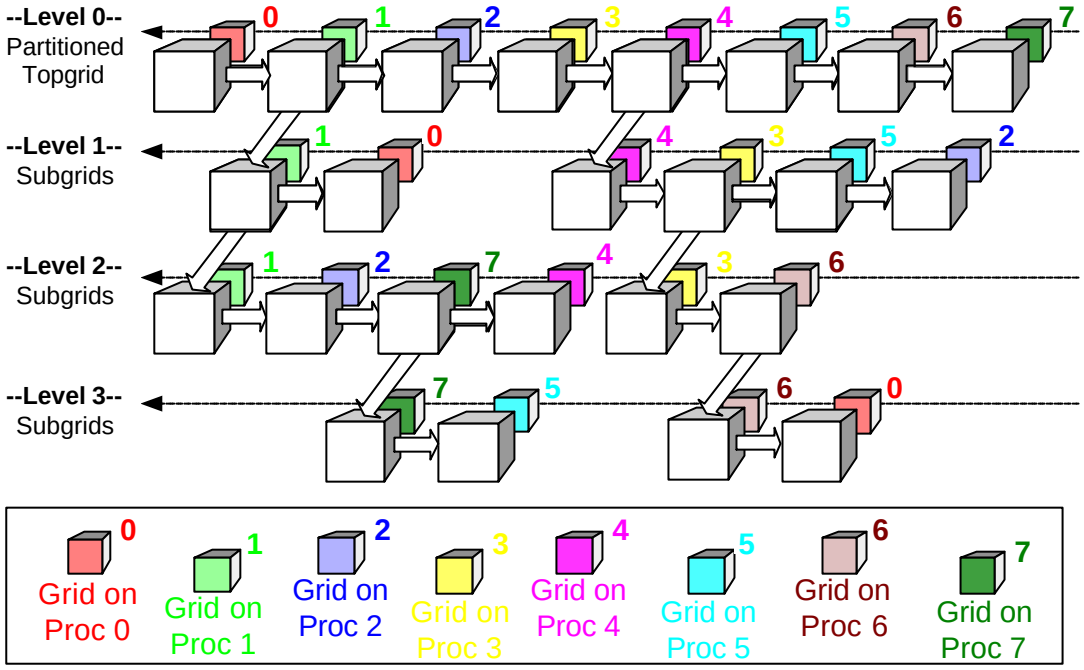
Execution Flow



- Read root grid and some initial pre-refined sub grids
- Partition the grids data among multiple processors
- Main loop of evolution over time:
  - Solve hydro-equations to advance the solution by *dt* on each grid
  - Recursively evolve the grid hierarchy on each level using AMR control algorithm
  - Check-pointing to write out grids data (hierarchical simulation results) at current time stamp
  - Adaptively refine the grids and rebuild the new finer hierarchy
  - Redistribute grids data to perform load balance

Parallelization

- Top grid is partitioned into multiple grids among all processors
- Sub grids are distributed among all processors
- Some sub grids can even be partitioned and redistributed in load balance



- A hierarchical data structure, containing grids metadata and gird hierarchy information, is maintained on all processors
- Each of the hierarchy nodes points to a real grid that resides only on the allocated processor
- A grid is owned by only one processor but one processor can have many grids.
- Each processor perform computation on its own grids.

ENZO Datasets

There are three kinds of datasets that are involved in ENZO I/O operations:

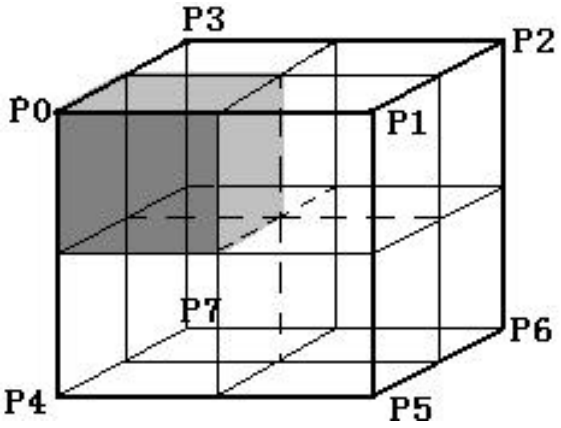
Baryon Fields, Particle Datasets contained in a grid and Boundary data

- Baryon Fields (Gas)
  - A number of 3-D float type arrays
    - Density Field
    - Energy Fields
    - Velocity [X, Y, Z] Fields
    - Temperature Field,
    - Dark Matter Field
    - ...

- Particle Datasets (Stars)
  - A number of 1-D float type arrays and 1 integer array
    - Particle ID
    - Particle Position [X,Y, Z]
    - Particle Velocity [X, Y, Z]
    - Particle Mass
    - Particle Attributes
    - …

- Boundary data (Boundary Types and Values on each boundary side of each dimension for each Baryon Field)
  - a number of two-dimensional float type arrays.

Data Partition



(Block, Block, Block)

- Partition of Baryon Field Datasets

- Partition of Particle Datasets

The 1-D particle data arrays are partitioned based on which grid sub-domain the particle position falls within, so the pattern is totally Irregular.

- Boundary data are maintained on all processors and not partitioned

I/O Patterns

- Real data
  - Read from initialized physical variables of starting grids
  - Periodically write out physical solutions of all grids. During the simulation loop, all grids will be dump to files in a timely sequence (check pointing)
  - The datasets are read/written in a fixed, pre-defined sequence order
  - Even the access to the grid hierarchy follows a certain sequence
- Metadata
  - Metadata of grid hierarchy is written out recursively
  - Metadata of each dataset is written out together with grid real data
- Visualizing data
  - In ENZO, there's no separate visualizing data directly written out by the simulation
  - Visualization is performed by another program taking the check pointing grid data (real and meta data) as input and doing projection

I/O Approach

- Original Approach:
  - Real data is read/written as one grid per file by the allocated processor
  - The read of initial grids is done by one processor and then partitioned among all processors
  - For the partitioned top grid, the data is first combined from other processor, and then written out by the root processor
  - Grid hierarchy metadata is written out in text files by root processor
  - Grid dataset metadata is written out to the same grid file
- Other choices:
  - Real data to be stored in one single file with parallel I/O access
  - Generate visualizing data directly without re-read and processing of the check pointing grid files

  ✍ Advantages:
    - Parallel I/O largely improves the I/O performance
    - Storing all real data in one single file makes pre-fetching easier
    - Visualization is real time. Re-reading a large number of distributed grid files and processing them is very time consuming.

  ✍ Disadvantages: Managing the metadata needs a lot more work, parallel I/O not so easy

Sequential HDF4 I/O

- This is the original I/O implementation
  - Each grid (real data and meta data) is read/written sequentially by its allocated processor independently using HDF4 I/O library
  - The grid hierarchy metadata is written to separate ASCII file using native I/O library

✍ Advantages:

  - HDF4 provides self-describing data format with metadata stored together with the real data in the same file

✍ Disadvantages:

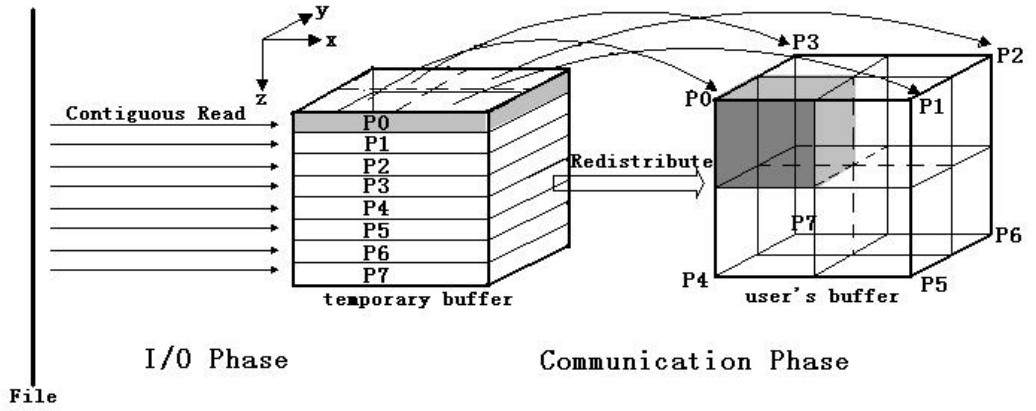
  - Does not provide parallel I/O facilities – low performance
  - Can not combine datasets into file, or have to spend extra time to explicitly combine datasets in memory and then write to file
  - Storing the metadata with real data bring some overhead and makes access of real data inefficient.

Native I/O

- Advantage
  - Flexible, apply any parallel I/O techniques at application level
  - Performance can be potentially very good
- Disadvantages:
  - Implementation will be trivial, user have to handle a lot of tasks
  - Hard for programmer to manage the metadata
  - Lots of work for performance
  - Platform dependent

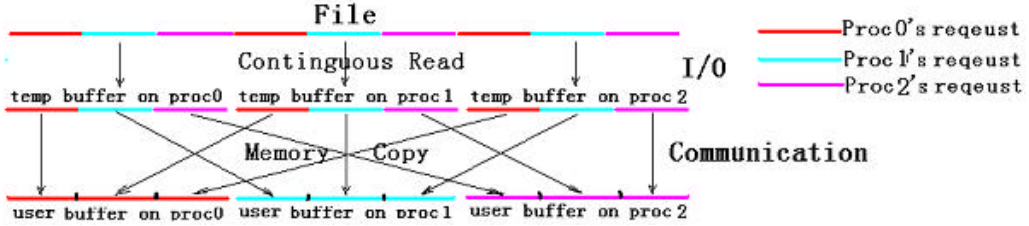
Parallel I/O using MPI-IO

- Advantages
  - Parallel I/O access
  - Collective I/O
  - Easy to implement application level two-phase I/O
  - Expecting high performance
  - Easy to combine grids into a single file
  - Also easy to directly write visualizing data



▲ **Collective read of a Baryon Dataset with two-phase I/O**

▼ **Application level two-phase I/O for a particle dataset**

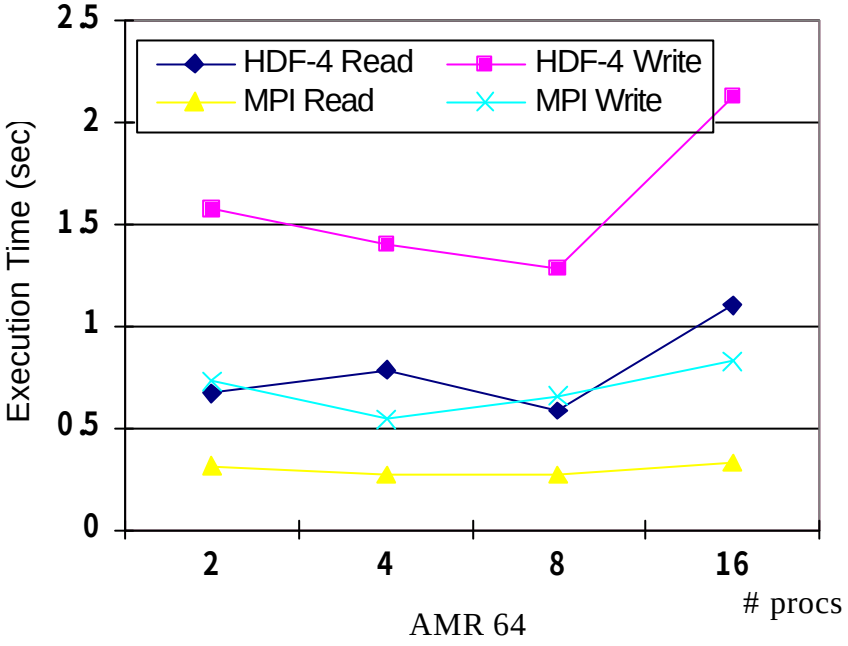


- Disadvantages
  - Only beneficial for raw data. Metadata is usually small and reading or writing metadata using MPI-IO can make performance only worse.
  - Need more implementation to manage metadata separately
- One Possible solution: Using XML to manage the hierarchical metadata

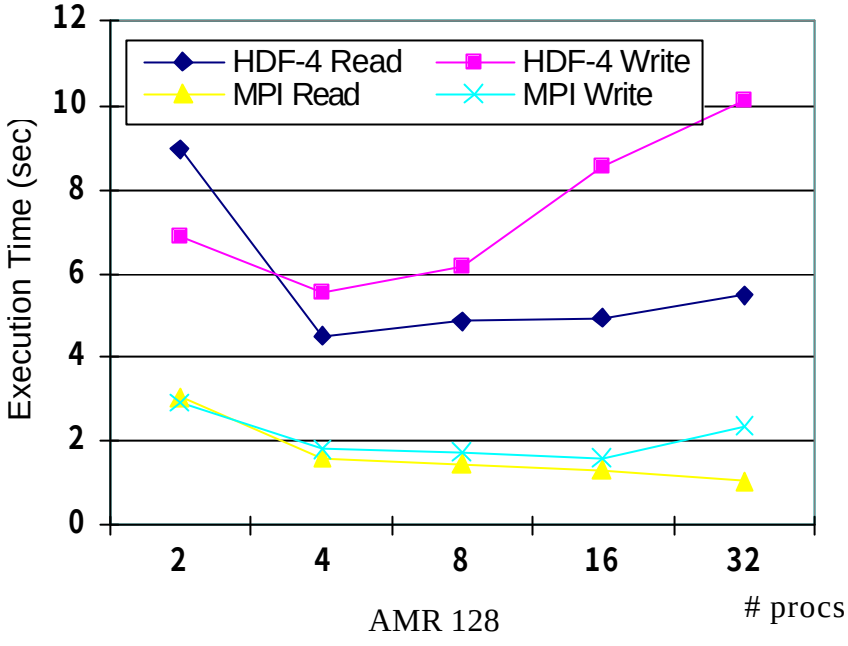
Parallel I/O using HDF5

- Advantages
  - Self-describing data format, easy to manage metadata
  - Parallel I/O access on top of MPI-IO
  - Groups and Datasets organized in a hierarchical/tree structure, easy to combine grids into a single file
- Disadvantages
  - Implementation overhead. Packing and unpacking hyperslabs are currently handled recursively, taking a relatively long time.
  - Some features are not yet completed. Creating and closing datasets are collective which produces additional synchronization in parallel I/O access. Adding/changing attributes can only be done on processor 0, which also limits the parallel performance on writing real data.
  - Mixing the metadata with real data makes the real data not aligned on appropriate boundaries, which results in a high variance in access time between processes

Performance Evaluation on Origin2000



AMR 64



AMR 128

- I/O Performance of the ENZO application on SGI Origin2000 with XFS

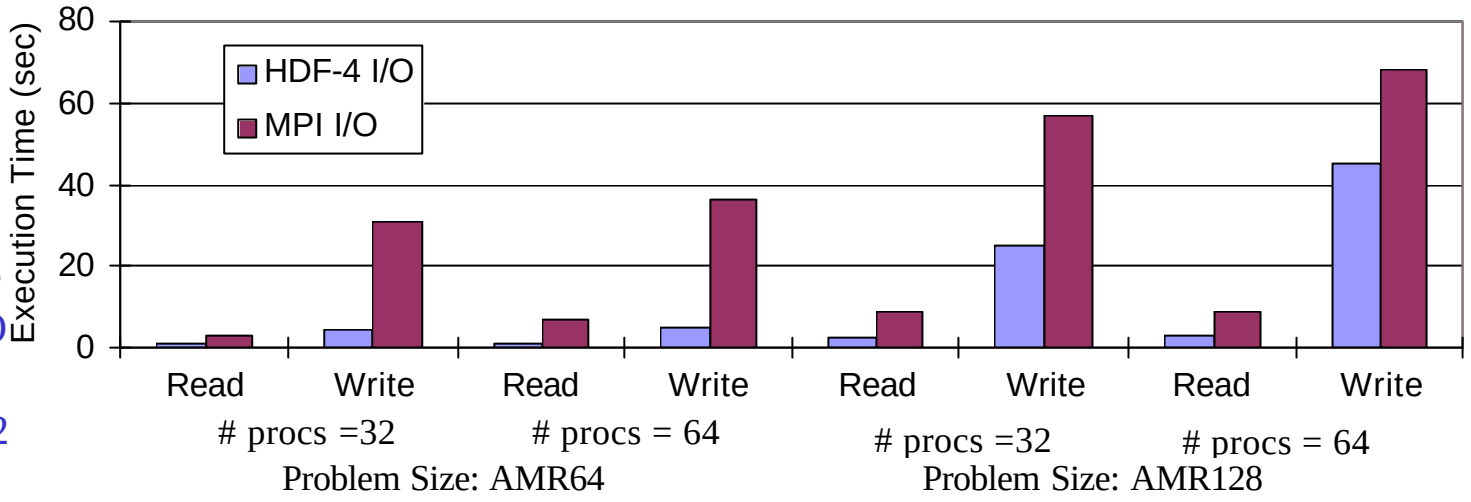
- The amount of data read/written by ENZO Cosmology Simulation with different problem sizes

	AMR64	AMR128	AMR256
Read	2.78 MB	22.15 MB	177.21 MB
Write	13.12 MB	66.42 MB	525.27 MB

Result: significant I/O performance improvement of MPI-IO over HDF4 I/O

Performance Evaluation on IBM SP (Using GPFS)

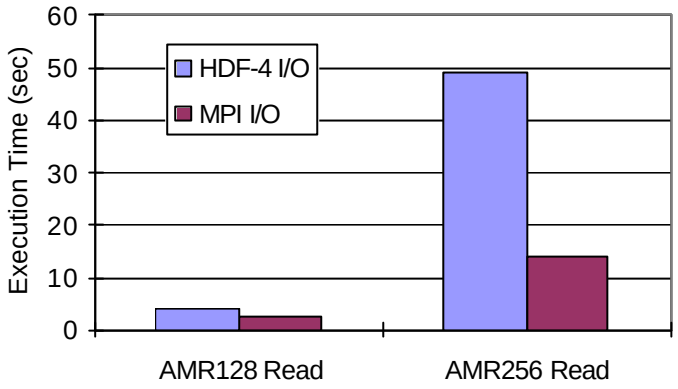
I/O Performance of the ENZO application on IBM SP-2 with GPFS

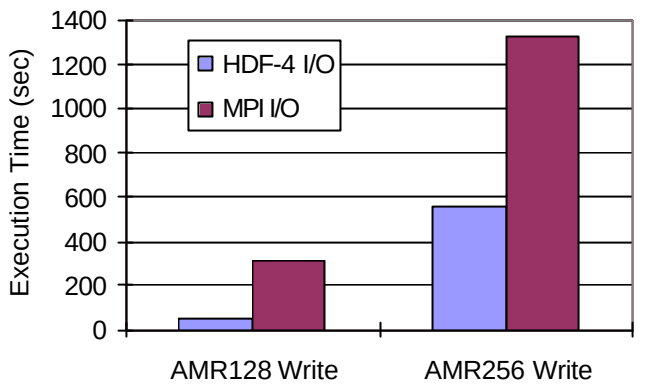


The performance of our parallel I/O using MPI-IO is worse than that of the original HDF4 I/O. This happens because the data access pattern in this application does not fit in well with the disk file striping and distribution pattern in the parallel file system. Each process may access small chunks of data while the physical distribution of the file on the disks is based on very large striping size, the chunks of data requested by one process may span on multiple I/O nodes, or multiple processes may try to access the data on a single I/O node.

Performance Evaluation on Linux Cluster (Using PVFS)

I/O Performance of the ENZO application on Linux cluster with PVFS (8 compute nodes and 8 I/O nodes)

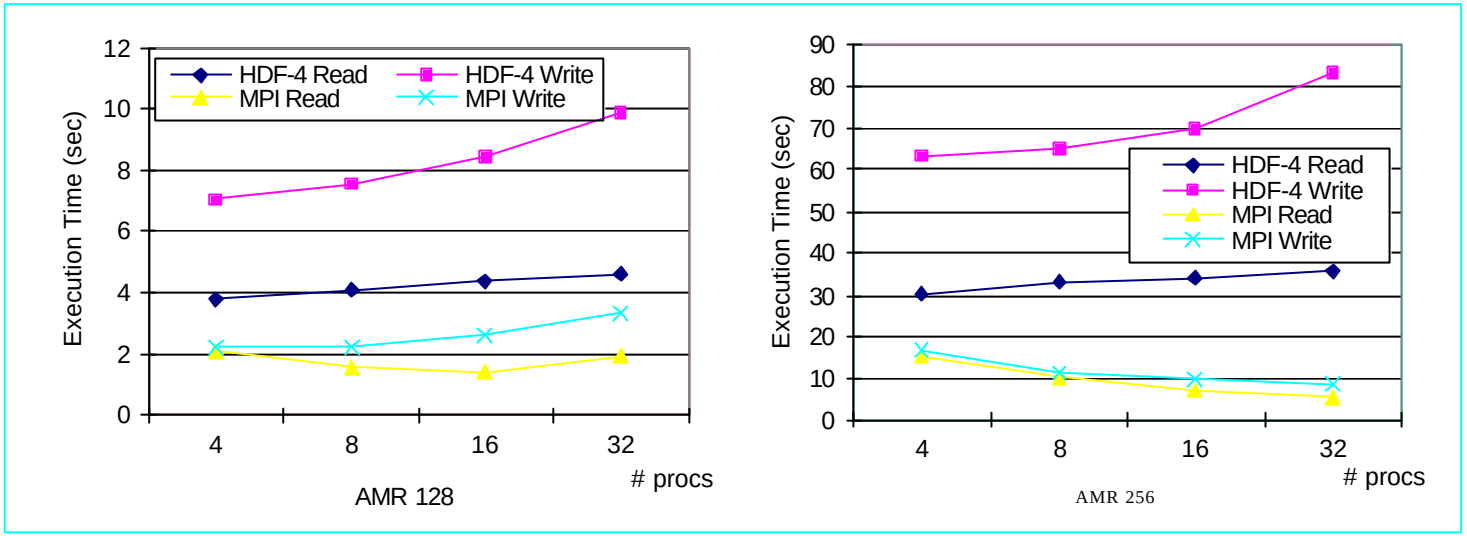




Like GPFS, the PVFS for MPI-IO uses fixed striping scheme specified by the striping parameters at setup time and the physical data partition pattern is also fixed, hence not tailored for specific parallel I/O applications. More importantly, the striping and partition patterns are uniform across multiple I/O nodes, which is good for efficient utilization of disk space but not flexible hence not good enough for performance. For various types of access patterns, especially those in which each process accesses a large number of stridden, small data chunks, there may be significant skew between application access patterns and physical file partition/distribution patterns. So the communication (between compute nodes and I/O nodes) overhead of using parallel I/O may be very large.

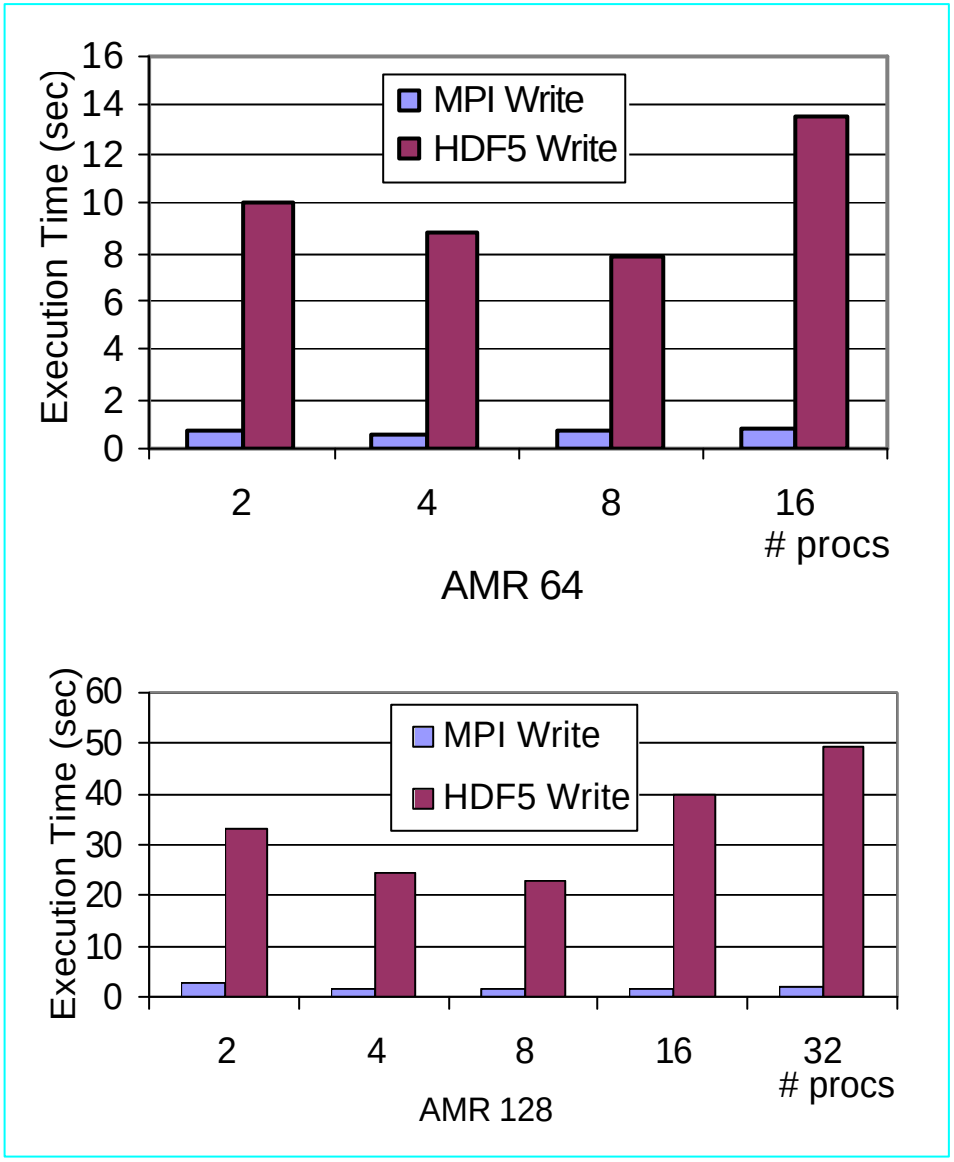
Performance Evaluation on Linux Cluster (Using Local Disk)

I/O Performance of the ENZO application on Linux cluster with each compute node accessing its local disk using PVFS interface



The I/O operation of each compute node is performed on its local disk. The only overhead of MPI-IO is the user-level inter-communication among compute nodes. As expected, the MPI-IO has much better overall performance than the HDF4 sequential I/O and it scales pretty well with increasing number of processors. However, unlike the real PVFS which generates integrated files, the file system used in this experiment does not keep any metadata of the partitioned file and there's no way to extract the distributed output files for other applications to use.

Performance Evaluation: HDF5



Comparison of I/O write performance for HDF5 I/O vs MPI-IO (on SGI Origin2000)

The performance of HDF5 I/O is much worse than we expected. Although it uses MPI-IO for its parallel I/O access and has optimizations based on access patterns and other metadata, the overhead of HDF5 is very significant, as we mentioned in previous discussion