# Improving Parallel IO Performance of Cell-based AMR Cosmology Applications


Yongen Yu
*Department of Computer Science*
*Illinois Institute of Technology*
*Chicago, IL*
*yyu22@iit.edu*

Douglas H. Rudd
*Yale Center for Astronomy and Astrophysics*
*Yale University*
*New Haven, CT*
*douglas.rudd@yale.edu*

Zhiling Lan
*Department of Computer Science*
*Illinois Institute of Technology*
*Chicago, IL*
*lan@iit.edu*

Nickolay Y. Gnedin
*Theoretical Astrophysics Group*
*Fermi National Accelerator Laboratory*
*Batavia, IL*
*gnedin@fnal.gov*

Andrey Kravtsov
*Department of Astronomy and Astrophysics*
*The University of Chicago*
*Chicago, IL*
*andrey@oddjob.uchicago.edu*

Jingjin Wu
*Department of Computer Science*
*Illinois Institute of Technology*
*Chicago, IL*
*jwu45@iit.edu*



***Abstract*—To effectively model various regions with different resolutions, adaptive mesh refinement (AMR) is commonly used in cosmology simulations. There are two well-known numerical approaches towards the implementation of AMR-based cosmology simulations: block-based AMR and cell-based AMR. While many studies have been conducted to improve performance and scalability of block-structured AMR applications, little work has been done for cell-based simulations. In this study, we present a parallel IO design for cell-based AMR cosmology applications, in particular, the ART(Adaptive Refinement Tree) code. First, we design a new data format that incorporates a space filling curve to map between spatial and on-disk locations. This indexing not only enables concurrent IO accesses from multiple application processes, but also allows users to extract local regions without significant additional memory, CPU or disk space overheads. Second, we develop a flexible N-M mapping mechanism to harvest the benefits of N-N and N-1 mappings where N is number of application processes and M is a user-tunable parameter for number of files. It not only overcomes the limited bandwidth issue of an N-1 mapping by allowing the creation of multiple files, but also enables users to efficiently restart the application at a variety of computing scales. Third, we develop a user-level library to transparently and automatically aggregate small IO accesses per process to accelerate IO performance. We evaluate this new parallel IO design by means of real cosmology simulations on production HPC system at TACC. Our preliminary results indicate that it can not only provide the functionality required by scientists (e.g., effective extraction of local regions and flexible process-to-file mapping), but also significantly improve IO performance.**



***Keywords*-parallel I/O, adaptive mesh refinement, data layout, high performance computing, cosmology simulations**


## I. Introduction

Computational simulation is vital for many scientific inquiries, especially in those fields where researchers cannot build and test models of complex phenomena in the laboratory. For example, astrophysics and cosmology is such a field where astronomers cannot perform direct experiments with the objects of their study. Instead, they have to build up numerical simulations to model the universe. To effectively model various regions with different densities in the universe, adaptive mesh refinement (AMR) is widely adopted for cosmology simulations, which enables high resolution in localized regions of dynamic, multidimensional numerical simulations [1]. There are two different numerical approaches towards the implementation of AMR-based cosmology simulations: block-based and cell-based. Block-structured AMR achieves high spatial resolution by inserting smaller grids ("blocks") at places where high resolution is needed [2, 3, 4]. Cell-based AMR instead performs refinement on a cell-by-cell basis. In practice, these two methods are very different, as they use different data structures and methods for distributing the computational load. While many studies have been conducted to improve performance and scalability of block-structured AMR applications [5, 6, 7], little work has been done for cell-based AMR applications. In this paper, we target cell-based AMR cosmology applications, in particular, the adaptive refinement tree (ART) code where a fully threaded tree(FTT) is used to represent cells and relationships [8, 9].

IO performance is vital for cell-based AMR cosmology applications like ART. These applications typically need to store intermediate data periodically, for the purpose of data analysis/visualization or fault tolerance [10]. Each I/O phase of these applications may involve huge amount of data transmission due to the complexity of data structures employed. Moreover, as simulations become more sophisticated and involve more physical processes, they require more memory to accommodate all the information about various physical variables. Limited by the available memory resource, these simulations may require out-of-core execution such that they transfer calculated data to disk and read them back when needed. Finally, file access is a simple yet effective way to exchange data among processes.

In this paper, we present a set of techniques to promote

parallel IO performance of the ART code. Our design contains three major components. First, we design a new data format that incorporates a space filling curve to map between spatial and on-disk locations. It not only enables concurrent IO accesses from multiple application processes, but also allows users to extract local regions without significant additional memory, CPU or disk space overheads. Moreover, the design is self-describing such that the metadata are described within the format itself, ensuring robust data sharing between different analysis tools. Second, we develop a flexible N-M file mapping format to harvest the benefits of N-N and N-1 mappings, where N is the number of application processes and M is a user-tunable parameter standing for the number of files. It not only overcomes the limited bandwidth issue of N-1 mapping by allowing the creation of multiple files, but also enables users to efficiently restart the application at a variety of computing scales. Third, we develop a user-level library called AppAware to transparently and automatically aggregate small IO accesses from the same process to accelerate IO performance.

We evaluate our design by means of real ART simulations on the production system called *l*onestar at TACC [11]. We examine the performance of our new IO design under a variety of configurations such as different computing scales and different buffer sizes. Our preliminary results indicate that the newly developed N-M mapping and data layout are flexible and can be used to efficiently extract local regions without significant memory or disk requirements. The new IO design also enables the application to efficiently restart on different computing scales without sacrificing performance. Experimental results also indicate that the user-level buffering mechanism adopted in AppAware can boost the IO performance of cosmology simulations.

Although our implementation is specific to the ART code, two out of three main I/O techniques we present in the paper are general to other applications using SFC and fully threaded tree[12, 13, 14, 15]. They are the two-level indexing designed for flexible N-M mapping and the I/O optimization techniques encapsulated in the AppAware.

This paper is organized as follows. Section II provides the background information. We then describe our parallel IO design in Section III-V. Section VI presents our experiments. Section VII discusses the related studies. Finally, we draw conclusions in Section VIII.

## II. Background

AMR proposed by Berger et al. is a type of multi-scale algorithm that achieves high spacial resolution in localization regions for dynamic and multidimensional numerical simulations [1].

### A. Cell-based AMR

In a cell-based AMR application, we start with an uniform grid and the cells in this grid are called root cells. Each root cell is an individual computing unit. If higher spatial resolution is required, a cell can be refined into multiple finer cells. With each level up, the cell size is decreased by a refinement factor $r$. Figure 1 shows an example of 3D cell-based AMR with a refinement factor of 2. In the example, the root cell is refined into eight($2^{dim}$) children cells, and two of them are further refined to a higher level.

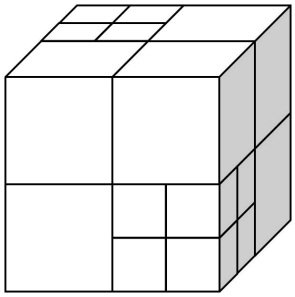


Figure 1. An example of a root cell in 3D cell-based AMR.

To effectively index root cells, Hilbert's space-filling curve (SFC) is commonly used. It assigns a unique SFC identification number (denoted as SFC index) to each root cell according to its spacial coordinates in the overall computational volume. A fully threaded tree is commonly used to represent refinement cells and their relationships. According to [8], "fully threaded" means that every cell, whether a leaf or not, has an easy access to its children, neighbors, and parents. Figure 2 illustrates the relationship between SFC index and its corresponding fully threaded tree[1].

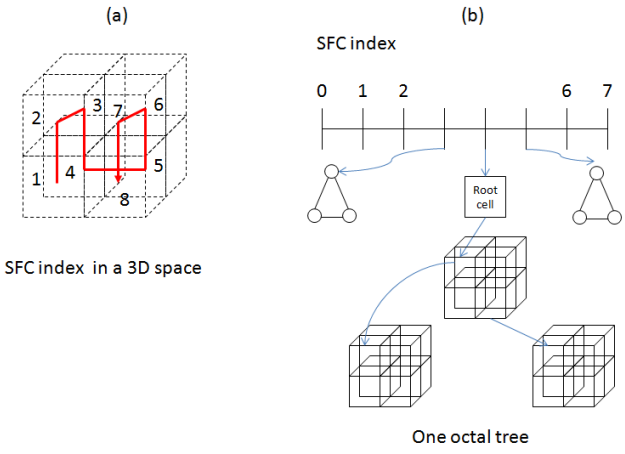



Figure 2. A space filling curve (SFC) and its corresponding fully threaded tree. Figure (a) gives an example of SFC index in a 3D space. The top layer of Figure (b) shows the SFC index and the bottom layer of Figure (b) shows the corresponding octal tree for the 5th root cell.

[1]In this paper, we use "fully threaded tree" and "octal tree" interchangeably.

### B. Cosmology Simulation Code

As mentioned earlier, in this paper we will evaluate our design with a production cosmology simulation code called ART (Adaptive Refinement Tree) [9]. The code was originally developed by A. Kravtsov and A. Klypin, and is currently in use at several sites including the University of Chicago, Fermilab, New Mexico State University, Yale University, and University of Michigan. It combines multi-level particle mesh and shock-capturing Eulerian methods to simulate the evolution of the dark matter and gas, respectively. High dynamic range is achieved by applying cell-based AMR to both gas dynamics and gravity calculations. It is distinguished from other cosmological simulation codes in the number of physical processes it contains. Many physical calculations have been implemented first time in the ART code, making it unique in its capabilities [16, 17, 18].

ART is a hybrid "MPI+OpenMP" C code, with Fortran functions for computing-intensive routines. The MPI parallelization is used between separate computing nodes and the OpenMP parallelization is used within a multi-core node. This mixed parallelization mode enables users to take full advantage of modern multi-core architectures.

## III. Flexible N-M Mapping

N-1 mapping and N-N mapping are two widely used file mapping formats for parallel IO. An N-N mapping generally delivers good I/O bandwidth, but is much more complicated to use when restarting with different number of processes. In addition, when moving towards petascale simulations, the concurrent creation of hundreds of thousands of files is a great challenge for the underlying file system. On the other hand, an N-1 mapping appeals to application developers because a single file is much easier to manage, even when restarting with different number of processes; however, it suffers from low IO bandwidth. Neither of them is scalable when we move towards petascale simulations. In this section, we describe a flexible N-M mapping to combine the merits of N-1 and N-N strategies for cell-based AMR applications. Here, N is number of application processes (e.g., MPI processes) and M is a user-tunable parameter for number of files and typically smaller than N.

Figure 3 depicts our metadata design, which includes a parameter table and two-level index tables. It is designed to support flexible N-M mapping between processes and files. The parameter table records global parameters, such as the number of files (F) and the total number of octal trees (T). The first level index table records the starting SFC index of each of the F files. Based on this table, it can quickly locate the file which contains a needed octal tree via a binary search. Our design supports two mapping strategies: one is object-based and the other is process-based. In the object-based strategy, all the objects (octal trees) are equally distributed among the F files. In the process-based strategy, application processes are divided into F equal-sized groups,

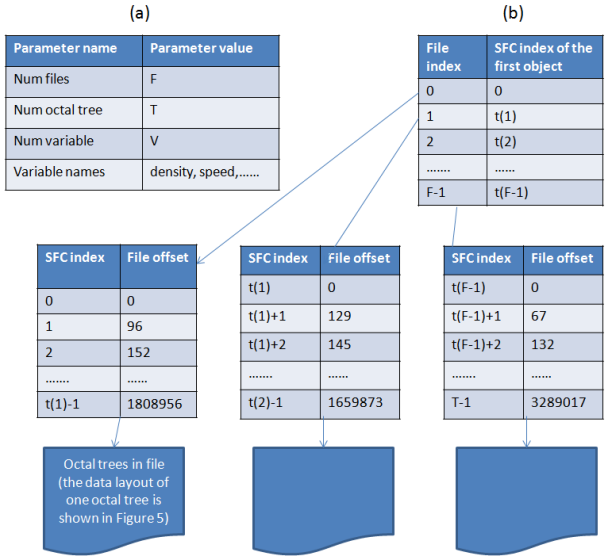



Figure 3. (a) Parameter Table and (b) N-M mapping via a two levels index table where the top table is the first level index table and the bottom tables are the second level index tables.

each accessing one of the F files. Hence, in the object-based strategy, each file contains similar number of octal trees, whereas in the process-based strategy, each file is supposed to be accessed by a similar number of application processes. A flag is provided to users to allow them to choose one of the strategies. We use the term of *process group* to denote the processes accessing the same file. Figure 4 gives an example to illustrate object-based strategy and process-based strategy, where we have 8 application processes writing to 4 files, and there are 100 octal trees in the simulation.

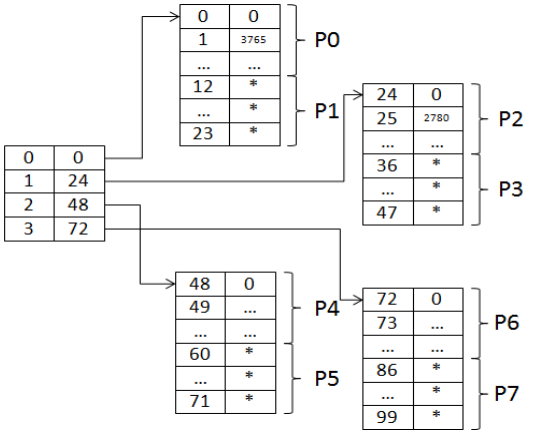
**Procedure 1** Create the first level index table

**Input** T: The number of trees, N: The number of processes, M: The number of files.

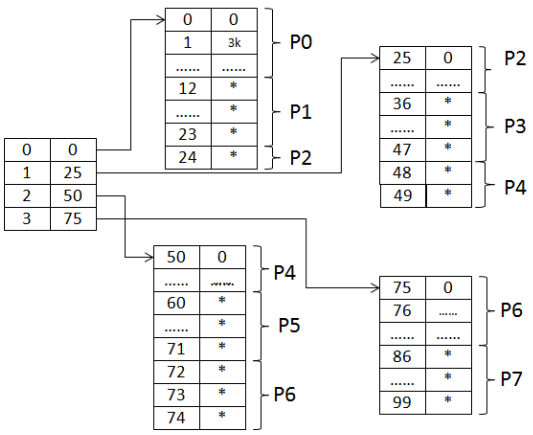
**Output** $global\_index[*]$: the first-level index table. $P_{start}[i]$ the process that owns the first octal tree of the $i^{th}$ file

1. **if** process-based mapping **then**
2. Divide the N processes into M groups and assign one file for each group
3. for each $i \in [1, M], P_{start}[i] \leftarrow$ the first process within group i.
4. for each $i \in [1, M], global\_index[i] \leftarrow$ the first SFC index of $P_{start}[i]$
5. **else if** object-based mapping **then**
6. for each $i \in [1, M], global\_index[i] \leftarrow i * T/M$
7. Based on $global\_index[i]$ and the SFC region of each process, identify $P_{start}[i]$
8. **end if**

In the creation of the first level index table, one of the




(a) **process-based mapping** File 0 is accessed by P0 and P1, storing SFC indices $\{0, ..., 11, 12, ..., 23\}$. File 1 is accessed by P2 and P3, storing SFC indices $\{24, ..., 35, 36, ..., 47\}$. File 2 is accessed by P4 and P5, storing SFC indices $\{48, ..., 59, 60, ..., 71\}$. File 3 is accessed by P6 and P7, storing SFC indices $\{72, ..., 83, 84, 85, 86, ..., 97, 98, 99\}$.




(b) **object-based mapping** File 0 is accessed by P0, P1 and P2, storing SFC indices $\{0, ..., 11, 12, ..., 23, 24\}$. File 1 is accessed by P2, P3 and P4 , storing SFC indices $\{25, ..., 49\}$. File 2 is accessed by P4, P5 and P6, storing SFC indices $\{50, ..., 74\}$. File 3 is accessed by P6, P7, storing SFC indices $\{75, ..., 99\}$.

Figure 4. An example to compare object-based mapping with process-based mapping. Assume there are 8 application processes owning 100 octal trees writing to 4 files. Assume that P0 owns the following SFC cells $\{0, ..., 11\}$; P1 owns the following SFC cells $\{12, ..., 23\}$; P2 owns the following SFC cells $\{24, ..., 35\}$; P3 owns the following SFC cells $\{36, ..., 47\}$; P4 owns the following SFC cells $\{48, ..., 59\}$; P5 owns the following SFC cells $\{60, ..., 71\}$; P6 owns the following SFC cells $\{72, ..., 83, 84, 85\}$; P7 owns the following SFC cells $\{86, ..., 97, 98, 99\}$

application processes is selected as the master and is responsible of storing the table onto the disk. The first level index table is cached in the memory of each application process for better access performance. Procedure 1 presents the pseudo-code of creating the first level index table denoted by $global\_index[i]$ (i is the file id) and $P_{start}[i]$(the process that owns the first octal tree of the $i^{th}$ file) which is needed for creating the second level index table.

The creation of the second level index tables is more complicated than that of the first level index table. First, there are F second level index tables, storing along with the F files. All application processes are involved in the creation of these tables. For each application process, based on its owned SFC cells (hence knowing the starting and end SFC indices), it can quickly determine the file(s) it should access based on the first level index table. A process may only access one file or possibly multiple files. By looking up the values in the first level index table, it is easy to determine *process group* for each file. During the creation of the second level index tables, each process in a process group calculates file offset for each SFC index it owns and fills in the right column of the corresponding second level index table in a sequential order. It then sends the file offset of the next available space to the following process in the same group. While the creation of each second level index table is sequential, the creation of the F second level index tables is concurrent.

When the application restarts, no matter whether it is at the same or a different computing scale, each process first reads the parameter table, followed by the first level index table. According to the SFC cells (i.e., the starting and end SFC indices) allocated to the process, it can quickly determine the files and the segments of the second level index table corresponding to its SFC cells. The first level index table, together with the segments of the second level index tables are cached in memory for quickly locating any octal tree in file.

## IV. Data Layout for Octal Trees

In the above section, we describe the flexible N-M mapping such that N processes write to M unique files. The two-level index tables shown in Figure 3 enable an efficient mapping from SFC index to file locations. In this section we describe our new data layout for octal trees, those shaded boxes shown in Figure 3. In our new design, the octal trees or root cells are kept in the M files ordered by SFC index, and each octal tree data is recorded at one contiguous region. Figure 5 shows our self-descriptive data layout. This data layout can be divided into several regions. The first region lists the variables, such as density and speed, for root cell t(i) (i.e., the $i^{th}$ octal tree), followed by its depth and number of nodes per level. To reduce the amount of reference links, the eight children of each tree node will share a single link to their parent, therefore we combine these eight children as one tree node. The second region records the variables for its eight level 1 children, followed by a reference tag array indicating whether each child is further refined into a higher level (e.g., level 2) or not. The third region records the data for level 2 children, and this recording proceeds until reaching the highest level of the octal tree.

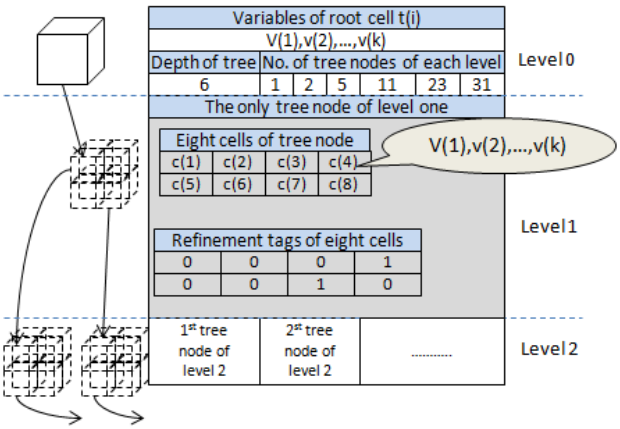



Figure 5. Data layout of one octal tree.

In Figure 5, we also give an example to illustrate an octal tree and its corresponding data layout in our new design. There are k variables for each cell. Hence the variables of root cell are stored in an array of size k. Following is a single number of 6, indicating there are 6 levels of the tree. The next information indicates that there are 1 and 2 tree nodes on the first two levels of the tree. Within each tree node, there are two arrays. An array of size 8*k recording the variables of the eight children cells, and another array indicates which cell is refined further into a higher level. In this example, the $4^{th}$ and the $7^{th}$ cell are refined. Since there is only one tree node on level 1, data layout moves to the nodes on level 2.

Our new data layout, along with the N-M mapping, allows users to easily extract a local region (e.g., an octal tree) without going through every file. Take an example, suppose a scientist wants to obtain the data regarding a specific octal tree for visualization. Only two lookups through the two-level index table will complete the work: (1) first to determine the file (out of the m files) by checking the first level index table; and (2) then to retrieve the file offset according to the cached second level index table. As shown in Figure 3 and 5, this process does not require significant memory, CPU, or disk space.

## V. I/O Optimization

The previous section describes our new data layout describing fully threaded tree used by cell-based AMR simulations. As the refinement proceeds, the depth and the number of tree nodes per level are dynamically changing during simulation. While a number of high-level IO libraries have been developed for efficiently describing and managing scientific data like parallel HDF5 and parallel netCDF [19, 20, 21, 22], it is very difficult to map the aforementioned fully threaded tree into the HDF or netCDF linear structures. As an example, each octal tree is often dynamically adjusted during the course of execution. Thus, in order to use HDF, we would have to allocate sufficient space to hold a full octal tree for each SFC cell. As a result, file size would increase exponentially with the number of refinement levels. Moreover, most of the octal trees are not full, thereby leaving many holes in the file and wasting lots of storage space. Thus, we cannot effectively describe the dynamic cell-based AMR datasets using either HDF or netCDF.

We therefore decided to use MPI-IO [23, 24, 25] as it provides a set of I/O interfaces allowing an application to access a shared file with a user-defined data format. Our initial MPI-IO implementation with ART didn't give us a satisfactory IO performance. For example, for a medium-scale cosmological simulation with $2^{21}$ octal trees, it took over a hundred of minutes to generate a 12GB intermediate data using MPI-IO. Data sieving and collective I/O are two key optimizations for MPI-IO [26]. In order to determine whether any of them is suitable for the application, we conducted a detailed IO performance analysis of ART. For the purpose of brevity, we present two major results here, namely Figure 6 and 7.

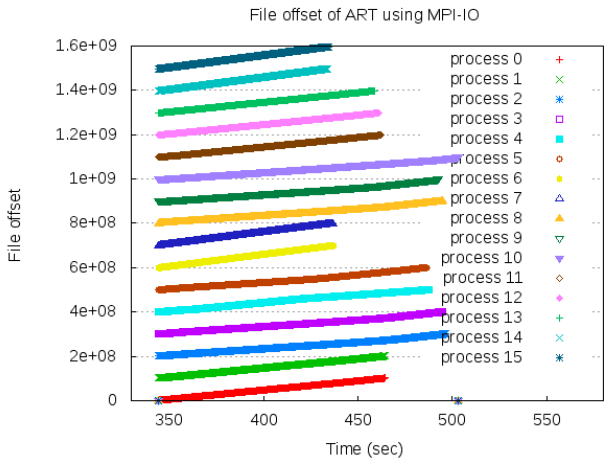



Figure 6. I/O access patterns of ART using MPI-IO. In this example, 16 processes write to a file collectively, and then read the data from the storage. We use points of different colors and shapes to represent the IO accesses of each process. The figure depicts the file offset as a function of time for each process during the reading phase.

Figure 6 plots IO access patterns of the application running on 16 computing nodes. We make two interesting observations from the figure. First, we can see that the access region of each process is disjointed with each other. According to Section IV, in the new data layout, the list of SFC cells is stored in files and each process writes its data (i.e., octal trees) to files in a contiguous manner. Hence, the access regions of application processes are not overlapping with each other. Collective I/O is good for the case where each process needs to access several non-contiguous portions of a file and these requests of different processes are interleaved. This is not the case for our application, therefore collective

I/O is not suitable for improving IO here. Second, the figure clearly shows that each process always accesses a contiguous region of the file. According to [26], data sieving improves IO performance in the case of non-contiguous request from each process by using derived datatypes to describe the access patterns. Unfortunately, it is hard or impossible to describe the access patterns of ART with derived datatypes as the data structures are dynamically changing during the course of execution, the structures of different octal trees are distinct and the file format does not have repeated patterns. Thus, data sieving is also not suitable for improving IO performance of the application.

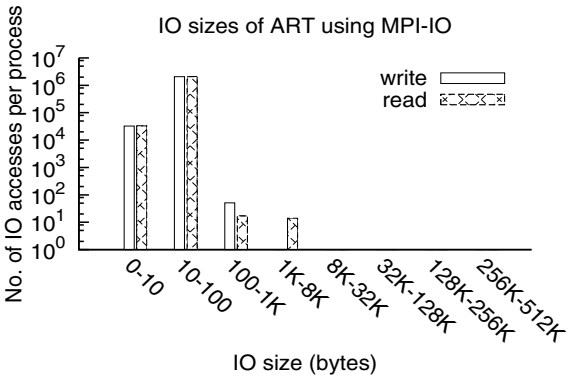



Figure 7. I/O request sizes of ART. In this example, 16 processes generate an intermediate file of 1.5 GB

We further analyzed sizes of IO requests and intended to understand why MPI-IO implementation cannot provide a satisfactory performance. Figure 7 shows the distribution of IO requests of the application. It is clearly shown that most of the IO requests are small sized. All the IO requests are within 8 KB, and more than 99.998% of the requests are within 100 bytes. The results are not surprising. According to Figure 5, while the new data layout allocates various data contiguously in a file, the dataset contains a large amount of variables, each with a small size. The large amount of small IO accesses is the primary reason for the poor performance shown in our initial MPI-IO implementation.

By considering the unique I/O patterns of ART (i.e., non-overlapping accesses from processes, contiguous IO access per process and a large amount of small IO accesses), we designed and developed our own IO optimization mechanism called AppAware (Application-Aware Write After and Read ahEad). In AppAware, when the application performs I/O, a user level buffer is allocated per process, which is used to aggregate sequential I/O requests of different types and sizes per process. During a write phase, various data of different types and sizes as shown in Figure 5 are converted to bytes and then combined together into the buffer. Once the buffer is full, the content of the buffer is transferred to the disk automatically as one IO access. As for reads, the data are moved in a reverse direction. Figure 8 gives an overview of AppAware.

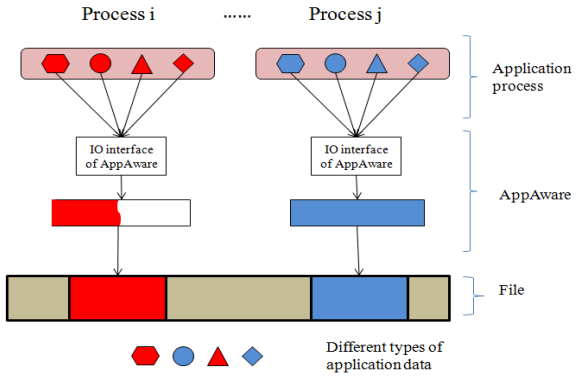



Figure 8. Description of AppAware.

In AppAware, the buffer size is a user tunable parameter specified by the user when file is open. At writes, the dirty data are flushed to the disk automatically by AppAware once the buffer is full. At reads, when the required data are not cached in the buffer, a large contiguous data block containing the required data will be loaded into the memory for read ahead operations.

Moreover, to relieve users from the complicated buffer management, we implement AppAware as a user-level library, which has the same APIs as MPI-IO. The buffer size, a user tunable parameter, is passed to AppAware by "MPI Hint". Another benefit of implementing AppAware as a user-level library is that it can also be used by other applications when they have the similar I/O patterns. If one wants to use AppAware, all s/he needs to do is to change the compilation option by linking to the AppAware library.

## VI. Experiments

We evaluate our IO design by means of real cosmology simulations using the ART code on the production system called *Lonestar* at TACC. *LoneStar* is a 1,888-node cluster and each node contains two 6-Core processors. Centos 5.5 is installed on each node and these nodes are connected by InfiniBand in a fat-tree topology with a 40Gbit/sec point-to-point bandwidth. Each node holds up to 24GB memory and a 146GB local disk. Parallel file system Lustre [27] is installed on this machine that provides 1PB storage capability. SGE is used for job scheduling and resource management.

In our experiments, each process owns the same number of octal trees, and we set these trees of the same size. By doing so, we can eliminate the variation brought by unbalanced workload and use the average time of application processes as our experiment results. Besides, for simplicity, we let the simulation first dump the intermediate data and then restart from this snapshot.

All the experiments were conducted during production mode, meaning other users coexisted in the system. To minimize the variation in the performance results, a minimum

of 5 runs were conducted for each test and we present the average values.

### A. IO Patterns W/ AppAware

In Section V, we present the I/O access patterns and I/O request sizes of ART using MPI-I/O (Figure 6 and 7). Here, we list the corresponding I/O access patterns and request sizes after using AppAware (Figure 9 and 10). They are drawn from the "strace" logs. "strace" [28] is a command line instrument which can be used for tracing system calls and signals of a single process. In this set of experiments, we modified the batch job script to generate a trace file for each process. Similar to the settings used in Figure 6 and 7, the simulation generates an intermediate file of $2^{18}$ octal trees via 16 computing processes, with a size of 1.5GB. The default buffer size of AppAware is set to 1 MB. The overall size of strace logs is decreased from 471 MB when using the MPI-IO to 774 KB when using AppAware. This indicates that the amount of system calls related to IO operations is reduced drastically by using AppAware.

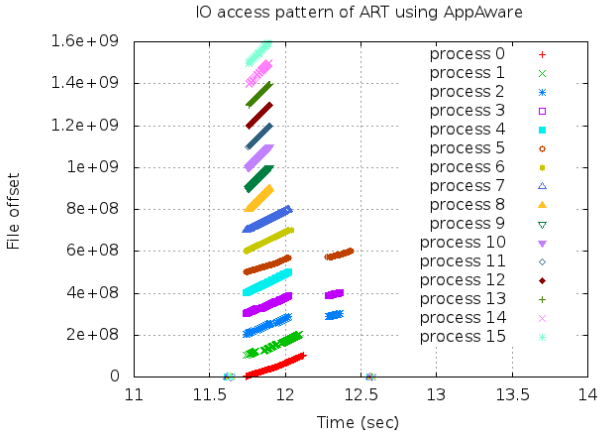



Figure 9. I/O access patterns of ART using AppAware.

Comparing Figure 9 and Figure 6, we can see that reading time is greatly reduced after using AppAware, from 150+ seconds to less than one second and the overall IO time of the application is reduced from 500 seconds to 12.5 seconds. Moreover, the number of IO operations is significantly reduced due to the IO aggregation used by AppAware. Similar to the pattern listed in Figure 6, the file offset per process increases with time, indicating the IO access patterns, the sequential and contiguous IO requests per process, are not changed. And further, the access regions of different processes are disjoint.

Comparing Figure 10 and Figure 7, we make several interesting observations. First, the total number of IO requests is reduced by using AppAware. For example, here the number of IO requests for each category is always less

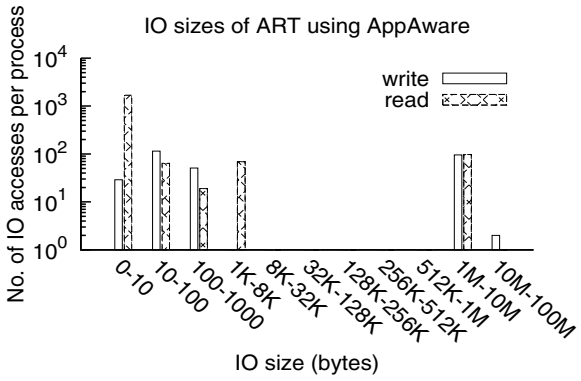



Figure 10. I/O request sizes of ART using AppAware.

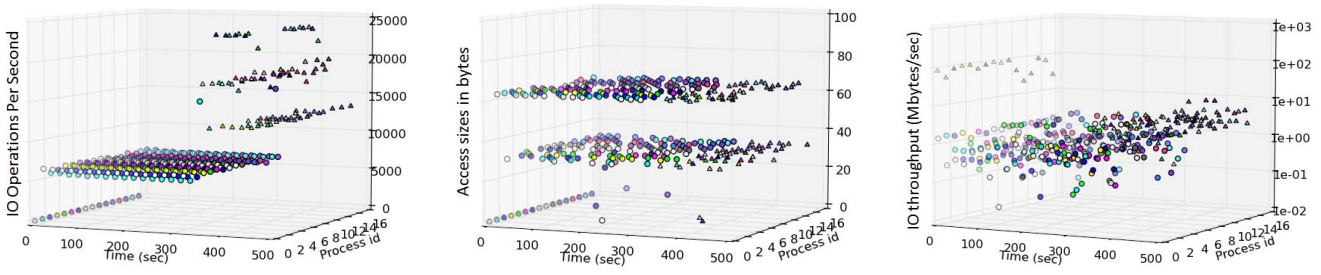
than $10^4$, whereas in Figure 7, there are more than $10^6$ IO requests of size of 10-100 bytes. Next, in Figure 7, all the IO requests are less than 8 Kbytes, whereas there are a number of IO requests of size of 1Mbytes-10Mbytes in Figure 10. The reason is that AppAware improves IO performance by aggregating a large amount of small IO requests into fewer yet larger IO requests. In short, Figure 9 and 10 illustrate that the use of AppAware can significantly reduce the amount of IO requests, thereby improving IO performance of the application.

### B. System-Level IO Performance

We now move to evaluate system level IO performance explicitly. IO throughput is a commonly used performance measurement which is determined by two measurements: IOPS (Input/Output Operations Per Second) and data access size. Hence, we compare IO performance by using AppAware as against using MPI-IO by means of these metrics. Here, "system-level" means that we use the performance data captured in the "strace" logs for performance comparison.

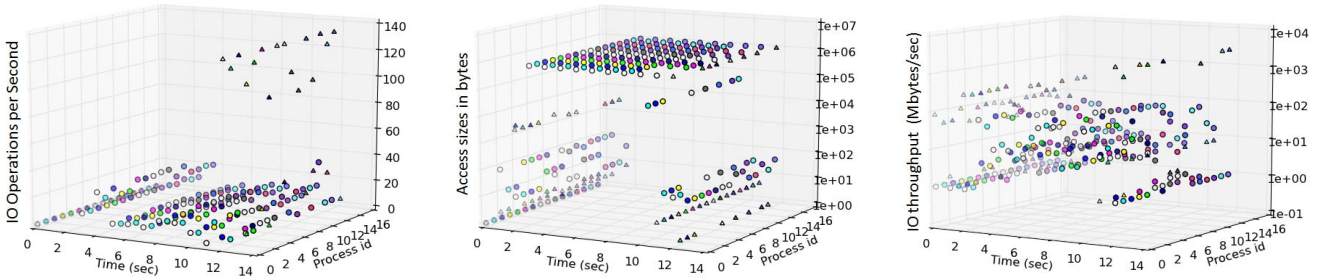
Figure 11(a) and Figure 12(a) depict the IOPS of ART using MPI-IO and AppAware respectively. By comparing these figures, we can observe that both the write and read IOPS are greatly reduced after using AppAware. The average write IOPS decreases from 6000 to less than 40 and the average read IOPS drops from 24000 to less than 140. It indicates AppAware can reduce the number of IO operations tremendously.

Figure 11(b) and 12(b) show the IO access sizes along with the time relative to the beginning of the run by using MPI-IO and AppAware respectively. Figure 11(b) shows that most of IO operations are less than 80 bytes when using MPI-IO interfaces. This is in line with the array sizes in Figure 5. In Figure 12(b), the access sizes are much bigger. Except for some small IO accesses at the beginning and end, both the write and read sizes maintain around a fixed value (about $10^6$) which is the buffer size used by AppAware. This




(a) IOPS (b) IO access sizes (c) IO throughput

Figure 11. The IOPS, access sizes and throughput of ART code using MPI-IO. In these figures, circles stand for writes and triangles stand for reads. The sampling interval is 20 seconds.




(a) IOPS (b) IO access sizes (c) IO throughput

Figure 12. The IOPS, access sizes and throughput of ART code using AppAware. In these figures, circles stand for writes and triangles stand for reads. The sampling interval is 1 second.

confirms that AppAware can aggregates small accesses into big ones.

Figure 11(c) and 12(c) present IO throughput of ART using different IO interfaces. As for MPI-IO, the write throughput ranges from $10^{-2}$ Mbytes/sec to several Mbytes/sec and the read throughput varies from $10^{-1}$ Mbytes/sec to several Mbyte/sec. Figure 12(c) shows the same experiment except that we use AppAware as the IO interface. Apart from the small accesses within the first 2 seconds, the write throughout varies from several Mbytes/sec to $10^2$ Mbytes/sec and the read throughput is from $10^2$ Mbytes/sec to $10^3$ MBytes/sec. The huge difference of IO throughput by using different IO interfaces indicates that AppAware can boost IO performance dramatically. Moreover, these figures also show that the performance improvement by using AppAware stems from the aggregation of IO operations.

### C. Application-Level IO Performance

Table I
EXPERIMENT PARAMETERS

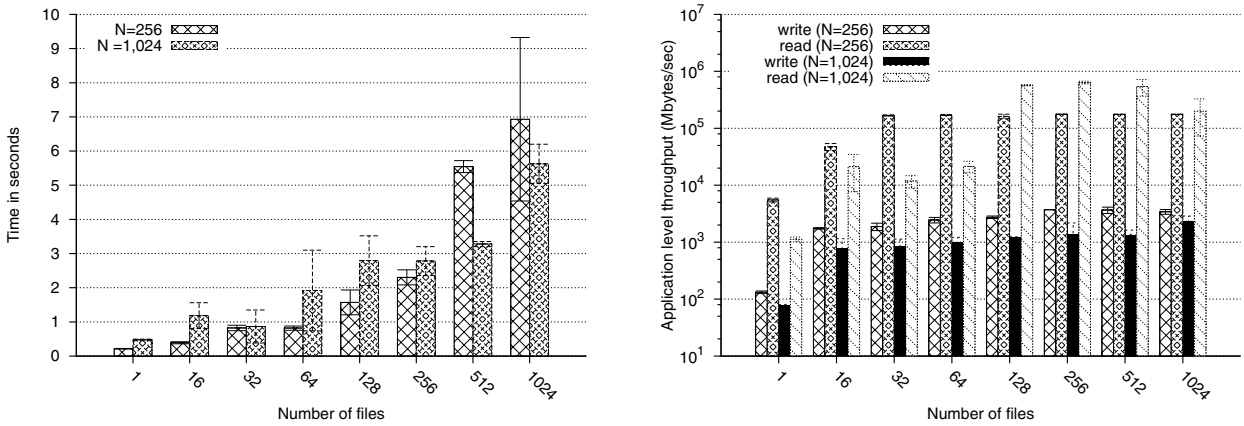
| parameter name | parameter value |
|---|---|
| number of octree | $2^{23}$ |
| M: number of files | $1 \Rightarrow 1024$ |
| file size | 48GB (in total) |
| N: number of application processes | $16 \Rightarrow 1024$ |
| file system | Lustre |
| AppAware buffer size | 1MB |

In this subsection, we evaluate application level IO performance. In particular, we compare application-level IO time and application-level throughput by using our IO design.

We carry out two sets of experiments. In the first set of experiments, we fix N (the number of application processes), while varying M (the number of files) from 1 to 1,024. In the second set of experiments, we fix M while varying N from 16 to 1,024. Table I summarizes the parameters used in these experiments. We use a cosmology simulation with over 8 million of octal trees, leading to intermediate files of about 48 GB.

Figure 13 presents the results of the first set of experiments, where N is fixed. Figure 13(a) plots file creation times. It shows that the creation time roughly goes up with the number of files. Here, file creation cost includes the time to create files and build the two-level index tables. Parallel file system like Lustre employs metadata server to handle IO requests. File creation operations, especially creating many files within the same folder, place a heavy burden on the metadata server. This explains why the creation time becomes longer when we create more files.

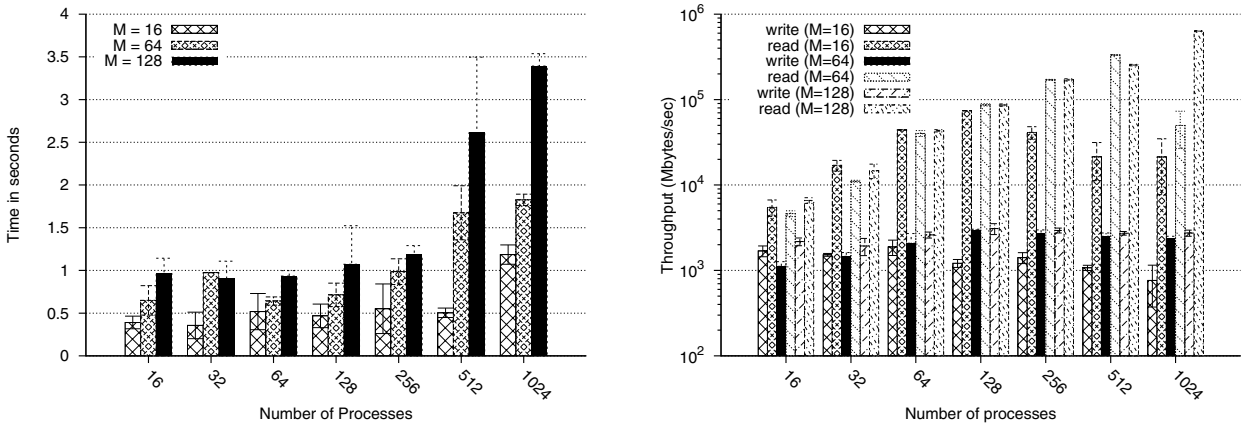
Figure 13(b) plots application-level IO throughput for the first set of experiments. This figure indicates that application-level write throughput reaches the highest value when the number of files equals the number of processes (i.e., in case of a N-N mapping). As for reads, the highest




(a) Creation cost of metadata

(b) Application level IO throughput

Figure 13. N-M IO performance, where N is fixed (256 or 1,024) and M is varying from 1 to 1,024




(a) Creation cost of metadata

(b) IO throughput

Figure 14. N-M IO performance, where M is fixed (16, 64 or 128) and N is varying from 16 to 1,024

read throughput is achieved when N is set to 256. We shall point out, when N is set to 1,024, the read throughput does not achieve the highest value when M equals to 1,024. This indicates that N-N mapping does not always deliver the best performance for large-scale applications like cell-based AMR cosmology simulations and a flexible N-M mapping is necessary.

Figure 14 presents experimental results for the second set of experiments, where M is fixed at 16, 64 or 128 while N is increasing from 16 to 1,024. Figure 14(a) plots file creation times. When M is fixed at 16, file creation cost roughly increases with the number of processes and the best case is achieved when the number of processes is set to 16 (i.e., a N-N mapping). When the number of processes is larger than the number of files, each second level index table is created by multiple processes in a pipeline manner. As a result, network communication may impact the creation process, thereby leading to longer file creation time. When N is fixed at 64, again the best case is achieved when M is set to 64. When N is greater than M, a larger creation cost is observed due to the involved network communication among processes; when N is smaller than M, each process needs to create multiple files and their corresponding second-level index tables, thereby resulting in longer creation time. When M is fixed at 128, the shortest creation time is not observed when N is set to the same value. One possible reason for this might be the interference from other applications running on the system when we collected our results. This figure shows that the standard deviation is very large when N is set to 128. That indicates the shortest creation time when N is set to 128 will be less than the shortest value when N is set to 32. It confirms the assumption. The key observation from Figure 13(a) and 14(a) is that the longest creation time is always smaller than 7 seconds, indicating our N-M mapping strategy does not cause much overhead to the application.

Figure 14(b) plots the application-level I/O throughput for the second set of experiments. In general, the peak write throughput roughly appears when N equals M. For writes,

data are moved from IO clients (application processes) to IO servers. More IO clients indicates higher data generation rate, therefore more data is transferred to IO servers. Thus, write throughput increases with the number of processes. However, the bandwidth to IO server is limited, the high data generation rate brought by more IO clients starts to overflow IO servers when there are too many application processes competing for the limited bandwidth, leading to a drop. The read throughput is more sensitive to the number of processes as compared to write throughput. Similar to write throughput, it initially increases with the number of application processes and then drops. However, the highest read throughput does not appear when N equals M: when M is set to 16, 64, and 128, the peak value is achieved at 128, 512, and 1024 respectively.

In practice, the number of processes is generally larger than the number of files, so we do not discuss the cases when N is smaller than M. In this set of experiments, the maximum value of M is 128, so we discuss the cases when N is larger than 128. Figure 14(b) shows that when N is larger than 128, write throughput with M being set to 64 is almost the same as that with M being set to 128, is greater than that with M being set to 16. The IO server bandwidth of Lustre is limited by the number of object storage targets (OSTs). By default, each file on Lonestar is stored to a single object storage target since the stripe size is set to 1. Hence, the IO server bandwidth is limited by the number of files. This explains why write throughput with M being set to 64 is greater than that with M being set to 16. Lonestar is configured with 30 OSTs, which limits the amount of IO bandwidth even if the number of files is more than 30. This is in line with the observation that write throughput with M being set to 64 is almost the same as that with M being set to 128.

### D. Sensitivity to AppAware Buffer Size

When using AppAware, IO throughput is influenced by its user-level buffer size. In this subsection, we tune the buffer size from 64bytes to 1GB. Moreover, in most parallel file systems like Lustre, one file can be either kept on one OST (vertical) or splitted into stripes and stored on multiple OSTs (horizontal). IO performance can be influenced by different placements as well. Therefore, we also compare performance with different strategies (i.e., horizontal placement versus vertical placement). We have conducted three groups of experiments with different N to M ratio.

Figure 15(a) shows experimental results of 16 processes writing to one file. In principle, IO throughput of horizontal data placement should be better than vertical strategy as it employs multiple disks to stored files. This hypothesis holds for most cases, except for three points (64 bytes, 512 bytes and 1 GB). This figure indicates that when buffer size is smaller than 4K, the differences with different data placement strategies are trivial. Application with N-1 mapping cannot benefit from horizontal placement when the IO access size is small. A sharp drop of read throughput is observed when the buffer is set to 1GB. The competition among processes, the available memory resource of each node and the network traffic congestions will overwhelm the benefits brought by large buffer.

Figure 15(b) plots experimental results for the second group (16 processes writing to 16 files). In this set of experiments, we let each file spans across 16 OSTs for the horizontal placement strategy. That means 16 OSTs will be used by the application. As for the vertical strategy, the application can also employ 16 OSTs since the file number is 16. Theoretically, the performance using different data placement is comparable. The figure is in line with the the assumption.

Figure 15(c) shows experimental results when N is set to 64 and M is set to 16. Here, we also let each file spans across 16 OSTs for the horizontal strategy. In this case, the bandwidth of Lustre is the same for vertical and horizontal strategies. The figure confirms the hypothesis.

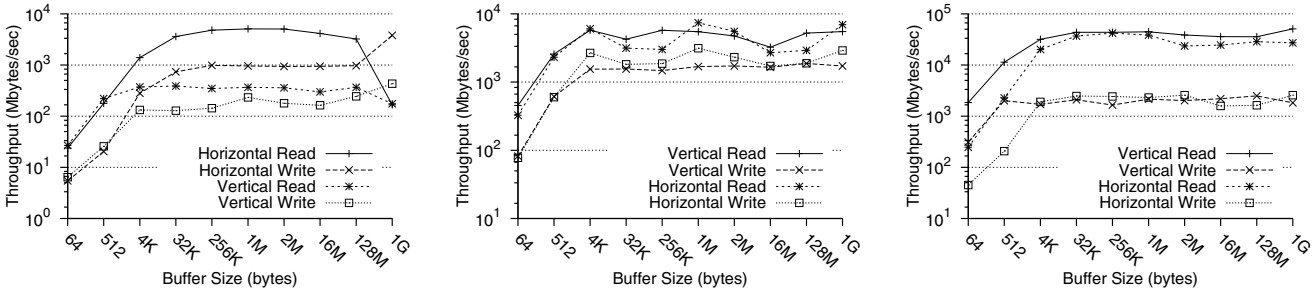
By comparing these figures, we can observe that the application can achieve the best performance when the buffer size is set to several Mbytes. Figure 15(a) and 15(b) indicate that the application has better IO performance by accessing multiple files. Figure 15(c) and 15(b) show that application-level read throughput increases with the number of processes, while write throughput is not.

## VII. Related work

Many studies have been presented to improve parallel IO performance. In general, they can be grouped into two broad categories: one is at system-level like list I/O, MPI caching, resonant IO and active buffering, and the other is at application-level like collective I/O and data sieving. In this section, we discuss relevant studies and point out key differences between our design and these studies.

List I/O is a file system enhancement, aiming at improving noncontiguous I/O access [29]. Active buffering and nonblocking I/O speed up applications performance by overlapping I/O operations with computation [30]. MPI caching boosts I/O performance by caching a single copy of data among multiple computing nodes [31]. Resonant IO rearranges IO requests to avoid unnecessary disk head movement [32]. These are all system-level optimization techniques, while our focus is on application-level optimizations in this study.

Collective IO and data sieving are two key optimizations for MPI-IO [26]. Data sieving aims at improving IO performance of noncontiguous requests from one process. Instead of making many small I/O requests, data sieving enables the application to access a single contiguous chunk of data from the first requested byte up to the last requested byte into a temporary buffer in memory. Collective IO intends for optimizing noncontiguous requests from multiple processes.




(a) M=1 and N=16 (b) M=16 and N=16 (c) M=16 and N=64

Figure 15. IO throughput with different buffer sizes and data placement strategies

It utilizes a two-phase strategy to first let each process access contiguous data chunks and then to redistribute these data among multiple processes. As mentioned in Section V, neither of them is suitable for our cell-based AMR application because of the unique IO access patterns of this application, i.e., disjoint IO requests from different processes and contiguous IO accesses per process without repeated patterns.

That many HPC applications employ application-level buffer to pack/unpack data before performing I/O operations provides the similar functionality. One distinguishing feature of our design is that we provide a user-level library, which frees users from the complicated buffer management so that they can focus on getting their science done. Moreover, the developed library can be used by other applications if they share the similar I/O patterns.

## VIII. Conclusions

In this paper, we have presented a parallel IO design for cell-base AMR simulations. Our design has the following key features:

- *Indexed* - The new data layout incorporates a space-filling curve index to map between spatial and on-disk locations, maintaining spatial locality. This allows users to efficiently extract a local region without significant additional memory, CPU, or disk space overhead.
- *Parallel* - The indexing scheme supports a flexible N-M mapping. It not only overcomes the limited bandwidth issue of an N-1 mapping by allowing the creation of multiple files, but also enables users to efficiently access the data at a variety of computing scales.
- *Extensible* - The new I/O system is designed to be both flexible and self-describing. The numbers, data types, and contents of physical variables are described within the format itself, ensuring robust data sharing and backwards compatibility as new features are added to the old code.
- *Self-contained* - The I/O code can be segregated into a separate library with a fixed API. This isolates users from the underlying file structure as well as platform and performance details. Thus it can be used to provide access to simulation data from a variety of analysis and visualization packages.

We have evaluated our new IO design by means of real cosmology simulations on production HPC system. Our preliminary results indicate that the design can not only provide the functionality required by scientists (e.g., effective extraction of local regions, flexible process-to-file mapping), but also improve IO performance significantly.

## Acknowledgement

This work is supported in part by National Science Foundation grants OCI-0904670.

## References

[1] M. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Computational Physics*, 1984.

[2] R. Rosner, A. Calder, J. Dursi, B. Fryxell, D. Lamb, J. Niemeyer, K. Olson, P. Ricker, F. Timmes, J. Truran, H. Tueo, Y. Young, M. Zingale, E. Lusk, and R. Stevens. Flash code: Studying astrophysics thermonuclear flashes. *Computing In Science and Engineering*, 2000.

[3] G. Bryan, T. Abel, and M. Norman. Achieving Extreme Resolution in Numerical Cosmology Using Adaptive Mesh Refinement: Resolving Primordial Star Formation. In *Proc. of Supercomputing '01*.

[4] A. Razoumov, M. Norman, T. Abel, and D. Scott. Cosmological Hydrogen Reionization with Three-dimensional Radiative Transfer. *Astrophysical Journal*, 572:695–704, 2002.

[5] A. Calder, B. Curtis, L. Dursi, B. Fryxell, G.Henry, P. MacNeice, K.Olson, P.Ricker, R. Rosner, F. Timmes, H. Tufo, J.Turan, and M.Zingale. High-performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors. In *SC '00: Proceedings of the International Conference for High Performance Computing,Networking,Storage and Analysis*.

[6] B. W. O'Shea, G. Bryan, J. Bordner, M. L. Norman, T. Abel, R. Harkness, and A. Kritsuk. Introducing Enzo, an AMR Cosmology Application. *ArXiv Astrophysics e-prints*, March 2004.

[7] Z. Lan, V. Taylor, and G. Bryan. A Novel Dynamic Load Balancing Scheme for Parallel Systems. *Journal of Parallel and Distributed Computing (JPDC)*, 2002.

[8] A. M. Khokhlov. Fully Threaded Tree Algorithms for Adaptive Refinement Fluid Dynamics Simulations. *Journal of Computational Physics*, 1998.

[9] A. V. Kravtsov, A. A. Klypin, and A. M. Khokhlov. Adaptive Refinement Tree: A new High-resolution N-body code for Cosmological Simulations. *Astrophys. J. Suppl*, 111:73–94, 1997.

[10] Z. Zheng, Z. Lan, R. Gupta, S. Coghlan, and P. Beckman. A Practical Failure Prediction with Location and Lead Time for Blue Gene/P. In *Proc. DSNW '10*, 2010.

[11] Extreme Science and Engineering Discovery Environment. https://www.xsede.org/.

[12] A.M. Khokhlov, E.S. Oran, and G.O. Thomas. Numerical Simulation of Deflagration-to-Detonation Transition: The Role of ShockFlame Interactions in Turbulent Flames. *Combustion and Flame*, 2180, 1999.

[13] R. Teyssier. Cosmological hydrodynamics with adaptive mesh refinement A new high resolution code called RAMSES. *Astronomy*, 364:337–364, 2002.

[14] *Performance of Vector / Parallel Orientated Hydrodynamic Code*, chapter Software. Springer, 2005.

[15] P. Diener, N. Jansen, A. Khokhlov, and I. Novikov. Adaptive mesh refinement approach to construction of initial data for black hole collisions. *Astrophysics*, pages 1–13, 2008.

[16] A. Kravtsov. Dark matter substructure and dwarf galactic satellites. *Advances in Astronomy*, 2010.

[17] N. Gnedin and A. Kravtsov. Environmental Dependence of the Kennicutt-Schmidt Relation in Galaxies. *The Astrophysical Journal*, 2011.

[18] D. Rudd, Andrew R. Zentner, and A. Kravtsov. Effect of Baryons and Dissipation on the Matter Power Spectrum. *The Astrophysical Journal*, 2008.

[19] HDF5. www.hdfgroup.org/HDF5/.

[20] Parallel NetCDF. cucis.ece.northwestern.edu/projects/PNETCDF.

[21] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proc. of Supercomputing '03: High Performance Networking and Computing*.

[22] K. Gao, W. Liao, A. Choudhary, R. Ross, and R. Latham. Combining I/O Operations for Multiple Array Variables in Parallel NetCDF. In *Proceedings of the Workshop on Interfaces and Architectures for Scientific Data Storage, held in conjunction with the the IEEE Cluster Conference, New Orleans, Louisiana*, September 2009.

[23] R.Thakur, W.Gropp, and E.Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proc. of IOPADS '99*, 1999.

[24] R. Latham, R. Ross, and R. Thakur. Implementing MPI-IO Atomic Mode and Shared File Pointers Using MPI One-Sided Communication. *International Journal of High Performance Computing Applications*, 21(2):132–143, 2007.

[25] R. Ross, R. Latham, W. Gropp, R. Thakur, and B. Toonen. Implementing MPI-IO atomic mode without file system support. In *Proceedings of CCGrid 2005*, May 2005. Superseded by IJHPCA paper.

[26] R.Thakur, W.Gropp, and E.Lusk. Data Sieving and Collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively parallel Computation*, pages 182–189, 1999.

[27] Lustre. http://wiki.lustre.org.

[28] strace. http://en.wikipedia.org/wiki/Strace.

[29] A. Ching, A. Choudhary, K.Coloma, and W.Liao. Noncontiguous I/O Accesses Through MPI-IO. In *Proc. of IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2003.

[30] X. Ma, M.Winslett, J. Lee, and S. Yu. Faster collective output through active buffering. In *Proc. of Supercomputing '02*, 2002.

[31] W. Liao, K. Coloma, Ward A. Choudhary, and L, E. Russell, and S. Tideman. Collective Caching: Application-aware Client-side File Caching. In *Proc. of HPDC '05*, 2005.

[32] X. Zhang, S. Jiang, and K. Davis. Making Resonance a Common Case: A High-Performance Implementation of Collective I/O on Parallel File Systems. In *Proc. of IPDPS '09*, 2009.