I/O Analysis and Optimization for an AMR Cosmology Application

Jianwei Li Wei-keng Liao Alok Choudhary Valerie Taylor
ECE Department, Northwestern University
{jianwei, wkliao, choudhar, taylor}@ece.northwestern.edu

Abstract

In this paper, we investigate the data access patterns and file I/O behaviors of a production cosmology application that uses the adaptive mesh refinement (AMR) technique for its domain decomposition. This application was originally developed using Hierarchical Data Format (HDF version 4) I/O library and since HDF4 does not provide parallel I/O facilities, the global file I/O operations were carried out by one of the allocated processors. When the number of processors becomes large, the I/O performance of this design degrades significantly due to the high communication cost and sequential file access. In this work, we present two additional I/O implementations, using MPI-IO and parallel HDF version 5, and analyze their impacts to the I/O performance for this typical AMR application. Based on the I/O patterns discovered in this application, we also discuss the interaction between user level parallel I/O operations and different parallel file systems and point out the advantages and disadvantages. The performance results presented in this work are obtained from an SGI Origin2000 using XFS, an IBM SP using GPFS, and a Linux cluster using PVFS.

1. Introduction

Scientific applications like Adaptive Mesh Refinement (AMR) [1] simulations usually are not only computation intensive but also I/O intensive. Parallel computing and load balancing optimizations have greatly improved the computation performance by evenly partitioning the workload among a large number of processors. However, the I/O part does not usually fit well into this parallel computing environment, either due to the lack of parallel I/O libraries that can provide good performance or due to the complexity of the parallel I/O operations themselves.

ROMIO [15], an implementation of MPI-IO in MPI standard 2 [19] developed at Argonne National Laboratory, is one such library that can be used in the parallel computing environment. It provides a high-level interface supporting partition of file data among processors and an optimal I/O interface supporting complete transfers of global data structures between processor memories and files.

While the MPI-IO can significantly improve the performance of I/O operations in regular data access patterns, the irregular access patterns still need further consideration in order to achieve full efficiency, which is usually not directly provided by the parallel I/O library. In such case, it is necessary to review the data access patterns at the application level as well as the data storage patterns at the file system level. These metadata can be used to improve the overall I/O performance.

In this work, we examine the I/O access patterns of a production astrophysics application, named the ENZO cosmology simulation [2, 8]. ENZO was designed using AMR technique [1] and originally implemented using HDF version 4 [17] library to carry out its I/O operations. Since HDF4 does not provide parallel I/O facility, two alternatives were implemented and discussed in this paper: using MPI-IO and HDF version 5 [18]. The MPI-IO approach uses explicit file offsets and data sizes when accessing the files. HDF5 is a complete new work from HDF project that incorporated the parallel I/O interfaces. The underlying implementation of parallel HDF5 uses MPI-IO framework. We evaluate these two alternatives on several parallel file systems, discuss the interaction between the user-level I/O patterns and different parallel file systems and point out the advantages and disadvantages, which is beneficial for future research in parallel I/O systems as well as the I/O designs for AMR applications.

The rest of this paper is organized as follows. Section 2 gives an overview of AMR and the ENZO application. Section 3 examines ENZO's I/O structure and describes two alternative I/O approaches, using MPI-IO and HDF5. Section 4 presents the experimental performance results and the evaluation of different I/O optimizations. Finally, Section 5 draws conclusions and identifies the future work.

2. Application Overview

ENZO is a parallel, three-dimensional cosmology hydrodynamics application that simulates the formation of a

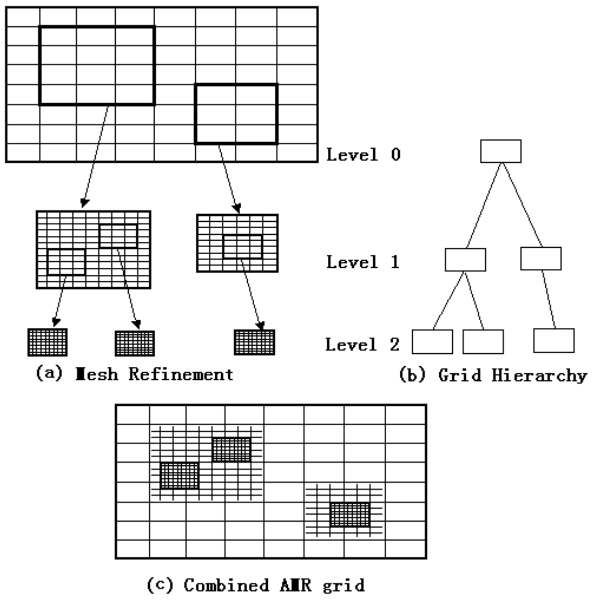

Figure 1. A 2-D AMR Grid Hierarchy.

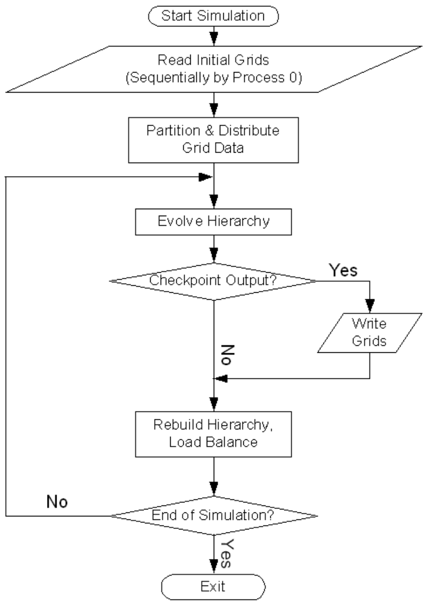
Start Simulation
Read Initial Grids (Sequentially by Process 0)
Partition & Distribute Grid Data
Evolve Hierarchy
Checkpoint Output?
Yes
Write Grids
No
Rebuild Hierarchy, Load Balance
End of Simulation?
No
Yes
Exit

Figure 2. ENZO Cosmology Simulation Flow.

cluster of galaxies consisting of gas and stars [2, 8]. The simulation starts near the beginning of the universe, a few hundred million years after the big bang, and normally continues until the present day. It is used to test theories of how galaxy and clusters of galaxies form by comparing the results with what is really observed in the sky today. Due to the nature that cosmic fluids, such as starts and dark matter, interact with each other mainly under the gravitational influence, the spatial distribution of the cosmic objects is usually highly irregular. If a static uniform mesh is used in the domain decomposition, the tradeoff will exist between increasing the resolution using finer mesh and reducing the computation costs using more coarse mesh. Adaptive mesh refinement (AMR) [1] has become a well-known technique that can properly address the problems that require high spatial resolution in localized regions of multidimensional numerical simulations. In the implementation of the ENZO cosmology simulation, AMR is used throughout the evolution process so that the high resolution grids can be adaptively placed where the condense stars and gas locate.

2.1. Adaptive Mesh Refinement

Adaptive Mesh Refinement (AMR) is a type of multiscale algorithm that achieves high spatial resolution in localized regions of dynamic, multidimensional numerical simulations [2, 8]. At the start of a simulation, a uniform mesh (root grid or top-grid) covers the entire computational domain, and in regions that require higher resolution, a finer subgrid is added. If more resolution is needed in a subgrid, an even finer mesh is added in a deeper level. That way, the AMR algorithm produces a deep, dynamic hierarchy of increasingly refined grid patches, which are organized in a tree-like grid hierarchy structure as shown in Figure 1. Each grid in this hierarchy has various sizes, shapes and spatial resolutions. The parent grids combine the results from the child grids at each refine level and the root grid represents the simulation results of the entire computational domain.

2.2. Simulation Flow and Data Structure

ENZO simulates the formation of the star, galaxy, and galaxy cluster by evolving the initial grid and taking “slapshots” — data dump of intermediate state of grids — during each cycle of evolutions until it reaches some end state. Figure 2 shows the structure of the code. In ENZO, the AMR grid contains two major types of data: the baryon field data that uniformly samples the grid domain for various kinds of cosmological properties, and particle data that represents the properties of a set of particles within the grid domain.

In ENZO’s implementation, the parallelism is achieved by partitioning the problem domain (the root grid) among processors and each processor starts from a sub-domain of the root grid and comes out with many subgrids in different refinement levels. Due to the implementations of AMR and load balance optimization [5, 6], the subgrids can be refined and redistributed among processors. An example of resulted hierarchy and distribution of the grids is illustrated in Figure 3.

The original ENZO application uses sequential HDF4 I/O library to perform file I/O. To start a new simulation, the

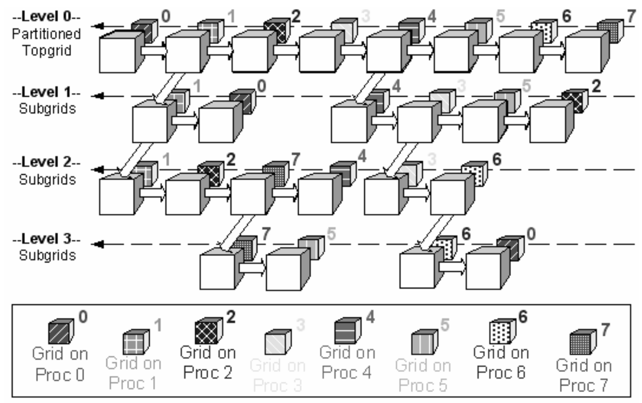

Figure 3. Example of Grid Hierarchy and Grid Distribution using 8 processors. The hierarchy data structure is maintained on all processors and contains grids metadata. Each node of this structure points to the real data of the grid. The grids themselves are distributed among processors.

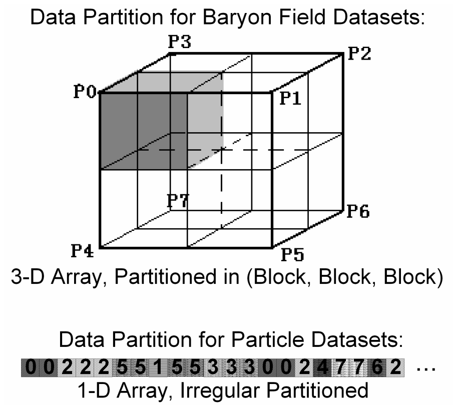
Data Partition for Baryon Field Datasets:
P0
P1
P2
P3
P4
P5
P6
P7
3-D Array, Partitioned in (Block, Block, Block)
Data Partition for Particle Datasets:
0 0 2 2 2 5 5 1 5 5 3 3 3 0 0 2 4 7 7 6 2 ...
1-D Array, Irregular Partitioned

Figure 4. ENZO Data Partition Patterns on 8 processors. Baryon field datasets are 3-D arrays and partitioned in a (Block, Block, Block) manner. Particle datasets are 1-D arrays and partitioned in an irregular pattern (by particle position).

application reads in some initial grids(root grid and some initial pre-refined subgrids) and partitions them among processors. During the evolution of the grid hierarchy, the application periodically writes out the intermediate grid data (a checkpoint data dump). The output files from the checkpoint data dump are used either for restarting a resumed simulation or for visualization.

3. I/O Analysis and Optimizations

In this section, we first examine the file I/O behaviors in the ENZO code and, then, analyze its data partition and access patterns. Useful metadata that can be used to potentially improve the I/O performance is also studied. Furthermore, different I/O approaches, including MPI-IO and parallel HDF5, are presented and investigated for their potential benefits to the overall I/O performance.

3.1. Application I/O Analysis

The file I/O operations in ENZO cosmology simulation are grid based. Reading/Writing a grid includes the read/write of the baryon field data containing a number of 3-D arrays (density, energy, velocity X, velocity Y, velocity Z, temperature, dark matter, etc.) and the particle data containing a number of 1-D arrays (particle ID, particle positions, particle velocities, particle mass, and other particle attributes).

There are three categories of I/O operations: reading the initial grids in a new simulation, writing grids during each cycle of grid hierarchy evolution and at the end of a simulation, and reading grids in a restart simulation.

In a new simulation, processor 0 reads in all initial grids including the top-grid and some pre-refined subgrids. Each grid is, then, evenly partitioned among all processors based on the spatial domain boundaries in each dimension by having processor 0 redistributing the grid data to all other processors. The data partition pattern of the 3-D baryon fields is in (Block, Block, Block) according to the partition of the grid domain. The 1-D particle arrays are partitioned based on which grid sub-domain the particle position falls within and, therefore, the particle access pattern can be highly irregular. Figure 4 illustrates the partition patterns of different data arrays. When reading a grid, a processor follows a fixed order of accessing the various data arrays of the grid.

When performing the checkpoint data dump, the partitioned top-grids are collected by processor 0 and combined into a single top-grid that is written to a single file by processor 0. During the top-grid combination, the particles and their associated data arrays are sorted in the original order in which the particles were initially read. When performing the data dump for subgrids, each processor writes its own subgrids into individual grid files. In this case, the write operations can be performed in parallel without communication.

The restart read is pretty much like the new simulation read, except that every processor reads the subgrids in a round-robin manner. Similarly, the access of a grid follows a fixed order in the series of arrays.

When analyzing the I/O characteristics of the ENZO simulation, several useful metadata are discovered: the rank

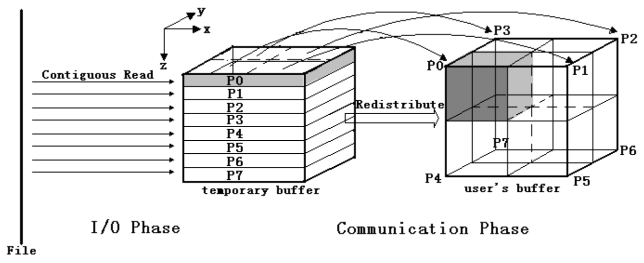

Figure 5. Reading (Block, Block, Block)-distributed subarrays from a 3-D array by using collective I/O. The 3-D array is stored in the file such that x-dimension is the most quickly varying dimension and z-dimension is the most slowly varying dimension. The write operation is similar except that the first phase of the two-phase operation is communication and the second phase is I/O.

and dimensions of data arrays, the access patterns of arrays, and the data access order. With the help of these metadata, the proper optimal I/O strategies can be determined for the AMR type of applications and the high performance parallel I/O can be achieved.

3.2. Parallel I/O Implementation Using MPI-IO

MPI-IO is part of MPI Standard 2 [19], a message passing interface specification implemented and used on a wide range of platforms. In this work, ROMIO [15], the most popular MPI-IO implementation is used in our I/O implementation. It is implemented portably on top of an abstract I/O device (ADIO) layer [13] that enables ROMIO to be ported to new underlying I/O systems. ROMIO is freely available for a wide range of systems including IBM SP, Intel Paragon, HP Exemplar, SGI Origin2000, NEC SX-4, and clusters of workstations or PC Linux.

One of the most important features in ROMIO is the collective I/O operations which adopts the two-phase I/O strategy [9, 11, 12, 14]. Figure 5 illustrates the idea of collective I/O. The collective I/O operations are decomposed into two phases, the I/O phase and communication phase. Taking the collective read as an example, processors, in the I/O phase, read data that is conformed to the distribution patterns in the file, which results in each processor making a single, large, contiguous access. In the communication phase, processors redistribute data among themselves towards the desired data distribution patterns in processor memories. This approach improves the I/O performance by significantly reducing the number of I/O requests that would otherwise result in many small non-contiguous I/O requests.

3.2.1. Optimizations by Access Pattern

As we have seen, the file I/O for the top-grid in the ENZO application involves two categories of access patterns, regular and irregular access patterns.

When accessing the 3-D arrays of the baryon field data in (Block, Block, Block) partition pattern, each processor needs to access a large number of non-contiguous blocks of data from the array. By using the collective I/O features of the MPI-IO along with the MPI derived datatypes, we can view the file data as a 3-D array, and the requested part of dataset as its subarray defined by the offsets and lengths in each dimension. After setting the file view for the requested subarray in each processor, all processors perform a collective I/O in which internally each processor accesses a contiguous data region in the file and the desired pattern is achieved by inter-communication among the processors participating the collective I/O, as described in Figure 5.

In the irregular access pattern presented in the ENZO implementation, the 1-D particle arrays can be accessed by using block-wise I/O followed by data redistribution based on the grid location. The read access is done by first performing contiguous read in a block-wise manner, checking the particle position and grid edges to determine which processor each particle belongs to, and then performing inter-processor communication to redistribute the particles to their destination processors. The write access is a little bit different since the particle arrays need to be globally sorted by the particle ID. To perform a parallel write for particle data, all processors perform a parallel sort according to the particle ID and then all processors independently perform block-wise MPI write. The MPI-IO operation here is non-collective because the block-wise pattern for 1-D arrays always results in contiguous access in each processor.

3.2.2. Making Use of Other Metadata

Note that each grid access involves the data arrays one by one in a fixed order. We can further optimize the I/O performance by letting all processors write their subgrids into a single shared file. Writing all grids into a single file can benefit the read performance when the simulation restarts. When data size becomes very large and needs to migrate to a tape device, writing grids into a single file can result a contiguous storage space in a hierarchical file system which will generate an optimal performance for data retrieval. In this work, we consider writing all grids into a single file for our parallel I/O implementations.

3.3. Parallel I/O Implementation Using HDF5

Since HDF5 provides parallel I/O interfaces, we expect performance improvement by changing the original HDF4

I/O in the ENZO code into parallel HDF5 I/O. In our implementation, we set the HDF5 to use MPI-IO for parallel I/O access. Similar to our previous implementation directly using MPI-IO, we also make use of the data access patterns and let all processors read/write the top-grid in parallel (collective I/O for regular partitioned baryon field data and non-collective I/O for irregular partitioned particle data). Similarly, the initial subgrid is read in the same way as the top-grid, and all subgrids are written to a single file shared by all allocated processor in parallel. However, the parallel access of a data array (dataset) is achieved by selecting a hyperslab from the file dataspace of that dataset, instead of using fileview and subarray.

4. Performance Evaluation

In this work, we run our experiments on different parallel systems, each with a different parallel file system. We vary the numbers of processors and the problem sizes to see how well the I/O performance can benefit from the optimizations when using MPI-IO and HDF5. It is also interesting to see how the optimized parallel I/O operations interact with different parallel file systems. We use three different problem sizes: AMR64 with grid dimensionality of $32\times32\times32$, AMR128 with $64\times64\times64$ and AMR256 with $128\times128\times128$. Table 1 gives the amount of I/O performed with respect to the three problem sizes.

Table 1. Amount of data read/written by ENZO application with three 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

4.1. HDF4 I/O vs MPI-IO

4.1.1. Timing Results on SGI Origin2000

Our first experiment is run on SGI Origin2000 using XFS. This system has 48 processors with 12 Gbytes memory and 1290 Gbytes scratch disk space. In this experiment, we use problem size AMR64 and AMR128 and run the original HDF4 I/O application and our MPI-IO version on different number of processors.

As SGI Origin2000 uses cach-coherent non-uniform memory access (ccNUMA) distributed shared memory architecture that employs bristled fat hypercube network for high bisection bandwidth, low-latency interconnect and because of its optimizations in remote and local memory latency as well as in data locality, the communication overhead is relatively low for MPI-IO, especially for those access patterns that involve many small non-contiguous data access and hence a lot of communication implied. In such case, our optimization using MPI-IO does get much benefit. The I/O performance improvement of MPI-IO over HDF4 I/O is shown in Figure 6.

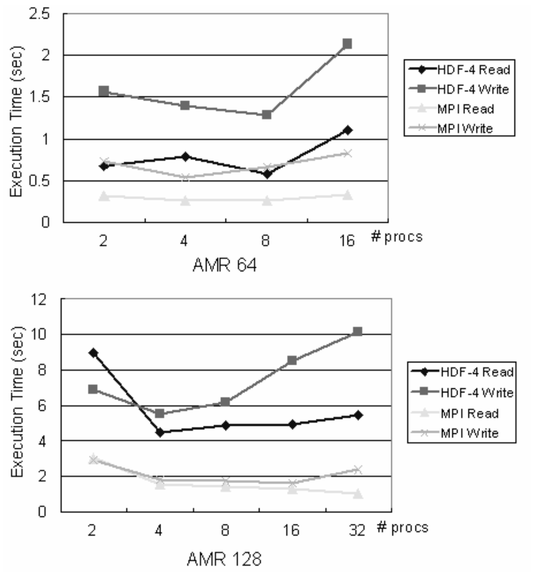

Figure 6. I/O Performance of the ENZO application on SGI Origin2000 with XFS.

4.1.2. Timing Results on IBM SP-2

The second experiment is done on an IBM SP-2 using GPFS. This system is a teraflop-scale Power3 based clustered SMP system from IBM with 144 compute nodes. Each compute node has SMP architecture with 4 GBytes of memory shared among its 8 - 375 MHz Power3 processors. All the compute nodes are inter-connected by switches. The compute nodes are also connected via switches to the multiple I/O nodes of GPFS.

We run the experiments with two different problem sizes (AMR64 and AMR128) on 32 processors and 64 processors. As is shown in Figure 7, the performance of our parallel I/O using MPI-IO is worse than 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 processor may access small chunks of data while the physical distribution of the file on the disks is based on very large, fixed striping size. The chunks of data requested by one processor may span on multiple I/O nodes, or multiple processors

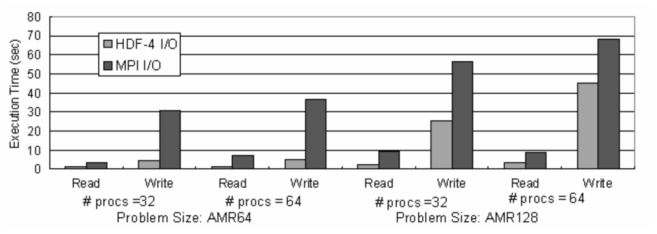

Figure 7. I/O Performance of the ENZO application on IBM SP-2 with GPFS.

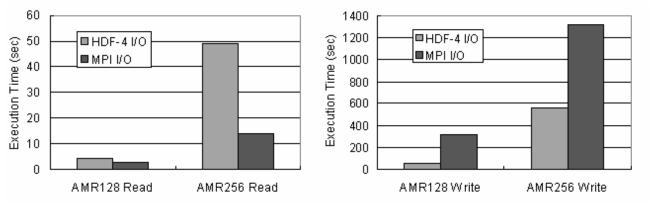

Figure 8. I/O Performance of the ENZO application on Linux cluster with PVFS (8 compute nodes and 8 I/O nodes inter-connected through fast Ethernet)

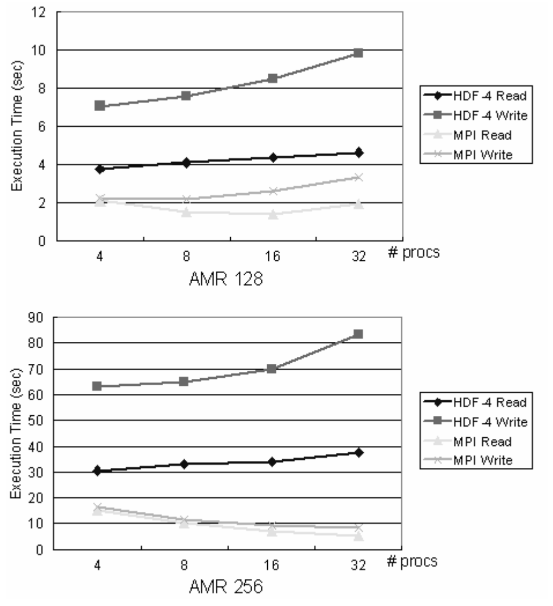

Figure 9. I/O Performance of the ENZO application on Linux cluster with each compute node accessing its local disk using PVFS interface.

may try to access the data on a single I/O node. Another reason for the parallel I/O performs worse on this system is that, in IBM SP, when too many processors on a single SMP node try parallel I/O access, the actual I/O may suffer from a long I/O request queue. But for larger problem size with proper more number of processors, as we see in the AMR128 case, this situation can be meliorated in some degree.

4.1.3. Timing Results on Linux Cluster

4.1.3.1. Using PVFS

The third experiment is obtained on the Linux cluster at Argonne National Laboratory, Chiba City [16], using PVFS [21]. Each of the compute nodes used in the experiment is equipped with two 500 MHz Pentium III processors, 512 Mbytes of RAM, 9 Gbytes of local disk, and 100 Mbps Ethernet connection with one another. This experiment is run on 8 compute nodes each with one process, and the PVFS is accordingly configured using 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 provides efficient utilization of disk space. For various types of access patterns, especially those in which each processor accesses a large number of small strided data chunks, there may be significant mismatch between the application partitioning patterns in memory and physical storage patterns in file. Due to the large communication overhead resulting from using the fast Ethernet in the current system configuration, the performance results shown in Figure 8 present the performance degradation of this overhead. However, the MPI read performance is a little better than HDF4 read because of the caching and ROMIO data-sieving techniques that overcome the communication overhead. From this figure, we discovered that the performance results tend to be better for larger size of problem, which is more unlikely having repeatedly accesses for small chunks of data.

4.1.3.2. Using Local Disk

Having noticed the significant performance degradation due to the high communication overhead between the I/O nodes and the compute nodes using fast Ethernet, we run the fourth experiment on the Chiba City using the disk local to the compute nodes. The I/O operations are performed through internally calling PVFS I/O interface. The only overhead of MPI-IO is the user-level inter-communication among compute nodes. The performance results are shown in Figure 9. As expected, the MPI-IO has much better over-

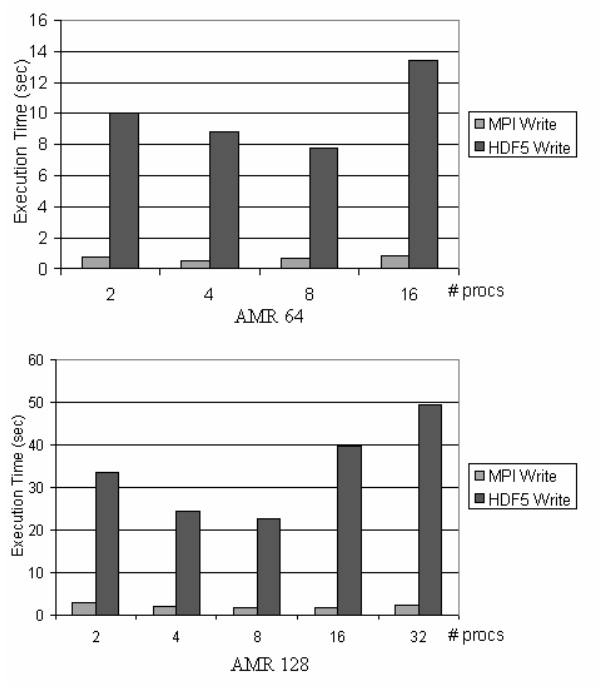

Figure 10. Comparison of I/O write performance for HDF5 I/O vs MPI-IO (on SGI Origin2000).

all performance than the HDF4 sequential I/O and it scales well with the number of processors.

However, unlike using the real PVFS which generates integrated files, the file system used in this experiment does not keep any metadata of the partitioned file and it requires additional efforts to integrate the distributed output files in order for other applications to use.

4.2. Evaluation of HDF5 I/O Performance

To evaluate the performance of HDF5 I/O for ENZO, we run our application on SGI Origin2000 with problem sizes AMR64 and AMR128, and compare the timing results for write operations with those of MPI-IO. At the time of our experiment, we use the official release version 5-1.4.3 of HDF5 from NCSA.

As seen in Figure 10, 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 in current release of HDF5 is very significant. First of all, synchronizations are performed internally in parallel creation/close of every dataset. Secondly, the HDF5 stores array data as well as the metadata in the same file, which can result in the real data ill alignment on appropriate boundaries. This design yields in a high variance in access time between processors. Thirdly, the hyperslab used for parallel access is handled as recursive operations in HDF5, which makes the packing of the hyperslab into a contiguous buffer relatively a long time. Finally, attributes (meta data) can only be created/written by processor 0, which also limits the parallel performance on writing real data. All these overheads lead to the worse performance of using parallel HDF5 I/O than using MPI-IO.

5. Conclusions and Future Work

This paper analyzes the file I/O patterns of an AMR application, and collects useful metadata that is used to effectively improve the I/O performance. These metadata includes the rank of arrays, the access pattern (regular and irregular), the access order of arrays. By taking advantages of two-phase I/O and other I/O optimization techniques, we develop useful I/O optimization methods for AMR simulations and show improvement of the I/O performance for such applications.

In the meanwhile, by testing different I/O approaches on different parallel file systems, we also find that the performance may not improved as much as we expected, due to the mismatch between the access patterns and the disk file striping and distribution patterns. There are other architectural reasons. On the IBM SP, the SMP configuration may result in I/O contentions and, on the Linux cluster, the overhead is due to the use of the slow communication network between compute and I/O nodes.

Our future work, on application level, includes using Meta-Data Management System (MDMS) [7] on AMR applications to develop a powerful I/O system with the help of the collected metadata. On lower level, this paper raises some important design issues of parallel file system, and the future work may be to improve the parallel file system so that it has flexible, application-specific disk file striping and distribution patterns that is convenient and efficient for MPI-IO applications.

Acknowledgements

This work is supported by National Computational Science Alliance under contract ACI-9619019 and by Department of Energy under the Accelerated Strategic Computing Initiative (ASCI) Academic Strategic Alliance Program (ASAP) Level 2, under subcontract No. W-7405-ENG-48 from Lawrence Livermore National Laboratories. We also acknowledge the use of SGI Origin2000 at NCSA, IBM SP-2 at SDSC and Linux cluster at ANL.

This work has been influenced by a number of people. First we are thankful to Greg Bryan for providing us the ENZO code and many helpful discussions on its I/O behaviors. We also thank Zhilin Lan and Xiaohui Shen for collaborations in the beginning part of this work.

References

[1] M. Berger and P. Colella, “Local Adaptive Mesh Refinement for Shock Hydodynamics”, *Journal of Computational Physics*, Vol. 82, No. 1, pp. 64-84, May 1989

[2] G. Bryan. “Fluid in the universe: Adaptive mesh refinement in cosmology”. In *Computing in Science and Engineering*, 1(2):46-53, March/April, 1999.

[3] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. “PASSION: Parallel And Scalable Software for Input-Output”, *Technical Report SCCS-636, NPAC, Syracuse University*, September 1994

[4] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*, MIT Press, Cambridge, MA, 1999

[5] Z. Lan, V. Taylor, and G. Bryan. “Dynamic Load Balancing For Structured Adaptive Mesh Refinement Applications”, in *Proceeding of ICPP'2001*, Valencia, Spain, 2001.

[6] Z. Lan, V. Taylor, and G. Bryan, “Dynamic Load Balancing of SAMR applications on Distributed Systems”, in *Proceeding of SC'2001 (formerly known as Supercomputing)*, Denver, CO, November, 2001.

[7] W. Liao, X. Shen, and A. Choudhary. “Meta-Data Management System for High-Performance Large-Scale Scientific Data Access”, in the *7th International Conference on High Performance Computing*, Bangalore, India, December 17-20, 2000

[8] M.L. Norman, J. Shalf, S. Levy, and G. Daues. “Diving deep: Data management and visualization strategies for adaptive mesh refinement simulations”, *Computing in Science and Engineering*, 1(4):36-47, July/August 1999

[9] J.M. Rosario, R. Bordawekar, and A. Choudhary. “Improved Parallel I/O via a Two-phase Run-time Access Strategy”, *IPPS '93 Parallel I/O Workshop*, February 9, 1993

[10] H. Taki and G. Utard. “MPI-IO on a parallel file system for cluster of workstations”, in *Proceeding of the First IEEE International Workshop on Cluster Computing*, 1999

[11] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. “PASSION Runtime Library for Parallel I/O”, *Scalable Parallel Libraries Conference*, Oct. 1994

[12] R. Thakur and A. Choudhary. “An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays”, *Scientific Programming*, 5(4):301-317, Winter 1996

[13] R. Thakur, W. Gropp, and E. Lusk. “An Abstract-Device interface for Implementing Portable Parallel-I/O Interfaces”(ADIO), in *Proceeding of the 6th Symposium on the Frontiers of Massively Parallel Computation*, October 1996, pp. 180-187

[14] R. Thakur, W. Gropp, and E. Lusk. “Data Sieving and Collective I/O in ROMIO”, in *Proceeding of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999, pp. 182-189

[15] Rajeev Thakur, Ewing Lusk, and William Gropp, “Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation”, Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Revised July 1998.

[16] Chiba City, the Argonne National Laboratory scalable cluster. *http://www.mcs.anl.gov/chiba.*

[17] HDF4 Home Page. The National Center for Supercomputing Applications. *http://hdf.ncsa.uiuc.edu/hdf4.html*.

[18] HDF5 Home Page. The National Center for Supercomputing Applications. *http://hdf.ncsa.uiuc.edu/HDF5/*.

[19] Message Passing Interface Forum. “MPI-2: Extensions to the message-passing interface”, July 1997. *http://www.mpi-forum.org/docs/docs.html*

[20] Pablo Research Group. “Analysis of I/O Activity of the ENZO Code”, University of Illinois at Urbana-Champaign, July 2000. *http://www-pablo.cs.uiuc.edu/Data/enzo/enzoData.htm*

[21] The Parallel Virtual File System Project. *http://parlweb.parl.clemson.edu/pvfs/*