Performance Emulation of Cell-based AMR Cosmology Simulations

Jingjin Wu*, Roberto E. González†, Zhiling Lan*, Nickolay Y. Gnedin†‡,
Andrey V. Kravtsov†, Douglas H. Rudd§, Yongen Yu*
*Dept of Computer Science, Illinois Institute of Technology, Chicago, IL
{jwu45,lan,yyu22}@iit.edu
†Dept of Astronomy & Astrophysics, The University of Chicago, Chicago, IL
{regonzar,andrey}@oddjob.uchicago.edu
‡Theoretical Astrophysics Group, Fermi National Accelerator Laboratory, Batavia, IL
gnedin@fnal.gov
§Yale Center for Astronomy & Astrophysics, Yale University, New Haven, CT
douglas.rudd@yale.edu

***Abstract*—Cosmological simulations are highly complicated, and it is time-consuming to redesign and reimplement the code for improvement. Moreover, it is a risk to implement any idea directly in the code without knowing its effects on performance. In this paper, we design an emulator for cell-based AMR (adaptive mesh refinement) cosmology simulations, in particular, the Adaptive Refinement Tree (ART) application. ART is an advanced "hydro+N-body" simulation tool integrating extensive physics processes for cosmological research. The emulator is designed based on the behaviors of cell-based AMR cosmology simulations, and quantitative performance models are built toward the design of the emulator. Our experiments with realistic cosmology simulations on production supercomputers indicate that the emulator is accurate. Moreover, we evaluate and compare three different load balancing schemes for cell-based cosmology simulations via the emulator. The comparison results provide us useful insight into the performance and scalability of different load balance schemes.**

***Keywords*-performance emulator; cosmology simulation; adaptive mesh refinement; load balancing**

I. Introduction

Numerical simulations are vital for many scientific inquiries, especially in those fields where researchers cannot build and test models of complex phenomena in the laboratory. Cosmology is such a field where scientists cannot perform direct experiments with the objects of their study. Instead, they have to build up numerical simulations to model the universe. As a result, cosmology simulations are critical to making quantum leaps in understanding the formation and evolution of galaxies in the universe. To effectively model various regions with different densities in the universe, adaptive mesh refinement (AMR) [1] is widely adopted in cosmology simulations, which enables high resolution in localized regions (e.g., the regions around the galaxies of interest). There are mainly two different numerical approaches towards the implementation of AMR-based cosmology simulations, namely block-structured [2] and cell-based [3], each having their advantages and drawbacks.

As cosmologists are always making efforts to enhance cosmological codes, such as fully utilizing the ever-growing supercomputers for high-fidelity simulations and including new physical processes especially those that are ignored before, cosmology simulations become increasingly complex. Improving cosmology codes is time-consuming and complicated. Without rigorous performance analysis before making major code modification, it is very likely that the expected performance improvement is not achieved or even worse. Thus, it is critical to provide a new approach to examine potential performance impacts with different code changes in a time-efficient way.

We target cell-based AMR cosmology applications, in particular, the adaptive refinement tree (ART) code [3]. As an advanced cosmology simulation tool, ART is currently an "N-body+hydro" simulation code integrating both dark matter and gas dynamics, as well as radiative transfer. Many physical calculations have been implemented for the first time in the ART code, making it unique in its capabilities. In a typical cosmology simulation, the highest resolution regions can reach 7 to 10 refinement levels, resulting in a large range of dynamic multidimensional regions. For simulations with highly non-uniform grids, how to achieve good performance at scale is very challenging.

In this paper, we present our design of a performance emulator for cell-based AMR cosmology simulations. The emulator follows the flow of the original application, while the major physical computation and interprocess communication are replaced by runtime performance estimates provided by the emulator. We demonstrate the effectiveness of the emulator by means of realistic cosmology simulation data on production systems. Our experiments indicate that the emulator achieves good accuracy. Given the dynamic feature of AMR and the wide range of workload per cell, load balancing is an extremely challenging problem for cell-based cosmology simulations. In this paper, we further evaluate three load balancing schemes for cell-based AMR cosmology simulations via the performance emulator. The use of the emulator enables us to quickly identify the issues associated with different load balancing schemes.

The remainder of this paper is organized as follows. Section II introduces the background information of AMR and the ART code. Section III presents the design of ART performance emulator. Section IV describes three load balancing schemes that are evaluated in this study. Section V presents our experiments on realistic cosmology data including the assessment of the emulator and the comparison of different load balancing schemes via the emulator. Finally, Section VI concludes this paper.

II. Background

In this section, we introduce AMR and cell-based AMR, and present an overview of the ART code.

A. AMR Algorithm

The adaptive mesh refinement (AMR) algorithm was proposed by Berger et al. in 1989 [4]. It is a type of multiscale algorithm that achieves high spatial resolution in localized regions for dynamic and multidimensional numerical simulations. Its basic principle is straightforward. Initially, a uniform mesh is adopted for the entire computational domain. In the regions which require higher resolution, finer subgrids are added. If some regions still need more resolution, even finer subgrids are added. The mesh is refined recursively in this way, and turns into a tree of grids. This uniform mesh, as the tree's root, is at the top level. Each finer level decreases the mesh size by a factor r, which is defined as the *refinement factor*. This approach enables user to solve problems which are completely intractable on a uniform mesh, and it is efficient when only a small part of the computational domain needs to be refined to the highest resolution.

There are mainly two different approaches for implementing cosmology AMR applications: block-structured AMR [2] and cell-based AMR [3]. The former achieves high spatial resolution by inserting smaller grids ("blocks") at places where high resolution is needed. The latter instead refines the computational domain on a cell-by-cell basis. AMR techinique enables cosmologists to track the physical processes at very small scale. In practice, these two methods use different data structures and very different methods for distributing the computational load across a large number of processes. The cosmology applications based on block-structured AMR, e.g., Enzo [5], have been extensively studied in [6], [7] and [8]. However, there is little work on performance study of cell-based AMR cosmology applications. The ART code, as our main simulation tool, is a representative of such cell-based AMR applications.

B. Cell-based AMR

The cell-based AMR implements the AMR algorithm by performing grid refinement based on each cell. If higher spatial resolution is required for a cell, then it is refined into smaller cells, which are at the finer level of the overall grid hierarchy. With each level up, the cell size is decreased by the refinement factor r. Figure 1 shows a cell-based AMR example on the 2D mesh, including the grid hierarchy and the overall grid structure. For simplicity, there are only 3 levels of cells from level 0 to level 2. Initially, a uniform grid on level 0 covers the overall computational domain. The dotted cells need higher resolution, so they are further refined. Throughout the execution of the AMR application, the grid hierarchy changes adaptively, and the cells are organized in refinement trees [9]. In the refinement tree, the cell with children is a *non-leaf cell*. Otherwise, it is a *leaf cell*.

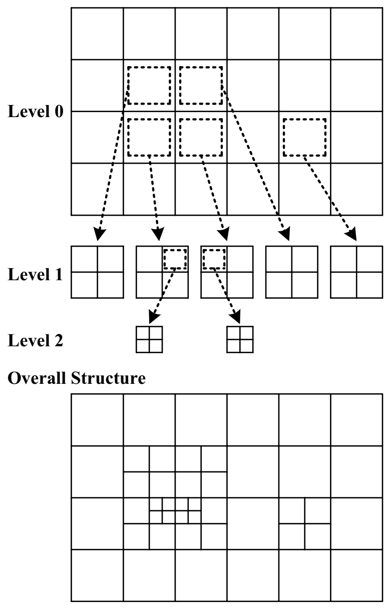

Figure 1. A 2D cell-based adaptive mesh refinement example: a quad tree with refinement factor = 2.

C. Cosmology Simulation Code ART

ART is a hybrid "MPI+OpenMP" C code, with Fortran functions for computationally-intensive routines. The MPI parallelization is used between separate computation nodes and the OpenMP parallelization is used inside a multi-core node. This mixed mode parallelization enables us to take full advantage of modern multi-core architectures. The ART code employs the cell-based AMR algorithm, performs refinement locally on individual cells and organizes cells in refinement trees. In order to model the universe, it adopts a cubic computational volume with a refinement factor of 2. For each cubic cell, the refinement operation evenly subdivides the cell into 8 cells, namely an *oct*. The refinement tree is also called *oct-tree*. With cell-based AMR, the ART code is able to control the computational mesh on the level of individual cells, such that the refinement mesh can easily be built and modified, and therefore, can effectively match the complex geometry of cosmologically interesting regions.

Figure 2 shows the basic flow of the ART code. First, it reads input files, including parameter files and cosmology data. Second, it initializes oct-tree and cell buffer. Then it checks whether the simulation time reaches the user specified time limit. If yes, the simulation stops, otherwise the ART code performs load balance and simulates another iteration by evolving time steps for the overall computational

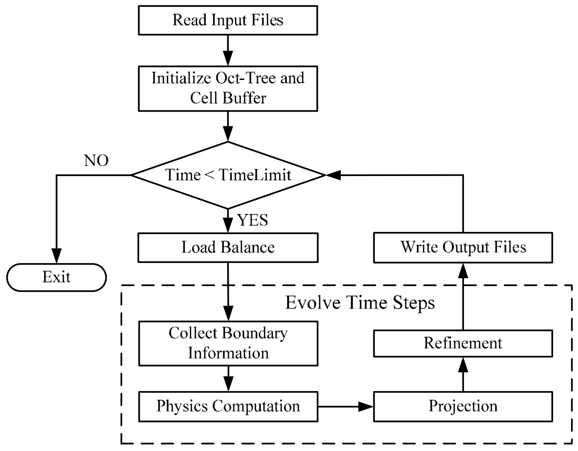

Figure 2. Flow control of the ART code.The four steps in the dotted region are the major steps for evolving a time step at a given level.

domain, including the cells at all the levels. For each level, the evolution of a time step mainly consists of four steps: collect boundary information, perform physics computation, project physics data to the coarser level, and adaptively refine/derefine the cells. At the end of each iteration, the ART code generates output files, including log files and cosmology data files if any.

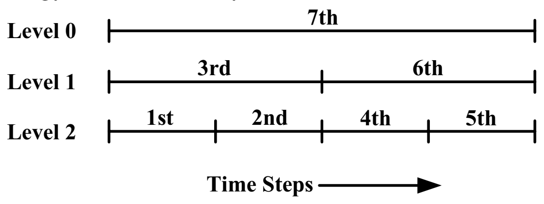

Figure 3. The order for evolving time steps on different levels (refinement factor = 2).

Specifically, in each iteration of the simulation, ART evolves a time step dt for the overall computational domain, which is adaptively refined into multiple levels of cells during simulation. This refinement in the spatial domain is accompanied by a refinement in the time domain, where finer level grids or cells evolve with a smaller time step according to the refinement factor. Since the ART code uses a refinement factor of 2, the time step size at level ℓ is $2^{-\ell}dt$. As a result, ART evolves 2^{ℓ} time steps at level ℓ in each iteration. Figure 3 shows the recursive execution order for evolving these time steps on three levels. Basically, finer levels evolve first, then coarser levels follow. Except the finest level, each level ℓ evolves a new time step when and only when level $\ell+1$ has evolved two time steps ahead.

At the beginning of each time step, each process collects boundary information by exchanging boundary data with other processes through MPI communication. In practice, each process first posts non-blocking receives (MPI_Irecv) to the process that it receives data from, and then sends data to other processes by non-blocking sends (MPI_Isend), and finalizes the communication by an MPI_Waitall for all processes. Such MPI communications are performed per time step at each level, and cannot be overlapped with computations because of data dependency.

III. Performance Emulation

The execution time of a cosmology simulation can be divided into three parts: physics computation, MPI communication, and others. Here, physics computation time includes the runtime spent on solving physics equations and managing the cells; the MPI communication time is the time spent on MPI function calls; and the other runtime mainly includes I/O and load balance time. As the sum of physics computation and MPI communication time accounts for more than 95% of the overall runtime, we focus on these two parts during the design of the emulator. In the following, we first present our performance models to estimate physics computation time and MPI communication time, and then describe the performance emulator.

A. Performance Models

Runtime performance models are built to estimate physics computation time and MPI communication time. Considering that different levels have different resolutions and time step sizes, our focus is to build models to estimate these runtimes of each time step per level. During simulation, each application process stores two types of cells: the cells within its computational domain, namely *local cells*; and the external neighboring cells of its computational domain, namely *buffer cells*. Each process conducts physical calculation on local cells, and communicates with other processes to get buffer cell data, which serves as important boundary information for simulating its local computational domain. These two types of cells are the main indicators of physics computation time and MPI communication time, respectively.

1) Physics Computation Time: In order to characterize the physics computation time accurately, we use principle component analysis (PCA) [10] to analyze the runtime, and observe that the number of local cells and particles are the dominant terms in determining the physics computation time. It is expected because the ART code solves two kinds of physics equations at each level: the physics equations for hydro dynamics, and the physics equations for N-body simulation. The former is only solved for leaf local cells, and the solution is projected to obtain the solution of non-leaf local cells at the coarser level, while the latter is solved for particles. Thus, we use the following linear model for the physics computation time T_P of each level.

$$\begin{aligned} T_P &= T_P^{nllc} + T_P^{llc} + T_P^{particle} \qquad (1)\\ &= w_1 \times N_{nllc} + w_2 \times N_{llc} + w_3 \times N_{particle}, \end{aligned}$$

where w_i $(i = 1, 2, 3)$ are constant coefficients for the level of interest. T_P^{nllc}, T_P^{llc} and $T_P^{particle}$ denote the computation time of non-leaf local cells, leaf local cells and particles, respectively. N_{nllc}, N_{llc}, and $N_{particle}$ denote the number of non-leaf local cells, leaf local cells and particles, respectively. Note that Eq.(1) is defined for each level of each process, while all the processes share common constant coefficients of each level.

To extract these coefficients, we use the cell and particle counts along with the physics computation time at each level of each process to formulate Eq.(1), and then solve a linear system in the form of $\mathbf{Ax} = \mathbf{b}$ for each level. As the number of equations is usually larger than the number of coefficients, we apply linear regression to compute the least square fit solution of these coefficients.

2) MPI Communication Time: The MPI communication time can be further divided into two parts: *data transmission time* and *synchronization time*.

Data transmission time is simply the runtime spent on transmitting data between processes. As updating the boundary information is a major communication routine and the amount of boundary data for each process is proportional to its buffer cell counts, the data transmission time is largely dependent on the number of buffer cells. In our model, data transmission time is modeled as follows.

$$T_{trans} = t_s + n \times t_c, \qquad (2)$$

where t_s can be considered as the latency for message passing, t_c is the inverse of the bandwidth and n is the data size for one time data transmission. The latency and bandwidth can be obtained by using Intel MPI Benchmarks (IMB) [11], and the number of transmitted bytes can be calculated using the number of buffer cells.

Synchronization time is incurred when processes do not start their communication routines simultaneously. For example, consider two processes with a maximum refinement level of 3 and 6, respectively, and assume that these two processes need to exchange boundary information from level 0 to 3. The first process starts evolving time steps at level 3, while the second process starts at level 6. Once the first process finishes a time step at level 3, it sits idle until the second process catches up to it, and then they can communicate to exchange boundary information. Even if two processes have the same refinement level, when there exists load imbalance, a process may still have to sit idle waiting for the boundary information from the other process. Such extra waiting time is recorded as part of the MPI communication time, because the process is stalled when executing MPI function calls, and we refer to it as *synchronization time*. The MPI communication time including both data transmission time and synchronization time can be characterized by emulating time steps as detailed in the next subsection.

To build these models, we need performance data so as to extract model coefficients. In practice, we can either use the performance data from previous simulations, or conduct a few iterations of the simulation to collect performance data for model construction.

B. Emulator Design

Figure 4 presents the design of our emulator. Comparing Figure 4 and Figure 2, we can see that the emulator uses the performance models to estimate physics computation and data transmission time for each time step, rather than conducting the actual simulation. During each iteration, we use

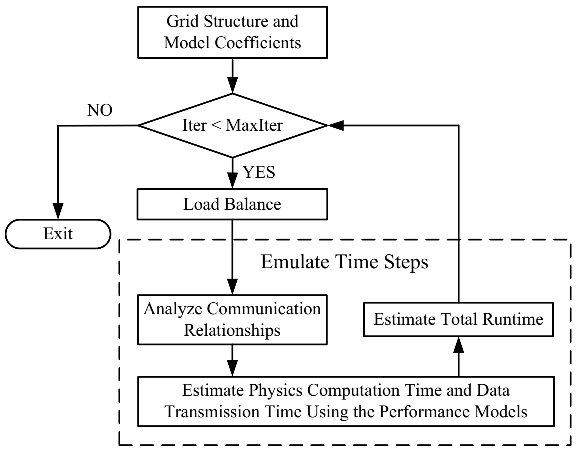
Grid Structure and Model Coefficients
Iter < MaxIter
NO
YES
Exit
Load Balance
Emulate Time Steps
Analyze Communication Relationships
Estimate Total Runtime
Estimate Physics Computation Time and Data Transmission Time Using the Performance Models

Figure 4. Flow of the performance emulator of ART.

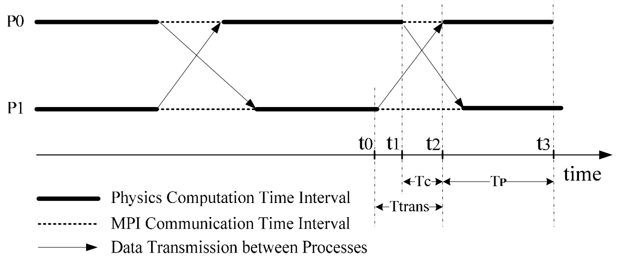
P0
P1
t0
t1
t2
t3
time
Tc
TP
Ttrans
Physics Computation Time Interval
MPI Communication Time Interval
Data Transmission between Processes

Figure 5. Part of the time axes of two processes.

a load balancer to determine workload distribution among processes, and then emulate time steps in exactly the same order as the ART code. For each time step, the emulator first analyzes the communication relationships among processes. Specifically, for each process, the emulator derives the amount of data that needs to be transmitted to and received from all the other processes. Next, according to the cell and particle counts of each process and the communication relationships, the emulator estimates physics computation time and data transmission time using the performance models shown in equation (1) and (2), respectively. Finally, the emulator estimates the total runtime. It is achieved by maintaining a time axis for each process to record the computation and communication intervals.

Figure 5 shows the time axes of two processes P0 and P1. The length of computation time intervals are the estimated physics computation times, while the arrows represent data transmission. Clearly, MPI communication time intervals are determined by computation time intervals and data transmissions. For example, the second MPI communication time interval of P0 can be computed as $T_c = t_2 - t_1 = (t_0 + T_{trans}) - t_1$, and the start time of P0 for the communication in the next time step is $t_3 = t_2 + T_P$. Therefore, using the estimated physics computation time and data transmission time for each time step of each process, the emulator is able to estimate the MPI communication time including both data transmission time and synchronization time without evaluating synchronization time separately.

In summary, by emulating time steps with the performance models to characterize runtime components, the emulator can estimate the performance of the ART code without executing physics solvers.

IV. Load Balancing Schemes

One of the major design goals of our performance emulator is to evaluate the performance of different load balancing schemes, thus avoiding time-consuming and complicated implementation in code without knowing potential effects of the modification. The ART code performs cosmology simulations using a cubic computational domain, which represents the universe. The computational domain is initially divided into many uniform cubic cells at level 0. We refer to such cells at level 0 as *root cells* since they are the roots of oct-trees. Each root cell keeps all its child cells at finer refinement levels as a single composite unit, thus being the basic unit for load balancing. In this section, we present three representative load balancing schemes which will be evaluated later in Section V.

A. SFC-Based Load Balancing (SFCLB)

Currently, the ART code employs a load balancing scheme based on Hilbert space-filling curve (SFC) [12]. We denote this scheme as SFCLB. It assigns a unique SFC ID for each root cell according to their spatial coordinates, then generates an SFC curve by connecting root cells with continuous SFC IDs, and finally divides the SFC curve into N_p (N_p is the number of processes) segments with similar amount of workload. Specifically, this scheme considers the total workload of each root cell, and adopts a greedy algorithm to split the SFC curve, so that the workload can be evenly distributed among all the processes. Currently, SFCLB is widely used for parallel AMR [13], [14], [15]. One salient feature of this SFCLB scheme is its good spatial locality, where each process gets root cells with continuous SFC IDs resulting in a continuous computational domain. However, SFCLB restricts the assignments of root cells to processes by the SFC curve, and does not consider the communication among processes when splitting the SFC curve.

B. Graph Partitioning-Based Load Balancing (GraphLB)

Graph partitioning is an alternative approach for the load balancing of cell-based AMR applications. We denote it as GraphLB. To apply the graph method, we need to map the load balancing problem into a graph partitioning problem. One straightforward mapping is to use vertices to represent root cells, and edges to represent communication relationships between neighboring root cells. The weight of each vertex is the workload of the corresponding root cell. Although this mapping does make sense, it results in a large graph, which is difficult to partition using acceptable amount of runtime and memory. For example, in our cosmology simulation, a cubic computational domain of 256^3 root cells maps into a graph with 256^3 vertices and about 6×256^3 edges. Partitioning such a large graph is almost prohibitive. Therefore, we must reduce the graph size for efficient partitioning. In our preliminary experiments, it is observed that only the root cells in a few localized regions are deeply refined to finer levels, while most root cells are not refined. Thus, we use a single vertex to represent the unrefined root cells with continuous SFC IDs, and create the edges accordingly in order to generate a manageable graph.

The assignments of root cells to processes can be obtained by partitioning the vertices in the graph into N_p partitions. Note that graph partitioning algorithms typically minimize the total edge-cuts subject to the constraints that the partitions are of similar size. When they are applied for the load balancing of cell-based AMR applications, we are actually minimizing the amount of communication among processes while ensuring that the workload is well-balanced. In our implementation, we use the graph partitioning tool METIS [17] to partition the graph.

C. Group-Based Load Balancing (GroupLB)

The aforementioned two load balancing schemes only consider the total workload of each root cell. However, as the ART code evolves time steps for each level, it is also critical to balance the workload of each level in order to reduce the synchronization time. Besides, it is observed that the communication at deep refinement levels usually results in large synchronization cost, so it is important to minimize the communication at deep refinement levels. To meet these requirements, we design a new load balancing scheme called GroupLB .

This scheme first assigns neighboring root cells into groups, where each group has the lowest possible boundary level and satisfies a set of group workload constraints to control the granularity. The assignment of root cells to groups is based on the Friends-of-Friends algorithm [18]. Second, GroupLB assigns root cell groups to processes by solving a constrained bin packing problem to balance the workload of each level, where each bin corresponds to a process. Specifically, we sort the groups in non-increasing order according to their workload, and pack them into bins sequentially. To achieve good spatial locality, we compute the distances between groups using Voronoi tessellations [19], and try to assign each group to the process which holds its neighboring groups. In this way, GroupLB is able to achieve good level-by-level load balance and spatial locality.

V. Experiments

In this section, we present two sets of experiments. In the first experiment, we assess the accuracy of the emulator. In the second experiment, we examine the load balancing schemes using the emulator with realistic cosmology data, and discuss their advantages and shortcomings.

A. Experimental Setup

We instrument the ART code with performance counters and timers for analysis. We use a real cosmological simulation of a box of 36 comoving Mpc on a side, covered with the uniform top level grid of 256^3 root cells. This simulation represents a scaled-down version of our future petascale cosmological simulations. In fact, the petascale

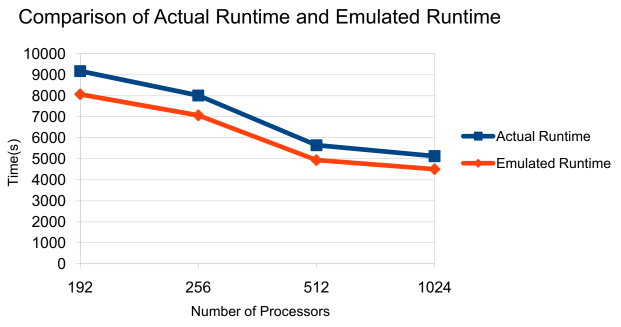

Figure 6. Comparison of actual runtime and emulated runtime of ART.

simulation is 100 times the volume of this dataset, and these 100 box pieces are essentially independent of each other.

Our testbed is the Intel 64 cluster *Abe* located at NCSA [20]. It is equipped with InfiniBand network and Lustre parallel file system. Each node has either 8GB or 16GB memory, and two quad-core CPUs running at 2.33 GHz. As ART is a hybrid "MPI+OpenMP" code and there are 8 processors available on each node of Abe, we assign an MPI process with 8 openMP threads to each node.

B. Accuracy of Performance Emulator

To measure the accuracy of the emulator, we conduct experiments using the emulator with the current load balancing scheme of ART – SFCLB, and compare the emulated performance results with the actual runtimes of the ART code. As the emulator emulates time steps without running physics solvers, it only consumes a few minutes to estimate the performance of ART.

Figure 6 presents the comparison of actual runtime and emulated runtime. Here, the runtime includes physics computation time and MPI communication time. The difference between the actual runtime and emulated runtime is within 12%. The error is mainly due to some extra communication routines in the ART code except for exchanging boundary information between processes. The emulator does not model such extra communications because they will be removed in our next version of ART. Therefore, the emulator is accurate. More importantly, we notice that both curves in the figure have exactly the same trend. This further indicates that our emulator can clearly predict the performance and scalability of realistic cosmology simulations, and the performance analysis based on this emulator is reliable.

C. Evaluation of Load Balancing Schemes

In this set of experiments, we assess three load balancing schemes presented in Section IV for cosmology simulations by means of the emulator. For the same cosmology dataset described in Section V-A, we test two different resolution cases: the *coarse resolution case* and the *fine resolution case*. The coarse resolution case represents a simulation with an intermediate resolution which reaches a maximum refinement level of 6. The fine resolution case represents a simulation with an extremely high resolution, which is allowed to refine dynamically to level 9.

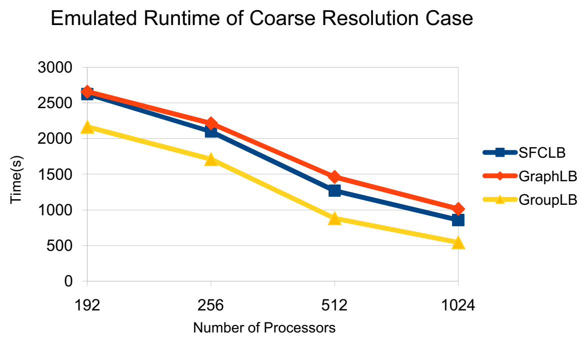

Figure 7. Comparison of emulated runtime by using different load balancing schemes for coarse resolution case.

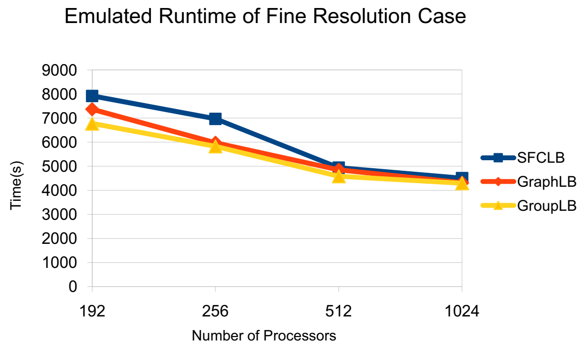

Figure 8. Comparison of emulated runtime by using different load balancing schemes for fine resolution case.

We use three metrics, namely execution time, load balance ratio, and communication time per level, for evaluation. Specifically, load balance ratio represents the quality of workload distributions among processes. It is defined as

$$Load\_Balance\_Ratio \triangleq \frac{\frac{1}{N_p}\sum_{i=0}^{N_p-1} W_i}{\max_{0\leq i\leq N_p-1} W_i} \times 100\%,$$

where W_i is the workload of process i and N_p is the number of processes. Note that *Load_Balance_Ratio* is always smaller than or equal to 100%, and a much closer value to 100% indicates a better load balance quality.

Figure 7 and 8 present emulated runtime by using different load balancing schemes for coarse resolution and fine resolution case, respectively. In both figures, the runtime using GroupLB scheme is smaller than that of the other two schemes, especially in the coarse resolution case. In Figure 8, we notice that as the number of processors increases, three curves are trending toward the same value. Such trend is caused by granularity, which is determined by the maximum workload of root cells. In the fine resolution case, there are several root cells whose individual workload is much larger than the average workload of each process when running on 512 and 1024 processors, thus introducing significant synchronization cost.

Table I presents the overall *Load_Balance_Ratio* among processes for different load balancing schemes. The closer the metric is to 100%, the better load balance is achieved.

Table I
OVERALL LOAD_BALANCE_RATIO OF DIFFERENT LOAD BALANCING SCHEMES

Number of	Coarse Resolution Case			Fine Resolution Case		
Processors	SFCLB	GraphLB	GroupLB	SFCLB	GraphLB	GroupLB
192	92.83%	64.89%	96.63%	90.21%	85.48%	91.17%
256	91.16%	73.54%	96.43%	71.35%	78.81%	88.51%
512	82.51%	53.76%	95.84%	53.74%	54.02%	51.73%
1024	70.49%	42.74%	92.21%	26.98%	27.01%	26.75%

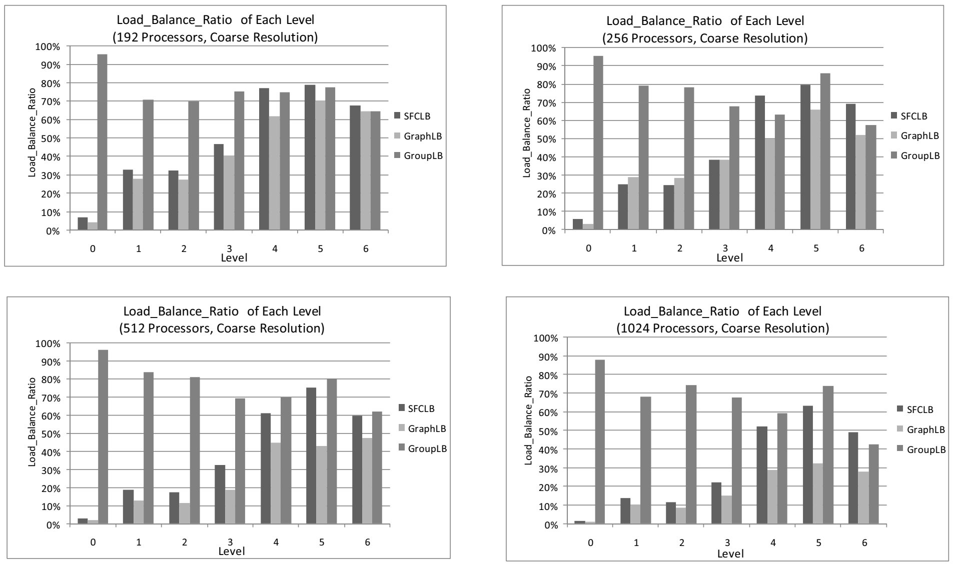

Figure 9. Load_Balance_Ratio of each level for coarse resolution case.

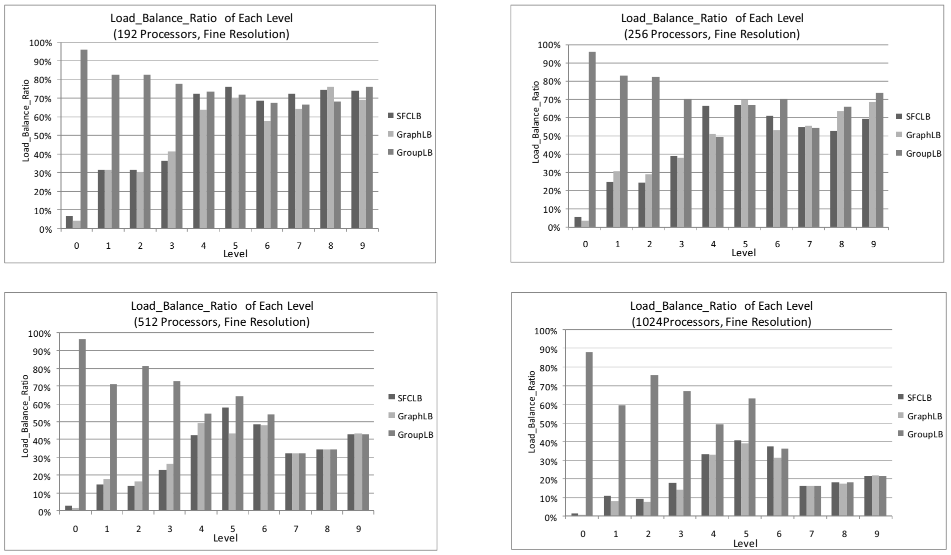

Figure 10. Load_Balance_Ratio of each level for fine resolution case.

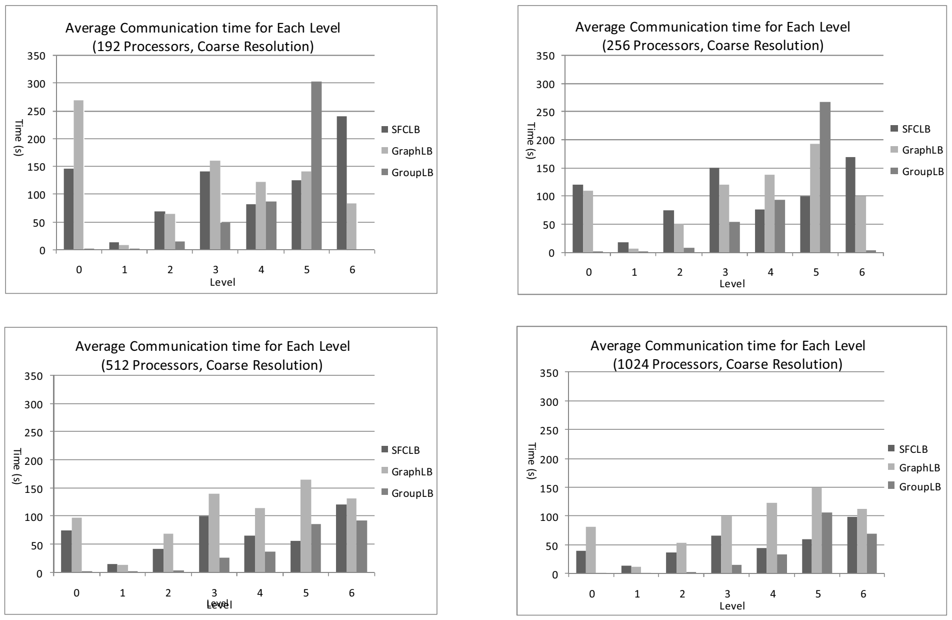

Figure 11. Average communication time of each level for coarse resolution case.

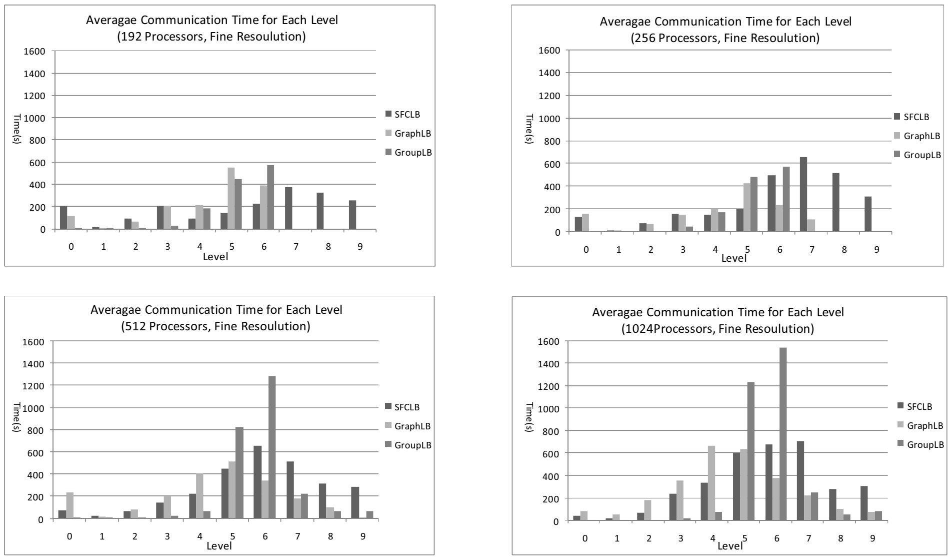

Figure 12. Average communication time of each level for fine resolution case.

Obviously, both SFCLB and GroupLB achieve better load balance for the coarse resolution case in comparison with the fine resolution case. This is because finer grid resolution increases the workload of root cells and results in larger granularity, which makes load balancing more challenging. GraphLB behaves differently since its primal objective is to minimize the communication among processes instead of balancing the workload. Comparing these schemes, GroupLB achieves the best load balance for the coarse resolution case and the first two tests of the fine resolution case. For the fine resolution case with 512 and 1024 processors, all of these three schemes fail to provide good load balance because of granularity problem.

Figure 9 and 10 show the *Load_Balance_Ratio* of each level for coarse resolution case and fine resolution case, respectively. In the coarse resolution tests, the *Load_Balance_Ratio* for almost all the levels of GraphLB is obviously smaller than that of the other two schemes; SFCLB and GroupLB provide similar load balance at refinement level 4 to 6; GroupLB also delivers well-balanced workload distribution at level 0 to 3. In the fine resolution tests, these schemes achieve comparable load balance at level 6 to 9, while GroupLB is more effective in balancing the workload at lower levels. In general, GroupLB achieves better level-by-level load balance than SFCLB and GraphLB for both coarse and fine resolution cases.

Figure 11 and 12 illustrate the average communication time of each level for the two resolution cases, respectively. In Figure 11, as the number of processors increases, the level-by-level communication time is decreasing. Generally, GroupLB introduces smaller level-by-level communication time than SFCLB and GraphLB because it achieves better load balance at each level. In Figure 12, the communication time is not scaling down with the increasing number of processors because the granularity problem results in poor load balance. GroupLB and GraphLB have much smaller communication time than SFCLB at level 7 to 9 since both methods try to reduce communication. However, for 512 and 1024 processors, GroupLB and GraphLB still have large communication time due to the granularity.

In summary, by comparing these three load balancing schemes, we conclude that GroupLB provides the best performance. It achieves a good load balance quality by balancing both overall and level-by-level workload, and minimizes communication cost by preserving spatial locality. While SFCLB maintains an overall load balance and keeps spatial locality using the SFC curve, it does not take into consideration of level-by-level load balance, thereby introducing non-trivial synchronization cost. Although GraphLB minimizes communication cost, it doesn't provide satisfactory load balance quality.

VI. Conclusion

In this paper, we have presented a performance emulator for cell-based AMR cosmology simulations. It uses performance models to characterize computation time and data transmission time, and emulates time steps to estimate performance without executing physics solvers. Experimental results on realistic cosmology dataset have demonstrated that the performance emulator is accurate. More importantly, it serves as an efficient tool to evaluate the performance of various load balancing schemes. We have studied three typical load balancing schemes using the performance emulator. Our experiments have indicated that GroupLB provides the best performance. This scheme not only achieves good load balance quality, but also minimizes communication cost by reducing synchronization and preserving spatial locality.

Acknowledgements

This work is supported in part by National Science Foundation grants OCI-0904670. Jingjin Wu is in part supported by China Scholarship Council.

References

[1] T. Plewa, T. Linde, and V. G. Weirs, *Adaptive Mesh Refinement – Theory and Applications*, Springer, 2005.

[2] G. L. Bryan, "Fluids in the universe: adaptive mesh refinement in cosmology," *Computing in Science & Engineering*, vol. 1, no. 2, pp. 46–53, Mar./Apr. 1999.

[3] A. V. Kravtsov, A. A. Klypin, and A. M. Khokhlov, "Adaptive refinement tree: a new high-resolution N-body code for cosmological simulations," *Astrophysical Journal Supplement*, vol. 111, p. 73, Jul. 1997.

[4] M. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of Computational Physics*, vol. 82, no. 1, pp. 64–84, May 1989.

[5] Enzo, http://lca.ucsd.edu/portal/software/enzo.

[6] Z. Lan, V. E. Taylor, and G. Bryan, "Dynamic load balancing for structured adaptive mesh refinement applications," in *Proc. Int. Conf. Parallel Processing (ICPP)*, 2001, pp. 571–579.

[7] Z. Lan and P. Deshikachar, "Performance analysis of large-scale cosmology application on three cluster systems," in *Proc. IEEE Int. Conf. Cluster Computing*, 2003, pp. 56–63.

[8] Z. Lan, V. Taylor, and Y. Li, "DistDLB: Improving cosmology SAMR simulations on distributed computing systems through hierarchical load balancing," *Journal of Parallel and Distributed Computing*, vol. 66, no. 5, pp. 716–731, May 2006.

[9] A. M. Khokhlov, "Fully threaded tree algorithms for adaptive refinement fluid dynamics simulations," *Journal of Computational Physics*, vol. 143, no. 2, pp. 519–543, 1998.

[10] J. Shlens, "A tutorial on principal component analysis."

[11] Intel MPI Benchmarks, http://software.intel.com/en-us/articles/intel-mpi-benchmarks.

[12] A. R. Butz, "Alternative algorithm for Hilbert's space-filling curve," *IEEE Trans. Computers*, pp. 424–426, Apr. 1971.

[13] Flash, http://flash.uchicago.edu/site/.

[14] J. Steensland, S. Chandra, and M. Parashar, "An application-centric characterization of domain-based SFC partitioners for parallel SAMR," *IEEE Trans. Parallel and Distributed Systems*, vol. 13, no. 12, pp. 1275–1289, Dec. 2002.

[15] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. Wilcox, "Extreme-scale AMR," in *Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis*, 2010, pp. 1–12.

[16] C. Burstedde, L. C. Wilcox, and O. Ghattas, "p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM J. Sci. Comput.*, vol. 33, no. 3, pp. 1103–1133, May 2011.

[17] METIS, http://glaros.dtc.umn.edu/gkhome/views/metis.

[18] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. M. White, "The evolution of large-scale structure in a universe dominated by cold dark matter," *Astrophysical Journal*, part 1, vo. 292, pp. 371–394, May 1985.

[19] G. Voronoi, "Nouvelles applications des paramètres continus à La Théorie de formes quadratiques," *Z. Reine Angew. Math*, vo. 134, pp. 198–287, 1908.

[20] NCSA Abe, http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64Cluster.