Scalable Cosmological Simulations on Parallel Machines

Filippo Gioachin[1], Amit Sharma[1], Sayantan Chakravorty[1], Celso L. Mendes[1], Laxmikant V. Kalé[1], and Thomas Quinn[2]

[1] Dept. of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{gioachin, asharma6, schkrvrt, cmendes, kale}@uiuc.edu
[2] Dept. of Astronomy, University of Washington
Seattle, WA 98105, USA
trq@astro.washington.edu

Abstract. Cosmological simulators are currently an important component in the study of the formation of galaxies and planetary systems. However, existing simulators do not scale effectively on more recent machines containing thousands of processors. In this paper, we introduce a new parallel simulator called ParallelGravity. This simulator is based on the CHARM++ infrastructure, which provides a powerful runtime system that automatically maps computation to physical processors. Using CHARM++ features, in particular its measurement-based load balancers, we were able to scale the gravitational force calculation of ParallelGravity on up to one thousand processors, with astronomical datasets containing millions of particles. As we pursue the completion of a production version of the code, our current experimental results show that ParallelGravity may become a powerful resource for the astronomy community.

1 Introduction

Cosmological simulators are currently an important component in the study of the formation of galaxies and planetary systems. Galaxies are the most distinctive objects in the universe, containing almost all the luminous material. They are remarkable dynamical systems, formed by non-linear collapse and a drawn-out series of mergers and encounters. Galaxy formation is indeed a challenging computational problem, requiring high resolutions and dynamic timescales. For example, to form a stable Milky Way-like galaxy, tens of millions of resolution elements must be simulated to the current epoch. Locally adaptive timesteps may reduce the CPU work by orders of magnitude, but not evenly throughout the computational volume, thus posing a considerable challenge for parallel load balancing. No existing N-body/Hydro solver can handle this regime efficiently.

The scientific payback from such studies can be enormous. There are a number of outstanding fundamental questions about the origins of planetary systems which these simulations would be able to answer.

Candidate to the Best Student Paper Award

To address these issues, various cosmological simulators have been created recently. PKDGRAV [1], developed at the University of Washington, can be considered among the state-of-the-art in that area. However, PKDGRAV does not scale efficiently on newer machines with thousands of processors. In this work, we present a new N-body cosmological simulator that utilizes the Barnes-Hut tree topology to compute gravitational forces. Our new simulator, named ParallelGravity, is based on the Charm++ runtime system [2]. We leverage the object based virtualization [3] inherent in the Charm++ runtime system to obtain automatic overlapping of communication and computation time, as well as to perform automatic runtime measurement-based load balancing. ParallelGravity advances the state-of-the-art in N-Body simulations by allowing the programmer to achieve higher levels of resource utilization with moderate programming effort. In addition, as confirmed by our experimental results, the use of Charm++ has enabled ParallelGravity to efficiently scale on large machine configurations.

The remainder of this paper is organized as follows. In Section 2 we present an overview of previous work in the development of parallel simulators for cosmology. Section 3 describes the major components of ParallelGravity. Section 4 presents a detailed view of the various optimizations that we have applied to ParallelGravity, with the resulting improvement in performance measured for each particular optimization. Finally, Section 5 contains our conclusions and the future directions of our work.

2 Related Work

There have been numerous studies on the N-Body problem, which involves the evolution of interacting particles that are under the effects of Newtonian gravitational forces. One of the most widely used methods to address this problem was proposed by Barnes and Hut [4]. In their scheme, the particles are associated to a hierarchical structure comprising a tree. This tree is traversed and the forces between particles are computed exactly or by approximations, depending on the distance between the given particles. This approach achieves reduction in the complexity of the problem from the original $O(N^2)$ to $O(N \log N)$, where N is the number of particles.

Given the power of hierarchical methods for N-Body simulations, such methods have been adopted for quite some time by the astronomy community [5]. One of the most popular codes currently in the astronomy area is PKDGRAV [1]. PKDGRAV is a parallel hierarchical tree-structured code used to conduct cosmological simulations on shared-memory and distributed-memory systems. It is portable across different communication substrates (e.g. MPI, PVM, etc.), and contains support for adaptive decomposition of work among the processors. In its current production version, PKDGRAV has been used in simulations of systems with millions of particles, and has been shown to scale well on up to hundreds of processors. One restriction in PKDGRAV's current version, however, arises from its limited load-balancing capability. This effectively prevents scaling the code efficiently on newer machines with thousands of processors.

Other cosmological simulators have been in use as well. Among these, two of the major codes are GADGET [6], developed in Germany, and falcON [7], developed at the University of Maryland. However, despite claiming a good scalability with the number of particles, falcON is a serial simulator. Meanwhile, GADGET originally had some of the same limitations of PKDGRAV when scaling to a large number of processors. This has been addressed in a more recent version of their code (GADGET-2), but there are not yet results reported with more than around one hundred processors [8].

3 New ParallelGravity Code

In order to leverage the features that the Charm++ runtime system offers, we decided to develop a new cosmological simulator called ParallelGravity. Our goal in developing this new application is to create a full production cosmological simulator that scales to thousands of processors.

This new simulator is capable of computing gravitational forces generated by the interaction of a very large number of particles, integrating those forces over time to calculate the movement of each particle. Since most of the running time of the application is devoted to force computation, our focus has been in optimizing this aspect of the code. The integration over time is typically easier to parallelize, and is not the focus of our analysis in this paper.

Since the gravitation field is a long range force, the total force applied to a given particle has contributions from all the other particles in the entire space. The algorithm we applied is based on a Barnes-Hut tree topology [4], which enables achieving an algorithmic performance of $O(N \log N)$. The tree generated by this algorithm is constructed globally over all the particles, and distributed across elements that are named *TreePieces*. This distribution is done according to the particles contained in each internal tree node. Figure 1 shows an example of such distribution. In this scheme, some of the internal nodes are replicated in more than one element. At the leaves of the tree are the particles, which are grouped by spatial proximity into *buckets* of a user-defined size. While walking the tree to compute forces, a single walk is performed for all the particles contained in a given bucket. The code allows the user to choose between different available tree distributions. Currently, two types of distributions are implemented: *SFC*, where a Space Filling Curve is used to impose a total ordering on the particles, with a contiguous portion of the curve being assigned to each TreePiece; and *Oct*, where particles are divided based on the nodes of an Octree covering the entire space.

3.1 Charm++ Infrastructure

Our new ParallelGravity code is based on the Charm++ [2] infrastructure. Charm++ is a parallel C++ library that implements the concept of *processor virtualization*: an application programmer decomposes her problem into a large number of components, or objects, and the interactions among those objects. The

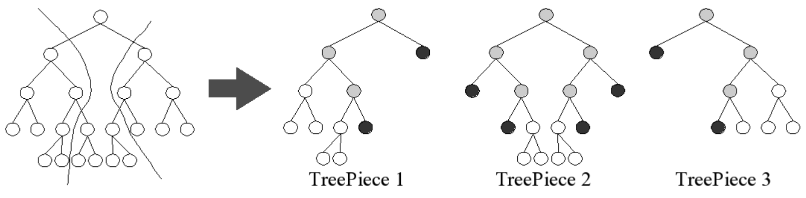


Fig. 1. Distribution of a tree across TreePieces (top levels). White nodes are owned by one TreePiece, black nodes are placeholders for remote nodes, gray nodes are shared among multiple TreePieces

objects, called *chares* in CHARM++ nomenclature, are automatically mapped to physical processors by the CHARM++ runtime system. Typically, the number of chares is much higher than the number of processors. By making the number of chares independent of the number of existing processors, CHARM++ enables execution of the same code on different machine configurations. This separation between logical and physical abstractions provides higher programmer productivity, and has allowed the creation of parallel applications that scale efficiently to thousands of processors, such as the molecular dynamics NAMD code [9].

The CHARM++ runtime system has the ability to migrate chares across processors during execution of an application. This migration capability is used by the powerful measurement-based load-balancing mechanism of CHARM++ [10]. The runtime system can measure various parameters in the chares, such as computational load or communication frequency and volume. CHARM++ provides a family of load balancers, targeting optimization of a variety of metrics. The user simply needs to select her desired balancers at application launch. During execution, the selected balancers will collect the measured chare values for the appropriate metrics, and dynamically remap chares across the available processors in a way that execution performance is optimized. This dynamic optimization capability is critical for applications such as particle system simulators, where particles can move in space and cause overloading on a given processor as the simulation progresses, while other processors become underutilized.

3.2 Major ParallelGravity Features

One of the early decisions in the design of ParallelGravity was to select where to compute the forces applied to a particle. Historically, there has been two main methods for that: (a) distributing the computation of the forces on that particle across all processors, with each processor computing the portion of the forces given by its subtrees, or (b) gathering at the processor owning that particle all the data needed to compute the forces on it. We decided to adopt the second scheme, since the capabilities of CHARM++ could be further exploited, as explained later in this section.

In our implementation of ParallelGravity, each TreePiece from Figure 1 is a CHARM++ chare. Thus, TreePieces are dynamically mapped to physical proces-

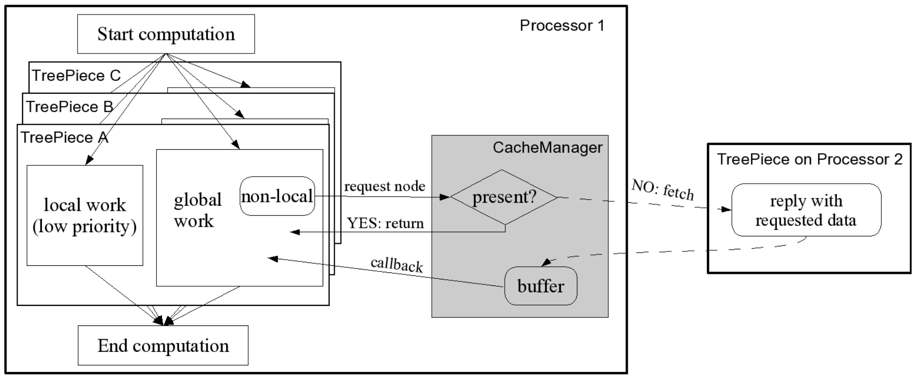


Fig. 2. Control flow of the execution of an iteration of force calculation

sors by the CHARM++ runtime system. The overall structure of how the code works is shown in Figure 2, and described in the next paragraphs.

To perform the computation of the forces on its particles, a TreePiece processes its buckets independently. For each bucket, the TreePiece must walk the overall tree and compute the forces due to all other particles. During the walk, visited nodes may be local (i.e. owned by this TreePiece) or non-local. For local nodes, the force computation can proceed immediately. For non-local nodes, a retrieval must be carried out, to bring the corresponding data into the TreePiece. A non-local node may reside either at another TreePiece of the same processor, or at a remote processor. In the first case, we use a direct data transfer between chares. In the second case, data must be requested to the remote processor. While waiting for remote data to arrive, the TreePiece can process other buckets.

Instead of repeating fetches of the same remote node for different bucket walks, we can use the property that buckets close in space will require similar remote portions of data. Therefore, we can buffer the imported data and have it used by all buckets in the TreePiece before discarding it. Because in CHARM++ we may have more than one chare in a single processor, we implemented this optimization at the processor level. This has been realized using a CHARM++ *group*, which we call *CacheManager*.

The purpose of the CacheManager is to serve all requests made by the TreePieces, and provide a caching mechanism to hide the latency of interprocessor data fetching. The CacheManager implements a random access to the cached data through the use of a hash table. To reduce the overhead of table lookup, the imported data is reconstructed into a local tree. Thus, once entering a subtree, TreePieces can iterate over direct pointers, until another cache miss occurs. Upon detecting a miss, the CacheManager will fetch the remote data and use callbacks to notify the requesting TreePiece when the data arrives. More advanced features provided by the CacheManager are presented in the next section, together with the observed experimental results.

Table 1. Characteristics of the parallel systems used in the experiments

System Name	Location	Number of Processors	Processors per Node	CPU Type	CPU Clock	Memory per Node	Type of Network
Tungsten	NCSA	2,560	2	Xeon	3.2 GHz	3 GB	Myrinet
BlueGene/L	EPCC	2,048	2	Power440	700 MHz	512 MB	Torus
HPCx	HPC-UK	1,536	16	Power5	1.5 GHz	32 GB	Federation

Because Charm++ executes chare methods in a non-preemptive fashion, a long sequence of consecutive tree walks might potentially prevent a processor from serving incoming data requests from other processors. In order to provide good responsiveness to incoming requests, we partitioned the processing of tree walks with a fine granularity. The grainsize is a runtime option, and corresponds to the number of buckets that will walk the tree without interruption. After that number of walks is performed, the TreePiece will yield the processor, enabling the handling of existing incoming data requests.

While dividing the computation into fine grains, we also distinguish between *local* and *global* computation. Local computation is defined as the interaction with the particles present in the same TreePiece. In contrast, global computation is defined as the interaction with the rest of the tree, i.e. the computation that involves non-local nodes. In particular, because this global computation is performed on the imported sections of the tree, it is on the more critical path. To express this different criticality, we utilized the prioritization mechanism embedded into Charm++. This mechanism allows establishing a total order of priority for the different operations performed by a TreePiece: the highest priority is assigned to accepting requests arriving from other processors, followed by sending replies to such requests, and finally the two types of computation (local and global), with the local one having the lowest priority. The Charm++ runtime system will schedule these operations according to such priorities.

4 Optimizations and Experimental Evaluation

After having a basic version of ParallelGravity in place, we studied its performance and added a number of optimizations to the code. Some of these optimizations were designed to exploit Charm++ aspects that enable high performance, whereas others were aimed at specific characteristics of particle codes. In this section, we describe the various optimizations that we have added, and present, in each case, the performance improvement that we obtained by applying such techniques to real cosmological datasets. Although the following subsections describe the effect of each optimization technique separately, our integrated version of ParallelGravity contains all the optimizations. It is this integrated, fully optimized version that we use in the last subsection, to show how the current code scales with increasing system size. In our experiments, we used the parallel systems described in Table 1, and the following particle datasets:

- **lambs**: Final state of a simulation of a $71Mpc^3$ volume of the Universe with 30% dark matter and 70% dark energy. 144^3 particles, i.e. nearly three million particles, are used (3M). Three subsets of this dataset are obtained by taking random subsamples of size thirty thousand (30K), three hundred thousand (300K), and one million particles (1M), respectively.
- **dwarf**: A snapshot at $z = .3$ of a multi-resolution simulation of a dwarf galaxy forming in a $28.5Mpc^3$ volume of the Universe with 30% dark matter and 70% dark energy. The effective resolution in the central regions is equivalent to 2048^3 particles in the entire volume. The total dataset size is nearly five million particles.

4.1 Uniprocessor Performance

While developing a parallel application like ParallelGravity, we are concerned not only with scalability, but also with performance (i.e. execution time). Hence, it is important to evaluate the single processor performance as well. To do this, we compared the serial performances of ParallelGravity and PKDGRAV, on different subsets of the *lambs* dataset.

Table 2 shows the execution times for the gravitational force calculation phase of the two simulators, running on one Xeon processor of Tungsten. As the table shows, ParallelGravity's serial performance is comparable to the performance of PKDGRAV, even for the larger datasets. The slightly greater times for ParallelGravity (increase of less than 6%) are due to the overheads caused by optimizations aimed at improving parallel performance. As the next subsections will demonstrate, this overhead is a very small price to pay in view of the large gains that we can achieve with those optimizations in the parallel case.

4.2 Software Cache Mechanism

As mentioned in Section 3.2, the CacheManager not only reduces the number of messages exchanged to fetch remote data, but also hides the latency of fetching data from other processors. We evaluated the effectiveness of the CacheManager on the 1 million subset of the lambs dataset running on varying numbers of HPCx processors. Table 3 shows that the CacheManager dramatically reduces the number of messages exchanged. The performance improvement due to sending a much lower number of messages, combined with the latency-hiding effects of the CacheManager, produces a sharp reduction in the execution time, as seen in Table 3. Thus, the software cache mechanism is absolutely necessary to obtain good parallel performance.

Table 2. Time, in seconds, for one step of force calculation in serial execution

Simulator	Number of Particles			
	30,000	300,000	1 million	3 million
PKDGRAV	0.83	12.0	48.5	170.0
ParallelGravity	0.83	13.2	53.6	180.6

Table 3. CacheManager effects in terms of number of messages and iteration time

		Number of Processors				
		4	8	16	32	64
Number of Messages	No Cache	48,723	59,115	59,116	68,937	78,086
($\times 10^3$)	With Cache	72	115	169	265	397
Time	No Cache	730.7	453.9	289.1	67.4	42.1
(seconds)	With Cache	39.0	20.4	11.3	6.0	3.3

4.3 Data Prefetching

As in PKDGRAV, we can take the principle of the software cache one step further by fetching not only the node requested by a TreePiece, but proactively also part of the subtree rooted at that node. The user can specify the *cache depth* (analogous to the concept of cache line in hardware) as the number of levels in the tree to recursively prefetch. The rationale for this is that if a node is visited, most probably its children will be visited as well. This mechanism of prefetching more data than initially requested helps to reduce the total number of messages exchanged during the computation. Since every message has both a fixed and a variable cost, prefetching reduces the total fixed cost of communication. On the other hand, a cache depth of more than zero might cause some data to be transferred but never used, thus increasing the variable part of the cost.

If a TreePiece requested data to the CacheManager only when required by the tree-walk computation, the CacheManager might not have it. This would trigger a fetch of the data from the remote node, but at the same time it would suspend the computation for the requesting bucket until the moment of data arrival. Both the interruption of the tree walk and the notification from the CacheManager incur an overhead. To limit this effect, we developed a *prefetching phase* which precedes the real tree-walk computation. During this phase, we traverse the tree and prefetch all the data that will be later used during the computation in the regular tree walk. This prefetching phase can work with different cache depths.

We used the lambs dataset on 64 processors of Tungsten to evaluate the impact of cache depth and the prefetching phase. Figure 3(a) shows the execution time for different cache depths with and without the prefetching phase. In both cases there is an optimal value of cache depth, at which the execution time is minimal. The optimal point is achieved when the fixed cost associated with every message and the variable cost of transferring data over the network are in balance. According to our results in Figure 3(a), the optimal cache depth seems to vary between 3 and 5.

We can also see that the prefetching phase improves performance for all considered values of cache depth. This is due to the increased hit rate of the cache. While executions without the prefetching phase generate a cache hit rate of about 90%, with the prefetching active the hit rate rises to 95-97% for SFC tree decomposition, and 100% for Oct decomposition. The greater accuracy in prefetching for Oct decomposition is due to the better prefetching algorithm we developed, given the constraint that prefetching must be lightweight. Although

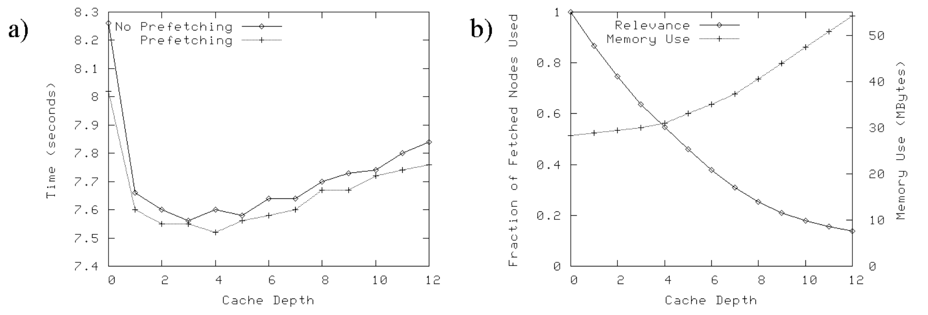


Fig. 3. Impact of cache depth and prefetching on (a) iteration time, and (b) relevance and memory use

Oct decomposition provides a clear benefit in terms of cache hit rate over SFC, the full effects on the entire execution time are more complex and will require more detailed studies to be fully characterized.

We define the *relevance* as the ratio between the number of nodes fetched and used, and the total number of nodes fetched. Ratios closer to 1.0 represent a better relevance. In Figure 3(b), we plot the relevance on the left vertical axis. The observed relevance decreases with increasing cache depth, leading to unnecessarily higher memory consumption, as plotted on the right vertical axis of the same graph. Nevertheless, this higher memory consumption due to caching is limited to a fraction of the total memory footprint for moderate values of cache depth. At a very low value of relevance, the cost of fetching a large amount of extra data is not offset by the benefit of having the data already present in the software cache when it is requested. This is why the execution time rises for large values of cache depth in Figure 3(a). The prefetching phase does not affect the relevance, since it does not change which data items are transferred. Prefetching simply causes those data transfers to occur earlier.

Thus, we see that using the prefetching phase along with a small but non-zero value of cache depth improves performance. In the following subsections, we will assume that the prefetching phase is active, and a reasonable value of cache depth is used.

4.4 Tree-in-Cache Effects

In Section 3 we introduced the concept of local and global computation.We pointed that the global work is on the critical path, and that the local work can be used to hide the latency of data transfers. From this, it is clear that we should have as much local work as possible. One point to notice is that in the Charm++ environment we fragment the particle dataset in more TreePieces than the number of physical processors available. This over-decomposition reduces the amount of local work per TreePiece. In some of our experiments, when

Table 4. Distribution of work between local and global computation

	Local	Global
Original Code	16%	84%
Code with Tree-in-cache	42%	58%

increasing the number of processors beyond one hundred, the local work became insufficient to maintain the processor busy during the entire computation.

By noticing that during the force computation there is no migration of TreePieces, we can consider collectively all the TreePieces residing on a given processor. We can attribute to local computation not only the work related to nodes/particles present in the same TreePiece, but also the work related to particles and nodes present in other TreePieces in the same processor. This is implemented by having each TreePiece registering to the CacheManager at the beginning of the computation step. The CacheManager will then create a superset tree including all the trees belonging to the registered TreePieces. Each TreePiece will now consider as local work this entire tree. During this operation, only the nodes closest to the root of the tree will be duplicated. According to our tests with datasets of a few million particles, less than one hundred nodes were duplicated.

Table 4 summarizes the percentage of local and global work for a simulation on 64 Tungsten processors with the lambs-300K subset. The percentages changed considerably before and after this optimization. In our tests, this new scheme enabled scaling the computation up to hundreds of processors. However, when reaching the limit of one thousand processors, even the extra work from co-resident TreePieces becomes insufficient. A solution that we are investigating is to split the global walk into multiple sub-walks.

4.5 Interaction Lists

After having preceded the computation with a prefetching phase, and verifying that it is accurate, we explored a faster algorithm for gravitational force computation. This algorithm is centered on the concept of *interaction lists*, which we describe in this subsection. The new algorithm is based on the same principle of the CacheManager: two buckets close in space will tend to interact similarly with a given remote node.

In the regular ParallelGravity algorithm, whenever a bucket walk visits a tree node, a fundamental test is carried out. In this test, we check the spatial position of the bucket in respect to the particles in that node. If the bucket is sufficiently *far* from the node, the forces on the bucket due to the entire subtree rooted at that node are immediately computed, using the subtree's center of mass. Otherwise, ParallelGravity *opens* the node, i.e. it recursively traverses the subtree rooted at that node. Thus, the threshold used to decide if a node is close enough to the bucket represents the *opening criteria* for deciding whether the visited node must be opened or not.

Table 5. Number of checks for opening criteria, in millions

	lambs 1M	dwarf 5M
Original algorithm	120	1,108
Modified algorithm	66	440

Instead of checking the opening criteria at a given node for each bucket independently, we can modify the algorithm and do that check for various local buckets at once. We can do this collective check using the buckets' ancestors in the local tree. These ancestors will be local nodes containing particles which are close in space. If an ancestor needs to open a visited node, that node will be opened for every bucket that is a descendent of such ancestor. On the other hand, if a node is far enough for that ancestor, this node will be far enough for all the ancestor's buckets too. In this second case, we can directly compute the interaction between the node and all these local buckets.

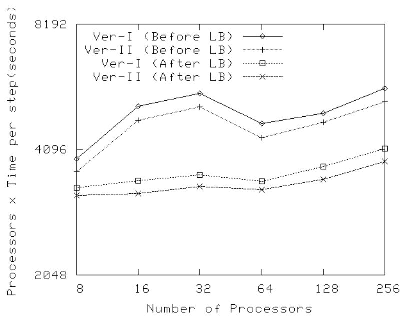
By grouping the checking for various local buckets, we can reduce the total number of checks for opening nodes. As an example, Table 5 shows the number of checks that are observed with the two algorithms, executing on the HPCx system with our two datasets. A potential problem in this modified algorithm is that it may cause less effective usage of the hardware cache: because the computation of interactions proceeds for various local buckets, one bucket's data may flush another bucket's data from the hardware cache. We can reduce the number of hardware-cache misses by storing all the nodes that interact with a given bucket in a bucket's *interaction list*, and perform the entire computation of forces on that bucket at the end of the tree walk. Performance is improved even further with interaction lists because compilers may keep a particle's data inside CPU registers while computing interactions with the nodes in the list.

Figure 4(a) plots the execution time for both the regular algorithm and the new algorithm employing interaction lists, showing also cases where load balancing was employed (load balancing is the subject of our analysis in the next subsection). The new algorithm shows a performance improvement over the entire range considered. This improvement varies between 7% and 10%. We used ParallelGravity with interaction lists for the uniprocessor tests of Section 4.1.

4.6 Load Balancer Importance

After describing all the optimizations applied to the basic ParallelGravity code, we assess the importance of the Charm++ automatic load balancing framework in improving the performance of our simulations. In particular, we emphasize the fact that the code instrumentation and the migration of chares in the system are totally automated, and do not require any programmer intervention.

Figure 4(a) shows the effect of load balancing on both versions of ParallelGravity, one with the regular algorithm and the other with the interaction-list implementation. The improvement from load balancing is similar in both algorithms. We see that, before load balancing, the behavior of the algorithms

a)


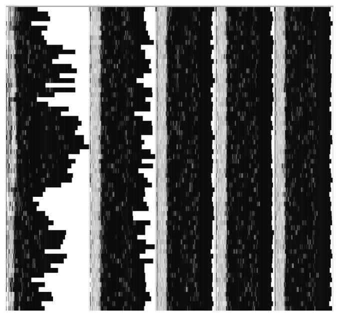
b)


Fig. 4. (a) Comparison between regular ParallelGravity (Ver-I) and the one with interaction lists (Ver-II) before and after load balancing on BlueGene/L for dwarf dataset. (b) Effect of Load Balancer for dwarf dataset on 64 BlueGene/L processors

is somewhat random and determined only by the particle decomposition. This happens because different particles in space require different amounts of computation. TreePieces owning heavy particles will be overloaded, hence cause bad performance. After load balancing, performance improves between 15% and 35%.

To further analyze the improvements from the load balancer, Figure 4(b) displays a view from our Projections performance analysis tool, a component of Charm++. This view corresponds to five timesteps of a simulation on 64 BlueGene/L processors. The horizontal axis represents time, while each horizontal bar represents a processor. Darker colors represent higher utilization, with black as full utilization and white as idleness. One can see that even starting from a very unbalanced situation on the first timestep, after two timesteps the load balancer improves performance quite significantly, approaching almost perfect balance. The gray region at the beginning of each timestep, where utilization is lower, corresponds to the communication overhead due to prefetching. The time spent by the application in load balancing and in domain decomposition is hardly visible in the figure. It corresponds to the period between the end of the longest black bar in one timestep and the beginning of the gray region of the next timestep. That time is negligible.

It is relevant to notice that the input dataset (dwarf) is highly clustered at the center of the simulation space, and its spatial distribution of particles is very uneven. This non-uniform particle distribution is reflected by the varying processor utilization in the first timestep of the simulation. Situations like this present the biggest challenge to obtain load balance across processors. Nevertheless, the Charm++ load balancers achieved very good balance.

4.7 Scalability with Number of Processors

By applying all the optimizations described in the previous subsections, and making use of various Charm++ features, we obtain our best performing version

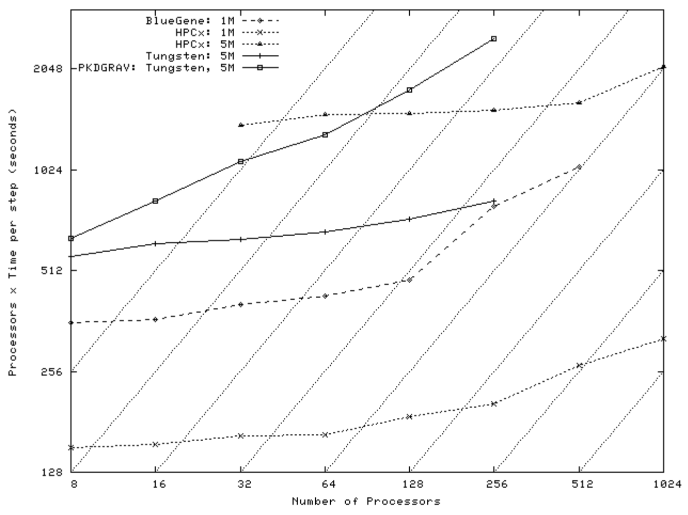


Fig. 5. ParallelGravity scaling on various systems and comparison with PKDGRAV

of ParallelGravity. We used this version to conduct scaling tests on large machine configurations, and to make scaling comparisons with PKDGRAV.

Figure 5 shows the scaling of ParallelGravity on BlueGene/L, HPCx and Tungsten. The vertical axis is the product of the time per iteration and the number of processors in the simulation. Horizontal lines represent perfect scalability, while the diagonal lines represent no gain in scaling.

For the lambs1M dataset, the algorithm scales well up to 128 processors on BlueGene/L and 256 processors on HPCx. Beyond these points, there is not adequate work available for each processor, and the gain is reduced. BlueGene/L has the most problems, and there is almost no advantage from the increased number of processors. The dwarf dataset, being larger with 5 million particles, allows good scaling up to 1024 processors of HPCx.

Figure 5 also presents the scaling comparison between ParallelGravity and PKDGRAV on Tungsten. We can see that ParallelGravity scales much better than PKDGRAV, maintaining a good performance over the entire range considered. Due to machine unavailability, we have not been able to run tests with more than 256 Tungsten processors.

5 Conclusions and Future Work

In this paper, we have presented a new cosmological simulator named ParallelGravity. Our design was guided by the goal of achieving good scalability on modern parallel machines, with thousands of processors. Our experimental results show that ParallelGravity's serial performance is comparable to that of top-level simulators existing today. Meanwhile, by employing various optimizations enabled by the Charm++ runtime system, the gravity calculation phase in ParallelGravity was shown to scale very well up to one thousand processors with real astronomical datasets. This level of scalability places ParallelGravity as a potentially powerful resource for the astronomy community.

Despite ParallelGravity's good observed scalability, we intend to study other load balancing schemes and parallelization techniques that may provide even further benefits. Moreover, ParallelGravity still needs to undergo a few more steps to become a production-level simulator. We are working on adding support for more physics, such as fluid-dynamics and periodic boundaries, as well as providing multiple timestepping. In addition, as we start our tests on thousands of processors, we are also analyzing the performance of other phases of the simulation, such as the construction of the particle tree. Given the support for various types of trees already present in the code, we intend to conduct a detailed study of their effects on the simulation.

Acknowledgments
This work was supported in part by the National Science Foundation, under grant number NSF ITR 0205611. We are thankful for the access to parallel systems at Edinburgh's HPCx Consortium, at Edinburgh's EPCC Center, and at Illinois' NCSA.

References

1. M. D. Dikaiakos and J. Stadel, "A performance study of cosmological simulations on message-passing and shared-memory multiprocessors," in *Proceedings of the International Conference on Supercomputing - ICS'96*, (Philadelphia, PA), pp. 94–101, December 1996.
2. L. V. Kale and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++* (G. V. Wilson and P. Lu, eds.), pp. 175–213, MIT Press, 1996.
3. L. V. Kalé, "Performance and productivity in parallel programming via processor virtualization," in *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, (Madrid, Spain), February 2004.
4. J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, December 1986.
5. G. Lake, N. Katz, and T. Quinn, "Cosmological N-body simulation," in *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, (Philadelphia, PA), pp. 307–312, February 1995.
6. V. Springel, N. Yoshida, and S. White, "GADGET: A code for collisionless and gasdynamical simulations," *New Astronomy*, vol. 6, pp. 79–117, 2001.
7. W. Dehnen, "A hierarchical $O(N)$ force calculation algorithm," *Journal of Computational Physics*, vol. 179, pp. 27–42, 2002.
8. V. Springel, "The cosmological simulation code GADGET-2," *MNRAS*, vol. 364, pp. 1105–1134, 2005.
9. J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé, "NAMD: Biomolecular simulation on thousands of processors," in *Proceedings of SC 2002*, (Baltimore, MD), September 2002.
10. G. Zheng, *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.