A Parallel Monte Carlo Code for Simulating Collisional N-body Systems

Bharath Pattabiraman[1,2]

bharath@u.northwestern.edu

Stefan Umbreit[1,3]

Wei-keng Liao[1,2]

Alok Choudhary[1,2]

Vassiliki Kalogera[1,3]

Gokhan Memik[1,2]

and

Frederic A. Rasio[1,3]

Received ______________________; accepted ______________________

[1]Center for Interdisciplinary Exploration and Research in Astrophysics, Northwestern University, Evanston, USA.

[2]Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, USA.

[3]Department of Physics and Astronomy, Northwestern University, Evanston, USA.

Abstract

We present a new parallel code for computing the dynamical evolution of collisional N-body systems with up to $N \sim 10^7$ particles. Our code is based on the the Hnon Monte Carlo method for solving the Fokker-Planck equation, and makes assumptions of spherical symmetry and dynamical equilibrium. The principal algorithmic developments involve optimizing data structures, and the introduction of a parallel random number generation scheme, as well as a parallel sorting algorithm, required to find nearest neighbors for interactions and to compute the gravitational potential. The new algorithms we introduce along with our choice of decomposition scheme minimize communication costs and ensure optimal distribution of data and workload among the processing units. The implementation uses the Message Passing Interface (MPI) library for communication, which makes it portable to many different supercomputing architectures. We validate the code by calculating the evolution of clusters with initial Plummer distribution functions up to core collapse with the number of stars, N, spanning three orders of magnitude, from 10^5 to 10^7. We find that our results are in good agreement with self-similar core-collapse solutions, and the core collapse times generally agree with expectations from the literature. Also, we observe good total energy conservation, within less than 1% throughout all simulations. We analyze the performance of the code, and demonstrate near-linear scaling of the runtime with the number of processors up to 64 processors for $N = 10^5$, 128 for $N = 10^6$ and 256 for $N = 10^7$. The runtime reaches a saturation with the addition of more processors beyond these limits which is a characteristic of the parallel sorting algorithm. The resulting maximum speedups we achieve are approximately $60\times$, $100\times$, and $220\times$, respectively.

Subject headings: Methods: numerical, Galaxies: clusters: general, globular clusters: general

1. Introduction

The dynamical evolution of dense star clusters is a problem of fundamental importance in theoretical astrophysics. Important examples of star clusters include globular clusters, spherical systems containing typically 10^5 - 10^7 stars within radii of just a few parsec, and galactic nuclei, even denser systems with up to 10^9 stars contained in similarly small volumes, and often surrounding a supermassive black hole at the center. Studying their evolution is critical to many key unsolved problems in astrophysics. It connects directly to our understanding of star formation, as massive clusters are thought to be associated with major star formation episodes, tracing the star-formation histories of their host galaxies. Furthermore, globular clusters can trace the evolution of galaxies over a significant cosmological time span, as they are the brightest structures with which one can trace the halo potential out to the largest radii, and they are very old, potentially even predating the formation of their own host galaxies. Unlike stars and planetary nebulae, globular clusters are not simply passive tracers of galaxy kinematics as their internal dynamics are affected by the galactic tidal field. Therefore, their internal properties and correlations with their host galaxies are, thus, likely to contain information of the merger history of galaxies and haloes.

Dynamical interactions in dense star clusters play a key role in the formation of many of the most interesting and exotic astronomical sources, such as bright X-ray and gamma-ray sources, radio pulsars, and supernovae. The extreme local stellar densities, which can reach of the order of $10^6\,\mathrm{pc}^{-3}$, give rise to complex dynamical processes: stellar encounters, tidal captures, physical collisions, mergers, and high-speed ejections (Heggie & Hut 2003). The primary challenge in modeling dense clusters lies in the tight coupling of these processes and their scales as they influence and inform one another both locally, e.g., through close encounters or collisions on scales of $1 - 100\,\mathrm{R}_\odot$, or $10^{-8} - 10^{-6}$ pc, and globally on the scale of the whole system through long-range, gravitational interactions. Close binary interactions can occur frequently every $10^6 - 10^9$ yr depending on the cluster density, relative to the global cluster evolution timescale. Furthermore, in the time between close encounters, stellar particles, single and binary, change their physical properties due to their internal chemical and nuclear evolution and due to mass and angular momentum transfer or losses. All these changes affect the rates of close encounters and couple to the overall evolution of the cluster.

Due to these enormous ranges in spatial and temporal scales that have to be modeled, simulating dense star clusters with a million stars or more is a formidable computational challenge. A thorough analysis of the scaling of the computational cost of the direct N-body methods is presented in Hut et al. (1988). Although direct N-body methods are free of any approximations in the stellar dynamics , their steep $\propto N^2$ scaling has limited simulations to an initial $N \sim 10^5$ stars (Zonoozi et al. 2011; Jalali et al. 2012). However, the number of stars in real systems like globular clusters and galactic nuclei can be several orders of magnitude larger.

On the contrary, the Monte Carlo method calculates the dynamical evolution of the cluster in the Fokker-Planck approximation, which applies when the evolution of the cluster is dominated by two-body relaxation, and the relaxation time is much larger than the dynamical, or orbital, time. In the end, this allows for a scaling closer to $N \log N$, but assumptions of spherical symmetry and dynamical equilibrium have to be made. The Hnon Monte Carlo (MC) technique (Hénon 1971) which is based on orbit averaging, represents a balanced compromise between realism and speed. The MC method allows for a star-by-star realization of the cluster, with its N mass shells representing the N stars in the cluster. Integration is done on a relaxation timescale, and the total computational cost scales as $N \log N$ (Hénon 1971). Our work here is based on the Hnon-type MC cluster evolution code, CMC ("Cluster Monte Carlo"), developed over many years by Joshi et al. (2000, 2001); Fregeau et al. (2003); Fregeau & Rasio (2007); Chatterjee et al. (2010); Umbreit et al. (2012). CMC now includes a detailed treatment of strong binary star interactions and physical stellar collisions (Fregeau & Rasio 2007), as well as an implementation of single and binary star evolution (Chatterjee et al. 2010), and the capability of handling the dynamics around a central massive black hole (Umbreit et al. 2012).

A typical simulation of about a million stars up to average cluster ages of 12 Gyr using the CMC code can be run on a modern desktop computer in a reasonable amount of time (days to weeks). However, given the scaling of the computational cost, simulations of clusters of 10^7 stars will take a prohibitive amount of time which makes simulating nuclear star clusters not feasible. Scaling up to even larger number of stars becomes possible only through parallelization of our code.

In this paper, we present in detail the latest version of CMC, which is capable of simulating collisional systems of up to $N \sim 10^7$. In Section 2, we take a look at the components of the code and summarize both its numerical and computational aspects. In Section 3, we describe the flow of the parallel code, elucidating how we designed each part to achieve optimal performance on distributed parallel architectures. In addition, we describe in the Appendix an optional CMC feature that accelerates parts of the code using a general purpose Graphics Processing Unit (GPU). We show a comparison of results and analyze the performance of the code in Section 4. Conclusions and lines of future work are discussed in Section 5.

2. Code Overview

2.1. Numerical Methods

Starting with an initial spherical system of N stars in dynamical equilibrium, we begin by assigning to each star a mass, position and velocity (radial and transverse components) by sampling from a distribution function $f(E, J)$, where E and J are the orbital energy and angular momentum (e.g., Binney & Tremaine 2008). We assign positions to the stars in a monotonically increasing fashion, so the stars are sorted by their radial position initially. The system is assumed to be spherically symmetric and hence we ignore the direction of the position vector and transverse velocity. Following initialization, the algorithm goes through the following sequence of steps iteratively over a specified number of time steps. Figure 1 shows the flowchart of our basic algorithm.

1. *Potential calculation.* The stars having been sorted by increasing radial positions in the cluster, the potential at radius r, which lies between two stars at positions r_k and r_{k+1}, is given by
$$\Phi(r) = G\left(-\frac{1}{r}\sum_{i=1}^{k} m_i - \sum_{i=k+1}^{N}\frac{m_i}{r_i}\right) \ . \tag{1}$$
where m_i is the mass, and r_i the position of star i. It is sufficient to compute and store the potential $\Phi_k = \Phi(r_k)$ at radial distances r_k $(k = 1, ..., N)$ i.e., at the positions of all stars. This can be done recursively as follows::
$$\Phi_{N+1} = 0 \ ,$$
$$M_N = \sum_{i=1}^{N} m_i \ ,$$
$$\Phi_k = \Phi_{k+1} - GM_k\left(\frac{1}{r_k} - \frac{1}{r_{k+1}}\right) \ , \tag{2}$$
$$M_{k-1} = M_k - m_k \ .$$

To get the potential $\Phi(r)$ at any other radius, one first finds k such that $r_k \leq r \leq r_{k+1}$ and then computes:

$$\Phi(r) = \Phi_k + \frac{1/r_k - 1/r}{1/r_k - 1/r_{k+1}}(\Phi_{k+1} - \Phi_k) \ . \tag{3}$$

2. *Time step calculation.* Different physical processes are resolved on different time scales. We use a shared time step scheme where time steps for all the physical processes to be simulated are calculated and their minimum is chosen as the global time step for the current iteration. The time steps are calculated using the following expressions (see Fregeau & Rasio 2007; Goswami et al. 2011, for more details):

$$T_{\rm rel} = \frac{\theta_{max}}{\pi/2}\frac{\pi}{32}\frac{\langle v_{rel}\rangle^3}{\ln(\gamma N)G^2 n \left\langle (M_1+M_2)^2\right\rangle}, \tag{4}$$

$$T_{\rm coll}^{-1} = 16\sqrt{\pi} n_s \left\langle R^2\right\rangle \sigma \left(1 + \frac{G\left\langle MR\right\rangle}{2\sigma^2\left\langle R^2\right\rangle}\right), \tag{5}$$

$$T_{\rm bb}^{-1} = 16\sqrt{\pi} n_b X_{bb}^2 \left\langle a^2\right\rangle \sigma \left(1 + \frac{G\left\langle Ma\right\rangle}{2\sigma^2 X_{bb}\left\langle a^2\right\rangle}\right), \tag{6}$$

$$T_{\rm bs}^{-1} = 4\sqrt{\pi} n_s X_{bs}^2 \left\langle a^2\right\rangle \sigma \left(1 + \frac{G\left\langle M\right\rangle\left\langle a\right\rangle}{\sigma^2 X_{bs}\left\langle a^2\right\rangle}\right). \tag{7}$$

where $T_{\rm rel}, T_{\rm coll}, T_{\rm bb}$, and $T_{\rm bs}$ are the relaxation, collision, binary-binary and binary-single time steps respectively. Here θ_{max} is the maximum angle of deflection of the two stars undergoing a representative two-body encounter ; v_{rel} their relative velocities, and n the local number density of stars; n_s and n_b are the number densities of single and binary stars, respectively; σ is the one-dimensional velocity dispersion, and a is the semi-major axis. X_{bb} and X_{bs} are parameters that determine the minimum closeness of an interaction to be considered a strong interaction..

The value of T_{rel} is calculated for each star after which the minimum is taken as the value of the global relaxation time step. We use sliding averages over the neighboring 10 stars on each side to calculate the mean quantities shown in $< \ldots >$ in Equations

4 to 7. The other three time-steps, $T_{\rm coll}, T_{\rm bb}$ and $T_{\rm bs}$ are averaged over the central 300 stars as in the procedure followed in Goswami et al. (2011). Our choice of the number of stars averages are calculated over gives a good compromise between accuracy and computational speed.

3. *Relaxation and strong interactions.* Depending on the physical system type, one among the following three are performed on each pair of stars (i) Two-body relaxation evaluates an analytic expression for a representative encounter between two nearest-neighboring stars. (ii) Binary interactions (binary-binary and binary-single) are simulated using Fewbody, an efficient computational toolkit for evolving small-N dynamical systems (Fregeau et al. 2004). Fewbody does a direct integration of Newton's equations for 3 or 4 bodies using the 8th order Runge-Kutta Prince-Dormand method. (iii) Stellar collisions are treated in the simple "sticky sphere" approximation, where two bodies are merged based on the probability that their radii touch and their properties are changed correspondingly.

4. *Stellar Evolution.* We use the SSE (Hurley et al. 2000) and BSE (Hurley et al. 2002) stellar evolution routines, which are based on analytic functional fits to theoretically calculated stellar evolution tracks, to simulate the evolution of single and binary stars.

5. *New orbits computation.* Due to the changes in the orbital properties of the stars following the interactions they undergo, new positions and velocities are assigned in orbits that are consistent with their new energies and angular momentums. Then, a new position is randomly sampled according to the amount of time the star spends at a given point along the orbit.

   We start by finding the pericenter and apocenter distances of a star's new orbit. Given a star with energy E and angular momentum J moving in the gravitational potential $\Phi(r)$, its rosette orbit $r(t)$ oscillates between two extreme values $r_{\rm min}$ and

$r_{\rm max}$, which are roots of:

$$Q(r) = 2E - 2\Phi(r) - J^2/r^2 = 0 \ . \tag{8}$$

Since we store the potential values only at the positions of the stars, this equation cannot be analytically solved before determining the interval in which the root lies. In other words, we need to determine k such that $Q(r_k) < 0 < Q(r_{k+1})$. We use the bisection method, which, starting with two values of k, $k_{\rm left}$ and $k_{\rm right}$, repeatedly divides the interval into two parts, retaining only the one in which the solution is contained while discarding the other until the interval converges on to the root. Once the interval is found, Φ, and thus Q, can be computed analytically in that interval, and so can $r_{\rm min}$ and $r_{\rm max}$.

The next step is to select a position of the star in the new orbit between $r_{\rm max}$ and $r_{\rm min}$. The probability to choose a position in an interval dr should be equal to the fraction of time spent by the star in dr, i.e.:

$$\frac{dt}{T} = \frac{dr/\left|v_{\rm r}\right|}{\int_{r_{\rm min}}^{r_{\rm max}} dr/\left|v_{\rm r}\right|}$$

with the radial velocity $v_r = [Q(r)]^{1/2}$. We use the von Neumann rejection sampling technique to sample a position according to this probability. We take a number F which is everywhere larger than the probability distribution $f(r)$. Then we draw two random numbers X and X' and compute

$$r_0 = r_{\rm min} + (r_{\rm max} - r_{\rm min})X$$

$$f_0 = FX'$$

If the point (f_0, r_0) lies below the curve, i.e., if $f_0 < f(r_0)$, we take $r = r_0$ as the new position; else we reject it and draw a new point in the rectangle with a new pair of random numbers. We repeat this until a point below the curve is obtained. In our code, a slightly modified version of the method is used, since $f(r) = 1/|v_r|$ becomes infinite at both ends of the interval. A detailed description can be found in Joshi et al. (2000).

6. *Sort stars by radial distance.* This part uses the Quicksort algorithm (Hoare 1961) to sort the stars based on their new positions. Sorting the stars is essential to determine the nearest neighbors for relaxation and strong interactions, and also for computing the gravitational potential.

7. *Diagnostics and program termination.* These involve other minor calculations which include the computation of diagnostic values and control the program termination. Examples are, half-mass radius, core radius, number of core stars among others. This does not appear in the flowchart shown in Figure 1 since it represents minor book-keeping calculations that are done at various places in the code.

2.2. Time Complexity Analysis

In addition to the flow of the code, Figure 1 also shows the time complexity for each of the above steps. The calculation of $T_{\rm rel}$ involves averaging over a fixed number of neighboring stars and hence has constant time complexity, $\mathcal{O}(1)$. As these averages are performed on each star to estimate their individual time steps from which the the minimum is chosen, the time step calculation scales as $\mathcal{O}(N)$. The effect of relaxation and strong interactions is calculated between pairs of stars that are radially nearest neighbors. Since these calculations involve constant time operations for each star, the time complexity of the perturbation step is $\mathcal{O}(N)$. Stellar evolution operates on a star-by-star basis performing operations of constant time for a given mass spectrum, and hence also scales as $\mathcal{O}(N)$. Determination of new orbits for each star involves finding the roots of an expression on an

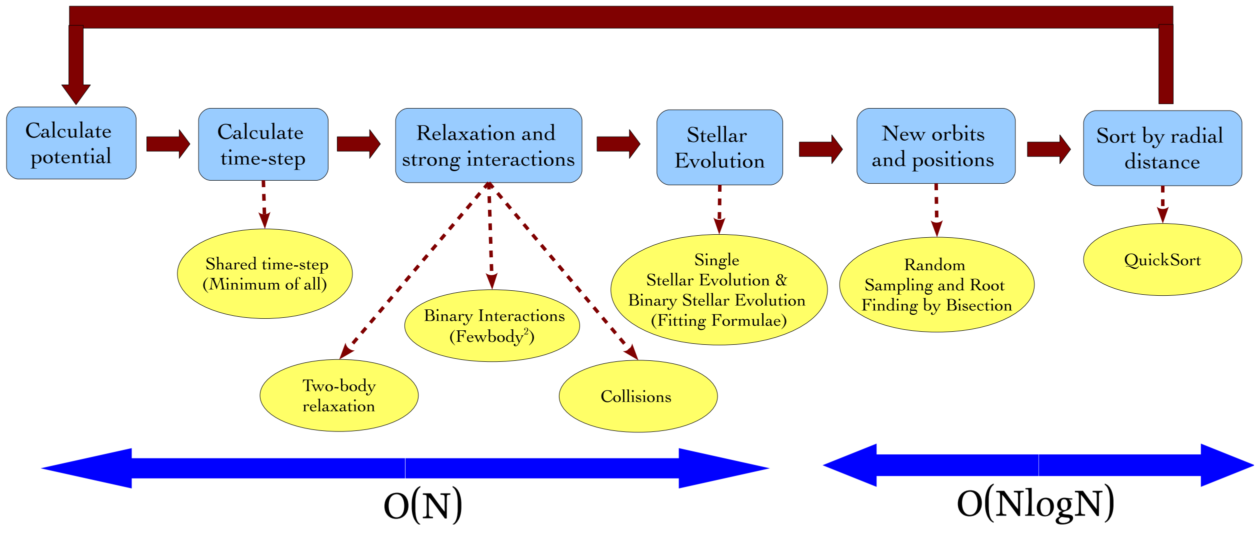
Calculate potential
Calculate time-step
Relaxation and strong interactions
Stellar Evolution
New orbits and positions
Sort by radial distance
Shared time-step (Minimum of all)
Binary Interactions (Fewbody^2)
Two-body relaxation
Collisions
Single Stellar Evolution & Binary Stellar Evolution (Fitting Formulae)
Random Sampling and Root Finding by Bisection
QuickSort
O(N)
O(NlogN)

Fig. 1.— A flowchart of the CMC (Cluster Monte Carlo) code with the following steps. (1) Potential Calculation-calculates the spherically symmetric potential. (2) Time-step Calculation-computes a shared time-step used by all processes. (3) Relaxation and Strong interactions-depending on the physical system type, performs two-body relaxation, strong interaction, or physical collision on every pair of stars. (4) Stellar Evolution-evolves each star and binary using fitting formulae (5) New Positions and Orbits-samples new positions and orbits for stars. (6) Sorting-sorts stars by radial position.

unstructured one-dimensional grid using the bisection method. The bisection method scales as $\mathcal{O}(\log N)$and as this is done for each star, this part has a time complexity of $\mathcal{O}(N \log N)$. The radial sorting of stars is done using Quicksort which has the same time complexity (Hoare 1961).

3. Parallelization

Achieving optimal performance of parallel machines requires algorithms to be carefully designed, and hence, parallel algorithms are often very different than their serial counterparts and require a considerable effort to develop. The key to a successful algorithm is (1) good load balance, i.e., the efficient utilization of the available processing units, and (2) minimal communication between these units. The communication cost depends directly on the choice of domain decomposition, i.e, the way in which work and/or data is partitioned into smaller units for processing in parallel. For example, a good domain decomposition for ideal load balance would be distributing the data of the stars evenly among the processors assuming the computational cost for processing each star is similar. This will ensure the workload is evenly balanced across processors given that they all perform the same number of operations, as in a Single Program, Multiple Data (SPMD) programming model. However, how such a decomposition would influence the need for communication between processing units is very specific to the algorithm. In essence, a thorough knowledge of the algorithm, and its data access patterns is necessary for designing any efficient parallel application.

3.1. Data Dependencies and Parallel Processing Considerations

While deciding upon the domain decomposition, we have to take into account any dependencies, i.e., the way the data is accessed by various parts of the application, as they may directly influence both the load balance and the communication cost between the processing units. A good parallel algorithm should distribute the workload in such a way that the right balance is struck between load balance and communication to ensure optimal performance.

In CMC, the physical data of each star (mass, angular momentum, position etc.) are stored in a structure, a grouping of data elements. The data for N stars are stored using N structures, one for each star, and grouping them together into an array. For a system with p processors and N initial stars, we will first consider the scenario where we try to achieve ideal load balance and navely distribute the data for N/p stars to each processor. We will assume here that N is divisible by p for now, and analyze the data dependencies in the various modules of CMC for this decomposition.

1. *Time-step Calculation*:

   (a) For calculating the relaxation time of each star we need the local density, which is calculated using the masses of the 10 nearest neighboring stars on either side of the radially sorted list of stars. A parallel program requires communication between processors to exchange data of the neighboring stars that are at the extreme ends of the local array.

   (b) Calculation of the time step requires the computation of quantities in the innermost regions of the cluster, in particular the central density, and the half-mass radius. If the particles are distributed across many processors, irrespective of the specific data partitioning scheme, identification of the particle up to which the enclosing stellar mass equals half the total mass of the cluster might require communication of intermediate results between adjacent data partitions, and also introduces an inherent sequentiality in the code.

2. *Relaxation and strong interactions:*

   For the perturbation calculation, pairs of neighboring stars are chosen. Communication might be needed depending on whether the number of stars in a processor is even or odd.

3. *New orbits computation*:

   To determine the new orbits of each star we use the bisection method which involves random accesses to the gravitational potential profile of the cluster. Since the data will be distributed in a parallel algorithm, communication will be needed for data accesses that fall outside the local subset.

4. *Sorting*:

   Putting the stars in order according to their radial positions naturally needs communication irrespective of the decomposition.

5. *Potential Calculation*: The potential calculation as explained in Section 2 is inherently sequential and requires communication of intermediate results.

6. *Diagnostics and program termination:*

   The diagnostic quantities that are computed on different computational units need to be aggregated before the end of the time step to check for errors or termination conditions.

3.2. Domain Decomposition and Algorithm Design

Based on the considerations in the previous section, we design the algorithms and decompose the data according to the following scheme so as to minimize communication costs, and at the same time not degrading the accuracy of the results.

Since the relaxation time-step calculation procedure introduces additional communication cost irrespective of the choice of data partitioning, we modify it in the following way. Instead of using sliding averages for each star, we choose radial bins containing a fixed number of stars to calculate the average quantities needed for the time-step calculation. We choose a bin size of 20 which gives a good trade off between computational speed and accuracy. We tested this new time-step calculation scheme, and we did not find any significant differences compared to the previous scheme. In addition, we distribute the stars such that the number of stars per processor is a multiple of 20 (for the time-step and density estimates) for the first $(p-1)$ processors and the rest to the last processor. Since 2 is a multiple of 20, this removes any dependencies due to the interactions part too. Our choice of data partitioning also ensures a good load balance as, in the worst case, the difference between the maximum and minimum number of stars among the processors could be at most 19.

The gravitational potential ($\Phi(r)$) is accessed in a complex, data dependent manner as we use the bisection method to determine new orbits of the stars. Hence, we do not partition it among the processors, but maintain a copy of it on all nodes. We also do the same for the stellar masses, to remove the dependency in the potential calculation. This eliminates the communication required by the new orbits and potential calculations. However, it introduces the requirement to keep these data arrays synchronized at each time step and hence adds to the communication. We estimate the communication required for synchronization to be much less than what would be added by the associated dependencies without the duplicated arrays.

Most modules of the code perform computations in loops over the local subset of stars which have been assigned to the processor. Depending on the computation, each processor might need to access data from the local and duplicated arrays. While the local arrays can be accessed simply using the loop index, any accesses of the duplicated arrays (potential, position, or mass) require an index transformation. For instance, let us consider a simple energy calculation routine that calculates the energy of each star in the local array over a loop using the equation

$$E_i = \Phi_{gi} + 0.5\,(v_{r,i}^2 + v_{t,i}^2)\,.$$

where $E_i, v_{r,i}$ and $v_{t,i}$ are the energy, radial and transverse velocities of star i in the local array. The potential array having been duplicated across all processors, the potential at the position of the ith star is Φ_{gi}, where gi is the global index given by the following transformation which directly follows from our data partitioning scheme explained above:

$$gi = \begin{cases} i + id\,\left\lfloor \left\lfloor \frac{N}{n_m} \right\rfloor \frac{1}{p} \right\rfloor\, n_m + id\, n_m & \text{for } id < \left\lfloor \frac{N}{n_m} \right\rfloor \bmod p \\ i + id\,\left\lfloor \left\lfloor \frac{N}{n_m} \right\rfloor \frac{1}{p} \right\rfloor\, n_m + \left\lfloor \frac{N}{n_m} \right\rfloor \bmod p\, n_m & \text{for } id \geq \left\lfloor \frac{N}{n_m} \right\rfloor \bmod p \end{cases}$$

where id the id of the processor that is executing the current piece of code, which ranges between 0 to $p-1$, n_m is the number of which we would want the number of stars in each processor to be a multiple of, which as per our choice, is 20, and the terms between $\lfloor \ldots \rfloor$ are rounded to the lowest integer.

3.3. Parallel Flow

The following gives an overview of the parallel workflow:

1. Initial partitioning of the star data and distribution of duplicated arrays (mass, and

radial positions)

2. Calculate potential

3. Perform interactions, stellar evolution, and new orbits calculation

4. Sort stars by position in parallel

5. Re-distribute/load-balance data to concur with domain decomposition

6. Synchronize duplicated arrays (mass, and radial positions)

Then the whole procedure is repeated starting from step 2. The first step is to distribute the initial data among processors as per our data partitioning scheme mentioned in Section 3.2. This is done only once per simulation. This also includes the distribution of a copy of the duplicated arrays. In Section 2, we saw that the calculation of the potential is inherently sequential requiring communication, since it is calculated recursively starting from the outermost star and using the intermediate result to compute the potential of the inner stars. However, since now every processor has an entire copy of the potential, the positions and mass arrays, it can calculate the potential independently. This does not give a performance gain since there is no division of workload, however nullifies the communication requirement. Performing interactions, stellar evolution and new orbits calculation too don't require any communication whatsoever due to our choice of data partitioning and use of duplicated arrays. We use sample sort as the parallel sorting algorithm which is described in much greater detail in a subsequent section. With a wise choice of parameters, the sample-sort algorithm can provide a near equal distribution of particles among processors. However, since we require the data to be partitioned in a very specific way, following the sort, we employ a re-distribution/load-balancing phase to redistribute the sorted data as per our chosen domain decomposition. Sorting and re-distribution are steps that naturally require the most communication. Before the beginning of the next time-step, we synchronize the duplicated arrays on all processors which requires message passing. Some non-trivial communication is also required at various places in the code to collect and synchronize diagnostic values.

3.4. Sorting

The input to a parallel sorting algorithm consists of a set of data elements (properties of N stars in our case), each having a key (radial positions in our case) based on which the data need to be sorted. An efficient parallel sorting algorithm should collectively sort data owned by individual processors in such a way that their utilization is maximum, and at the same time the cost of redistribution of data across processors is kept to a minimum. In order to implement a parallel sorting algorithm, there are a wide array of solutions to consider. Each of these solutions cater to a parallel application and/or in some cases a particular machine architecture/platform. In general, a parallel programmer should consider many different design decisions with careful consideration of the application characteristics. The proper assessment of application knowledge often can suggest which initial distributions of data among processors are likely to occur, allowing the programmer to implement a sorting algorithm that works effectively for that application.

The importance of load balance is also immense, since the application's execution time is typically bound by the local execution time of the most overloaded processor. Since our choice of domain decomposition requires a fairly good load balance, our sorting algorithm should ensure that the final distribution of keys among processors closely agree with our decomposition. This is a challenge since during their evolution, dense star clusters have a very strong density contrast, and stars are very unevenly distributed radially with a substantial number of stars in the high density core, and a the rest in the extremely low density halo. A good parallel sorting algorithm for our application should be able to judge the distribution of keys and perform the sort accordingly so that near-equal amount of data ends up in each processor at the end of the sorting phase.

Sample Sort is a splitter-based parallel sorting algorithm that performs a load balanced splitting of the data by sampling the global key set. This sampling helps judge the initial distribution of keys and accordingly perform the sort, hence resulting in a near-equal distribution of data among processors. Given N data elements distributed across p processors, sample sort consists of 5 phases, shown in Figure 2:

1. **Sort Local Data**: Each processor has a contiguous block of data in accordance with our data partition (close to N/p in number, see Section 3.2). Each processor, in parallel, sorts this local block using sequential Quicksort.

2. **Sampling**: All p processors, in parallel, uniformly sample s keys to form a representative sample of the locally sorted block of data. These set of p samples, of size s from each processor, are collected on one designated processor. This aggregated array of samples represent the distribution of the entire set of keys.

3. **Splitter Selection**: The combined sample key array is sorted, and keys at indices $s, 2s, 3s, ..., (p-1)s$ are selected as splitters and are broadcasted to all processors.

4. **Exchange partitions**: The positions of the $(p-1)$ splitter points in the local array are determined by each processor using binary search; this splits the local array into p partitions. In parallel, each processor retains the ith partition and sends the jth partition to the jth processor i.e. each processor keeps 1 partition and distributes $(p-1)$ partitions. At the same time it receives 1 partition from every other processor. This might not be true always, particularly in cases of a poor choice of sample size s, some splitter points might lie outside the local data array and hence some processors might not send data to all $(p-1)$ processors but only a subset of them. However, for

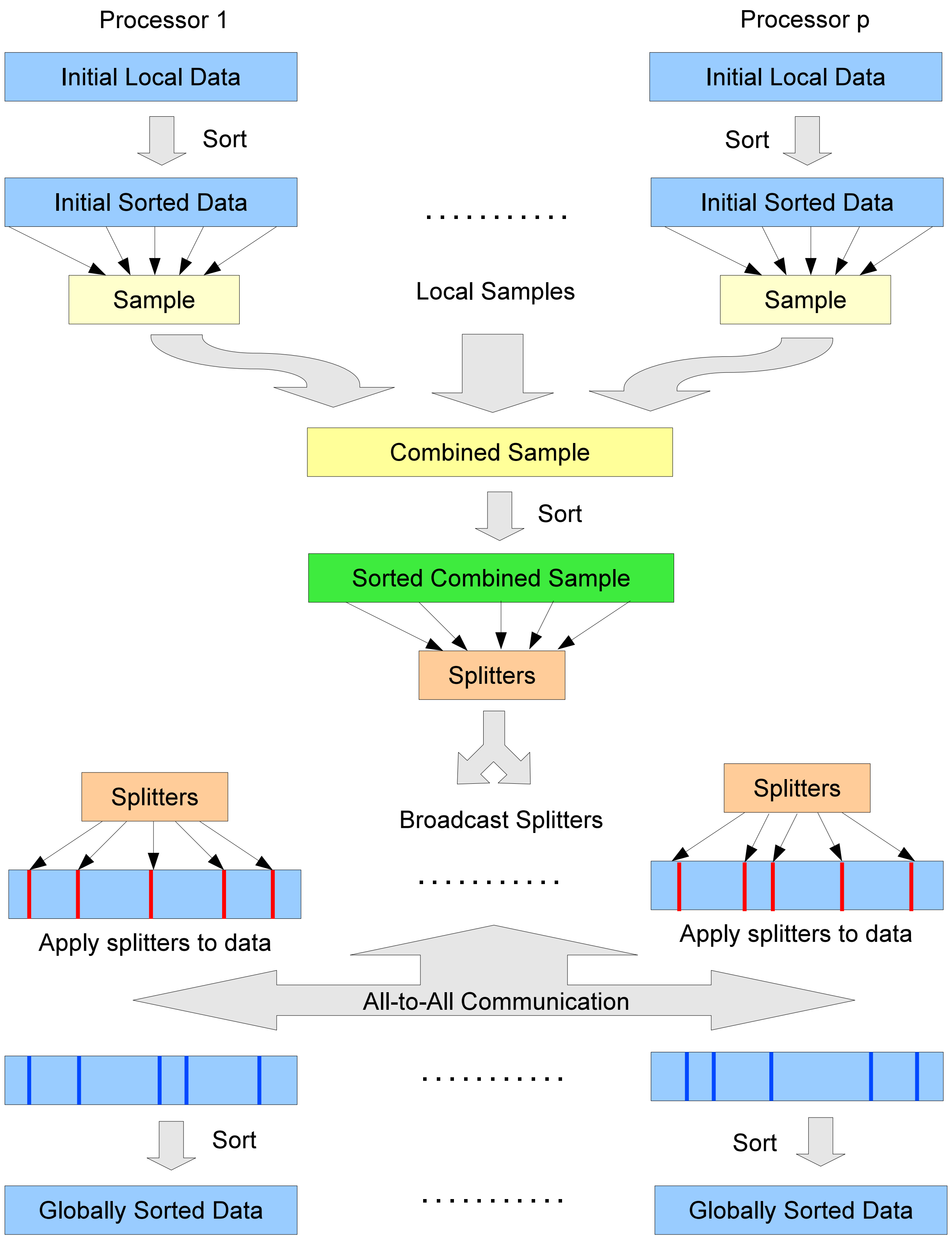

Fig. 2.— The Sample Sort Algorithm

the current discussion we will assume a good choice of sample size is made.

5. **Merge Partitions**: Each processor, in parallel, merges its p partitions into a single list and sorts it.

One chooses a sufficiently large enough value for the sample size s so as to sample the distribution of keys accurately, and hence this parameter varies depending on the distribution of keys as well the data size N. More comments on the choice of sample size can be found in Li et al. (1993).

Let us now try to derive the time complexity of sample sort on a hypercube parallel architecture with cut-through routing. The local sort requires $\mathcal{O}(\frac{N}{p}\log(\frac{N}{p}))$ since there are close to N/p keys per processor. The selection of s sample keys requires $\mathcal{O}(s)$ time. Collecting s keys from p processors on to one of the processors is a single-node gather operation for which the time required is $\mathcal{O}(sp)$. The time to sort these sp samples is $\mathcal{O}((sp)\log(sp))$, and the time to select $(p-1)$ splitters is $\mathcal{O}(p)$. The splitters are sent to all the other processors using an one-to-all broadcast which takes $\mathcal{O}(p\log p)$ time. To place the splitters in the local array using binary search takes $\mathcal{O}(p\log(\frac{N}{p}))$. The all-to-all communication that follows costs a worst case time of $\mathcal{O}(\frac{N}{p}) + \mathcal{O}(p\log p)$. So the time complexity of the entire algorithm becomes

$$\mathcal{O}\left(\frac{N}{p}\log\frac{N}{p}\right) + \mathcal{O}((sp)\log(sp)) + \mathcal{O}\left(p\log\frac{N}{p}\right) + \mathcal{O}(N/p) + \mathcal{O}(p\log p)\ . \qquad (9)$$

3.5. Data Redistribution

In theory, , with a good choice of sample size, sample sort guarantees to distribute the particles evenly among processors within a factor of two (Li et al. 1993). However, we would like to partition the data in such a way that each processor has close to N/p elements, and at the same time being multiple of 20. Since the final distribution of data among processors after sample sort is not deterministic, we include an additional communication phase to ensure the required data distribution is maintained.

We first calculate the global splitter points that would partitioned the entire data among processors as per our required data partitioning scheme. We also do a parallel prefix reduction (MPI_Exscan), so each processor knows the cumulative number of stars that ended up in processors before it. Using this, it can calculate the range of global indices corresponding to the list of stars it currently holds. Now, each processor checks if any of the global splitter points other than its own map on to its local array, and if they do, it marks the corresponding chunk of data to be sent to the respective processor. The, we do an all-to-all communication to send the data, which is then rearranged locally within each processor.

Let us consider and example where there are 4 processors and they receive 100, 130, 140 and 80 respectively after the sort phase. For a total of 450 stars to be divided among 4 processors, the expected distribution would be 120, 120, 100 and 110 stars respectively as per our scheme. The corresponding global splitter points would be 120, 240, and 340. By doing the prefix reduction on the received number of stars, processor 3, for instance, knows there are in total 230 stars in processors before it. Since it received 140 stars after sorting, it also calculates that it has stars with indices between 231 - 370. Now, two global splitter points i.e., 240 and 340 of processors 2 and 4 lie with this index range, and hence the corresponding stars i.e., 231 - 240 and 341 - 370 need to be sent to processors 2 and 4 respectively. These data chunks are exchanged by performing an all-to-all communication which is followed by a rearrangement if required.

3.6. Parallel Random Number Generation

The accuracy of results of a parallel Monte Carlo code depends in general on both the quality of the pseudo-random number generators (PRNGs) used and the approach adopted to generate them in parallel. In our case we need to sample events with very low probability, such as physical collisions between stars or binary interactions, which makes it necessary for the generated random numbers to be distributed very evenly. More specifically, given N_r random numbers uniformly distributed in the interval $(0, 1)$, there should be one random number in each sub-interval of size $1/N_r$. A special class of random number generators for which this property holds in even higher dimensions are the maximally equi-distributed generators and we choose here the popular and fast combined Tausworth linear feedback shift register PRNG in L'Ecuyer (1999).

PRNGs use a set of state variables which are used to calculate random numbers. Every time a random number is generated, the values of these states are updated. A PRNG can be initialized to an arbitrary starting state using a random seed. When initialized with the same seed, a PRNG will always produce the same exact sequence. The maximum length of the sequence before it begins to repeat is called the period of the PRNG. The combined Tausworthe PRNG we are using here has a period of $\approx 2^{113}$ (L'Ecuyer 1999).

While generating random numbers in parallel, special care has to be taken to ensure statistical independence of the results calculated on each processor. For instance, to implement a parallel version, we could simply allocate separate state information for the PRNGs on each processor and initialize them with a different random seed. However, choosing different seeds does not guarantee statistical independence between these streams.

An efficient way to produce multiple statistically independent streams is to divide a single random sequence into subsequences, with their starting states calculated using jump functions (Collins 2008). Taking a starting seed and a jump displacement, D, as inputs, jump functions can very quickly generate the Dth state of the random sequence . We use this method to generate multiple starting states, one for each processor, by repeatedly applying the jump function and saving intermediate results. The jump displacement is chosen as the maximum number of random numbers each processor might require for the entire simulation while still providing for a sufficiently large number of streams. Based on that we choose $D = 2^{80}$.

3.7. Implementation

CMC is written in C, with some parts in Fortran. We use the Message Passing Interface (MPI) library to handle communication. The MPI standard is highly portable to different parallel architectures, and available on practically every supercomputing platform in use today. The most common MPI communication calls used in our code are:

1. *MPI_Allreduce/MPI_Reduce*

   MPI_Reduce combines the elements of a distributed array by cumulatively applying a user specified operation as to reduce the array to a single value. For instance, when the operation is addition then the resulting value is the sum of all elements. MPI_Allreduce is MPI_Reduce with the difference that the result is distributed to all processors. The call is used in the following parts of the code:

   (a) *Diagnostics and program termination:* accumulating diagnostic quantities such as the half-mass radius, r_h, and the core radius, r_c.

   (b) time-step calculation: to find the minimum time-step of all stars across processors.

   (c) Sorting and data redistribution: Since stars are created and lost throughout the simulation, N is not a constant and changes during a time-step. It is calculated during the sorting step by summing up the local number of stars on each processor.

2. *MPI_Allgather/MPI_Gather*

   In MPI_Gather each process sends the contents of its local array to the root, or master, process, which then concatenates all received arrays into one. In MPI_Allgather this concatenated array is distributed to all nodes. The calls are used in the following parts of the code:

   (a) Synchronization of duplicated arrays, i.e., $\Phi(r)$ and the stellar masses.

   (b) Sorting and data redistribution: to gather samples contributed by all processors on to a single node. See Section 3.4 for details.

3. *MPI_Alltoall*

   In MPI_Alltoall the send and receive array is divided equally into p sub-arrays, where p is the number of processors. The position of each sub-arryay within the send or receive array determines to or from which processor the data is sent or received, respectively. MPI_Alltoall is only used in “Sorting and data redistribution”. See Section 3.4 for details.

4. *MPI_Bcast*

   In MPI_Bcast an array is sent from the root, or master, node to all other processes. Used in “Sorting and data redistribution” to communicate the new splitter points from a single specified processor to all other.

5. *MPI_Scan/MPI_Exscan*

   MPI_Scan essentially carries out a reduction as in MPI_Allreduce except that processor i receives the result of the reduction over the data of processors 0 to i. In

MPI_Exscan the data is reduced over processors 0 to i-1. MPI_Scan/Exscan is used in “Sorting and data redistribution”. See Section 3.5 for details.

We also use a number of optimizations for the MPI communication calls. Some examples include using MPI derived datatypes for data packing and sending, and combining multiple parallel reduction calls for the diagnostic quantities by packing all the data into a single buffer and performing the reduction together using a single call which is more efficient. However, the overlapping of communication calls with computation we did not explore so far, but intend to do so in the future.

4. Results

All our test calculations were carried out on Hopper, a Cray XE6 supercomputer at NERSC[1] that has a peak performance of 1.28 Petaflops, 153,216 processor-cores for running scientific applications, 212 TB of memory, and 2 Petabytes of online disk storage.

4.1. Accuracy and Reproducibility

In the parallel version, since we use jump functions, the way random numbers are assigned to stars is different from the serial version. This would bring in a problem of inconsistency in the results between serial and parallel runs, leaving us with no simple way to verify the correctness of the results of our parallel version. We tackle this problem by changing the serial version such that it uses the same mapping of random numbers to stars as followed in the parallel version.

We ran test simulations for 50 time-steps with the parallel version and serial versions

[1]http://www.nersc.gov/

with N = 10^5, 10^6, and 10^7 stars, and compared positions and masses of every star in the cluster. We found them to be matching accurately down to the last significant digit (all variables are in double precision). We also compared a few diagnostic quantities, and they were matching too except for the last four significant digits. This slight loss in accuracy is due to the *MPI_Reduce* calls, which perform cumulative operations (sum, max, min etc.) on data distributed among the processors. This introduces different round-off errors since one does not have control over the order in which the data aggregation is done.

4.2. Comparison to Theoretical Predictions

In order to verify that our code reproduces well-known theoretical results, we calculate the evolution of single-mass Plummer spheres (Binney & Tremaine 2008) until core collapse (without any binaries or stellar evolution). Using 10^5, 10^6, and 10^7 stars, this is the first time that self-consistent N-body simulations covering three orders of magnitude in N have been carried out. We used 128, 256 and 1024 processors for these runs, respectively, which deliver peak performance for the three cases (see Section 4.3).

One remarkable property realized early on is that the cluster evolution proceeds in a self-similar fashion, that is, the cluster density profiles differ only in scale and normalization at any given time (e.g., Cohn 1980; Binney & Tremaine 2008). This can be clearly seen in Figure 4, where we plot the density profile of the cluster with $N = 10^5$ at various times during its evolution to core collapse. For each profile we observe a well-defined core and a power law density profile, $\rho \propto r^{-\gamma}$, with $\gamma \approx 2.5$, in good agreement with the literature value of $\gamma = 2.23$ (Cohn 1980; Heggie & Stevenson 1988; Takahashi 1995) shown by the dashed line.

Figure 3 shows the core radius, $r_{\rm c}(t)/r_{\rm h}(t)$, as well as the core density evolution, $\rho_c(t)$,

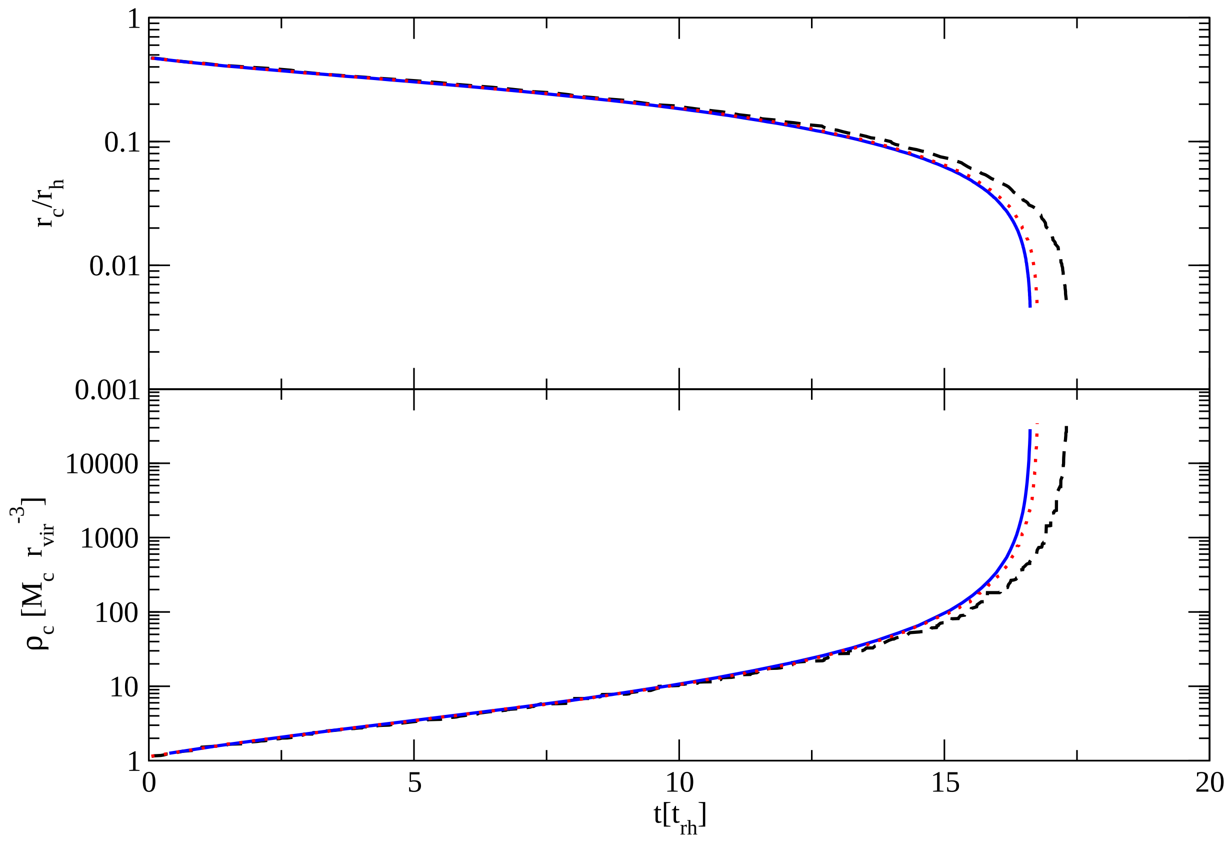

Fig. 3.— Evolution of an isolated Plummer model showing the ratio of core radius to half-mass radius (top) and the core density (bottom). Time is in initial half-mass relaxation times. The various lines represent different particle numbers, $N = 10^5\,(\text{dashed}), 10^6\,(\text{dots}),\ 10^7\,(\text{solid})$.

for the models with $N = 10^5$, 10^6 and 10^7 stars. One can immediately see that all three clusters reach core collapse at similar times, with $t = t_{cc} \simeq 17.4 t_{\rm rh}$,$16.7 t_{\rm rh}$ and $16.6\, t_{\rm rh}$, respectively, where $t_{\rm rh}$ is the initial half-mass relaxation time. Thus, the core collapse times are not only in very good agreement with previously published results that all lie between 15 and 18 t_{rh} (see, e.g., Freitag & Benz 2001, for an overview), but also confirm that our code can reproduce the scaling of $t_{\rm cc}$ with $t_{\rm rh}$ within $\approx 10\%$ over three orders of magnitude in N. The scaling behavior becomes even better, with deviations $< 1\%$, if only the runs with $N \geq 10^6$ are considered. The larger deviation in $t_{\rm cc}$ between the $N = 10^5$ run and the other two are probably because of the larger stochastic variations in low N runs.

Another consequence of the self-similarity of the cluster evolution to core collapse is that $-\gamma \approx \log(\rho_c(t))/\log(r_{\rm c}(t))$ (Binney & Tremaine 2008), which means that the decrease in $r_{\rm c}$ leads to an increase in $\rho_{\rm c}$ at a rate that is related to the shape of the density profile. Indeed, from Figure 3 one can see that the shape of $\rho_{\rm c}(t)$ mirrors the shape of $r_{\rm c}(t)$ as expected. Furthermore, by fitting $\rho_{\rm c}$(t) and $r_{\rm c}(t)$ with an exponential function up to $12\, t_{\rm rh}$, we find that $\rho_{\rm c} \sim r_{\rm c}^{-2.4}$, which is close to the power-law slope of -2.5 we find for $\rho(r)$ for $r > r_{\rm c}$.

Apart from the self-similar behavior, we also find that there is very little mass loss (less than 1%), and hence very little energy is carried away by escaping stars, in agreement with theoretical expectations (e.g., Lightman & Shapiro 1978). Finally, we find that our code conserves total energy to better than 1% throughout the entire simulation.

4.3. Performance Analysis

We tested out our parallel code for 3 clusters configurations with N= 10^5, 10^6, and 10^7, all of them having an initial Plummer density profile. We used 1 to 1024 processors

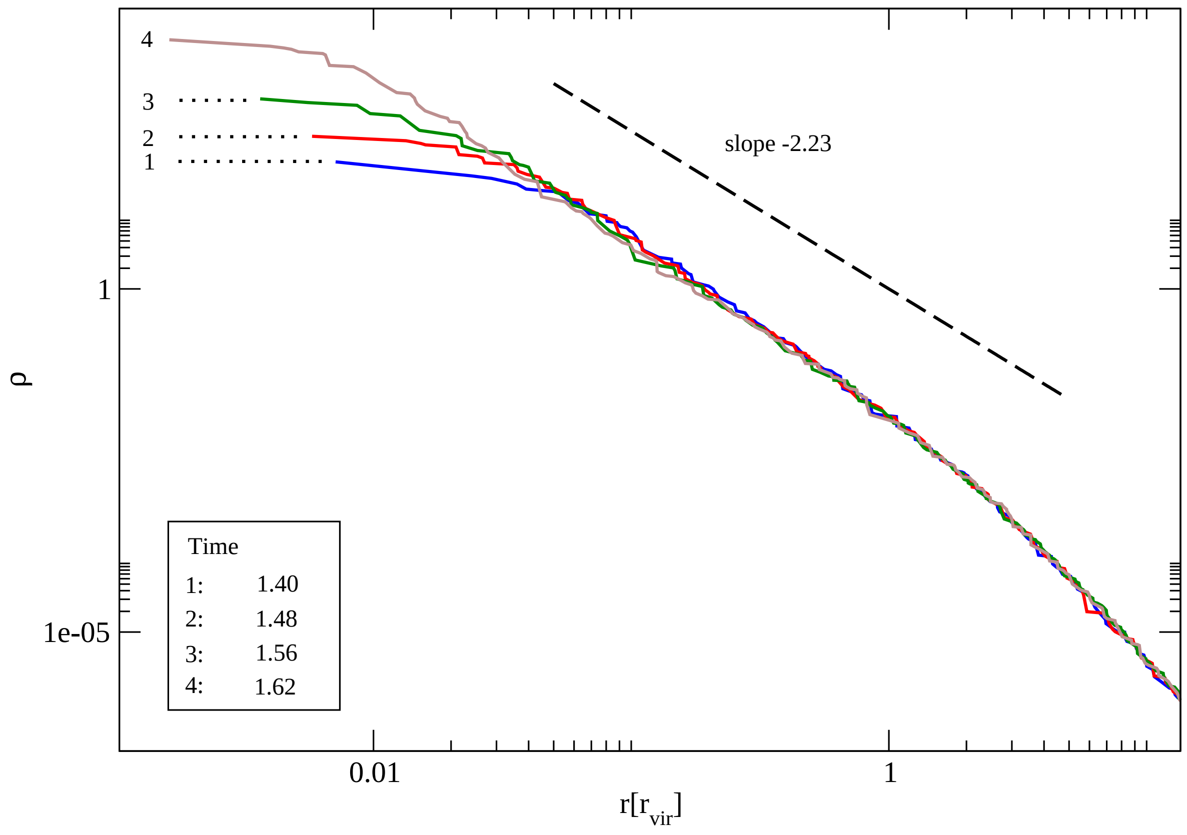

Fig. 4.— Evolution of the density profile at various times during core collapse for the $N = 10^5$ run. The dashed line shows the slope of the expected power-law density profile (Heggie & Stevenson 1988; Cohn 1980).

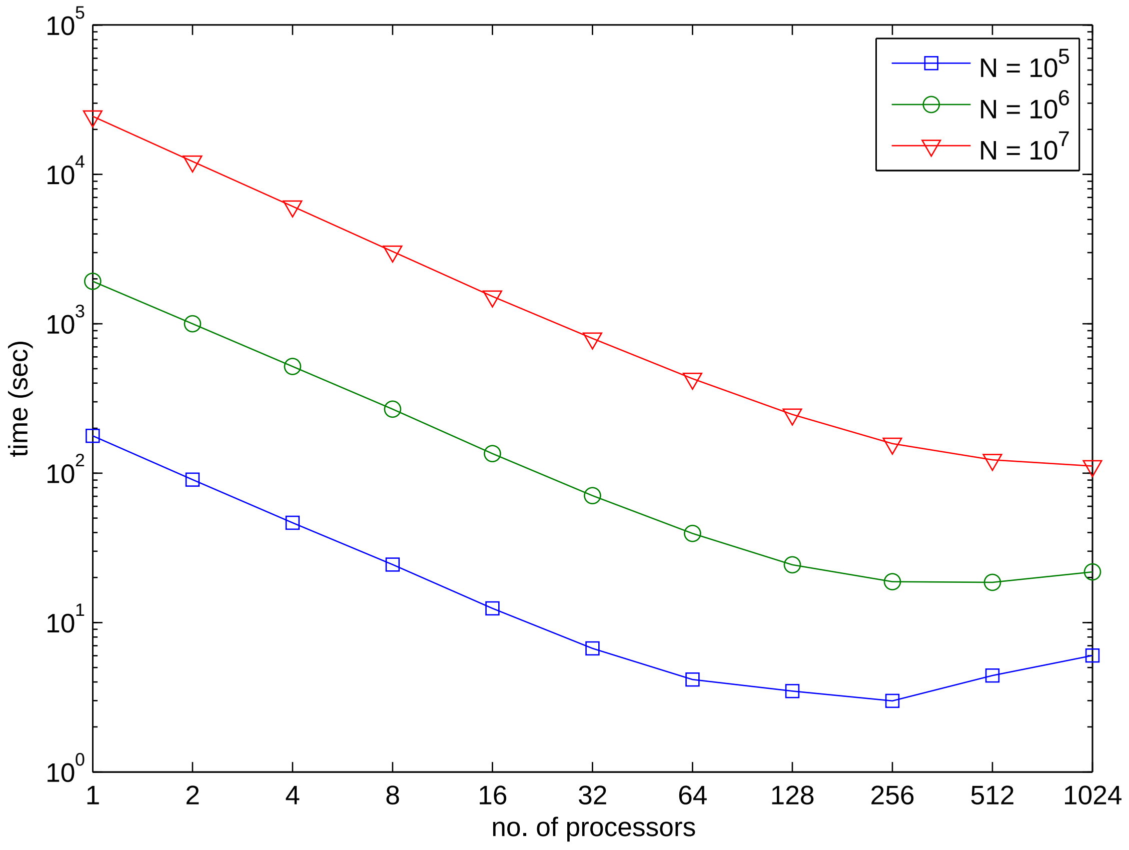

Fig. 5.— Scaling of the wall clock time taken with the number of processors for a parallel simulation of up to 50 time-steps. The various lines represent different particle numbers (see legend).

and measured the total time taken, and time taken by various parts of the code for up to 50 time-steps. For the sample size s for the sample sort, we chose 128, 256 and 1024 respectively for the three N values.

The timing results are shown in Figure 5 and a corresponding plot of the speedups in Figure 6. These results do not include the time taken for initialization including reading of the initial conditions. We can see that the speedup is nearly linear up to 64 processors for all three runs, after which there is a gradual decrease followed by saturation. For the 10^5, and 10^6 case, we also notice a dip after the saturation which is also expected for the 10^7 case for a larger number of processors than we consider here. We also see that the number of processors for which the speedup peaks is different for each run, which gradually increases with N. The peak is seen at 256 processors for the 10^5 run, somewhere between 256 and 512 for the 10^6 run, and 1024 for the 10^7 run. The maximum speedups observed are around $60\times$, $100\times$, and $220\times$ for the three cases respectively.

Figure 7 shows the scaling the time taken by various modules of our code for the 1 million run. One can observe that the dynamics, stellar evolution, and orbit calculation modules achieve perfectly linear scaling. The ones that do not scale as well are the sorting and "Diagnostics and Program Termination". part which involve the computation and communication of diagnostic book-keeping quantities. These quantities need to be cumulatively reduced and are difficult to parallelize efficiently. As the number of processors increase, the linear scaling of the former three parts of the code reduces their time to very small values, in turn letting the parts that do not scale dominate the runtime. This is the reason for the trend observed in the total execution time and speedup plots. We can also particularly see that the time taken by sorting starts to increase after reaching a minimum, and this explains a similar observation in those plots too.

Figure 8 shows the experimental timing results of our sorting phase for the three N

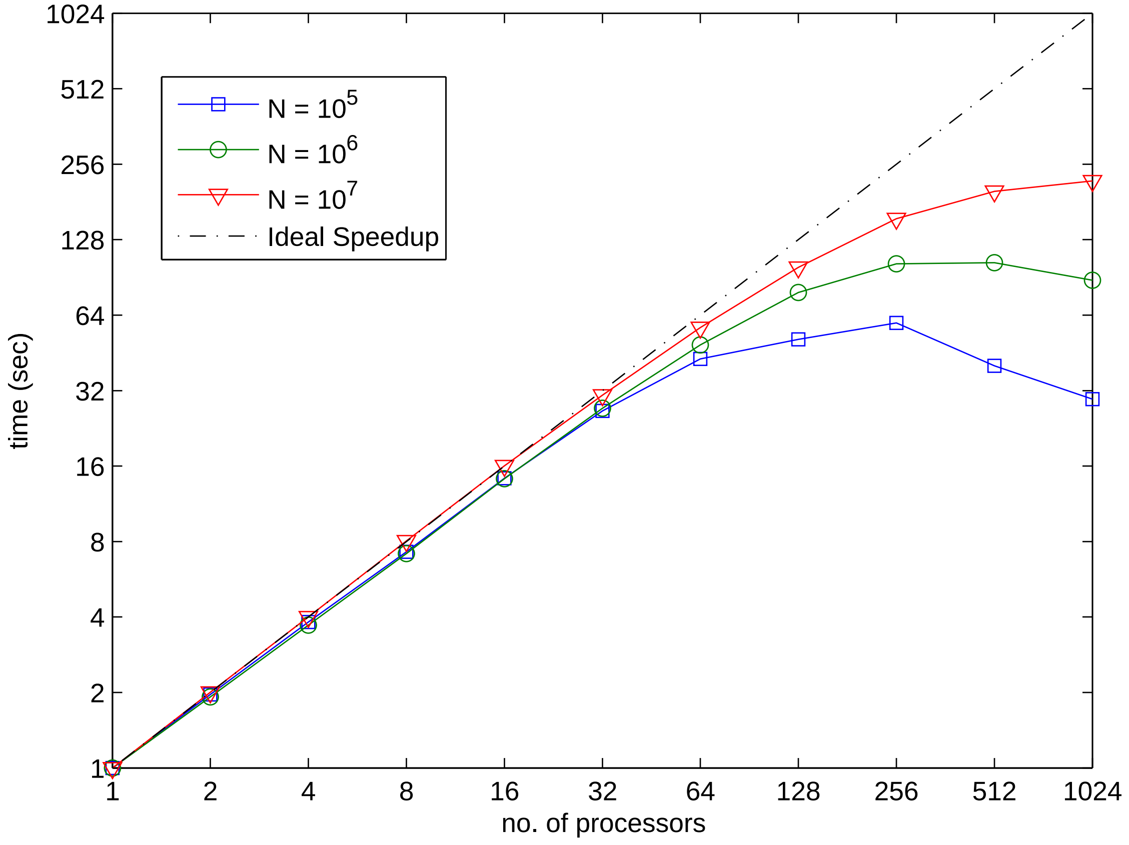

Fig. 6.— Speedup of our parallel code as a function of the number of processors. The various lines represent different particle numbers (see legend). The diagonal dashed line gives the ideal speedup.

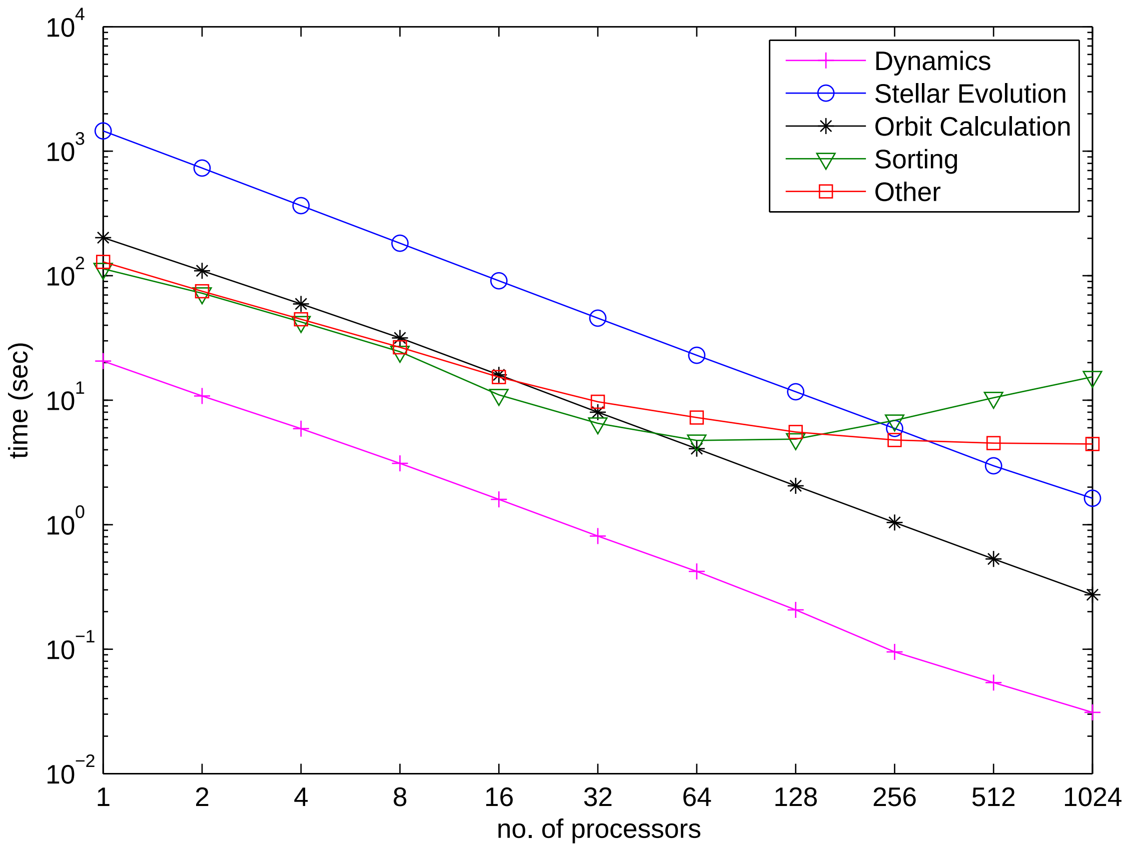

Fig. 7.— Time taken by the various parts of the code for a parallel run with 1 million stars. The various lines represent the different modules of the algorithm (see legend).

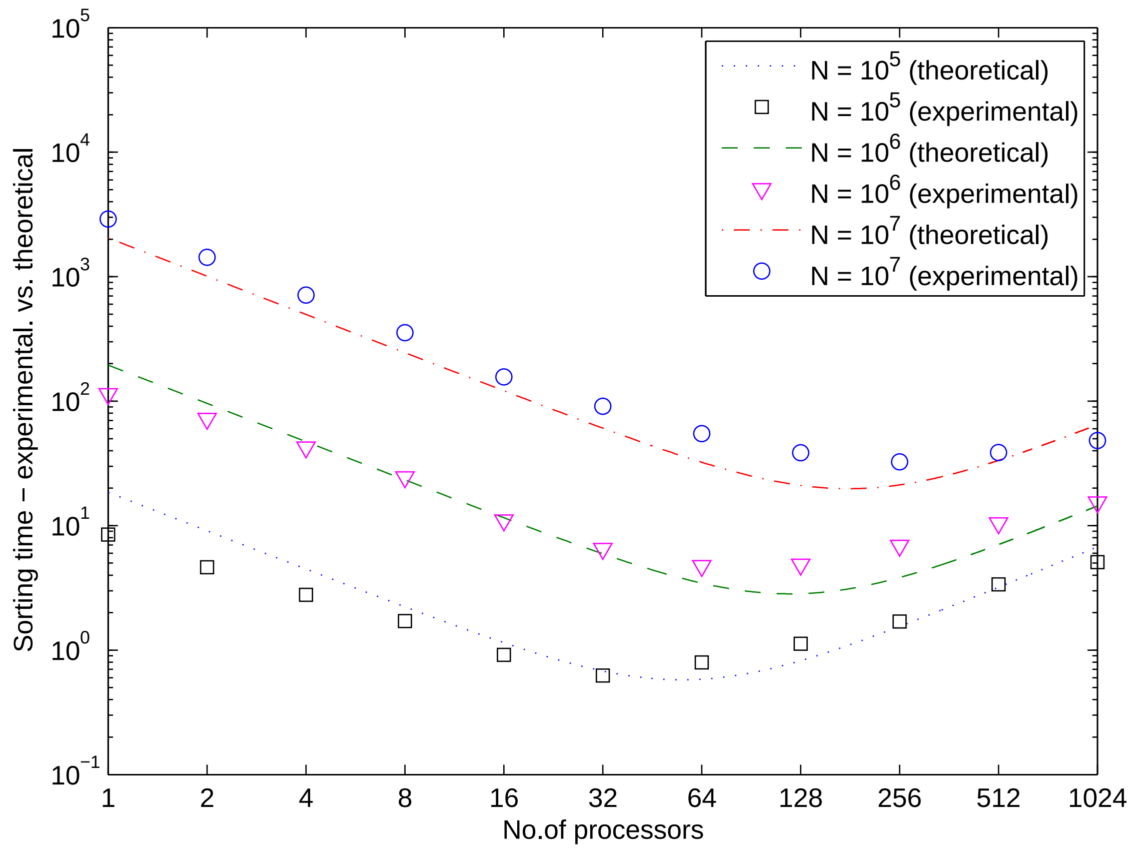

Fig. 8.— Time taken by the sorting routine of the parallel code plotted against the theoretical time complexity of sample sort based on Equation 9 for various values of N.

values plotted against the theoretical values calculated using Equation 9. While calculating the theoretical values, we used an appropriate proportionality constant for the all-to-all communication phase. Since the entire star data is communicated during this phase and not just the keys, and a data size of a single star is 45 times greater than that of a single key in our code, we multiply the $\Theta(N/p)$ term of Equation 9 with a proportionality constant of 45. We used a sample size of $s = 128, 256$ and 1024 for $N = 10^5$, 10^6 and 10^7, respectively. We see that for all three cases, the expected time linearly decreases reaching a minimum after which it shoots back upward. This implies that for every data size, there is a threshold for the number of processors that would achieve optimal performance beyond which it will only worsen. This is primarily due to the fact that as the number of processors increase, smaller amounts of data are distributed across many processors, which makes the communication overhead becomes much more noticeable and dominant. From Figure 8, we also see that our implementation very closely follows the trend predicted by the theoretical complexity formula. In addition, the number of processors at which maximum performance is attained match fairly closely too. We noted earlier that the total execution time followed a similar trend, but the places at which the minimum execution time was observed are not the same, but shifted to the right. This is due to the influence of the linearly-scaling parts of the code which push these points to the right until the substantially large time taken by the sorting and the "other" computations dominate the runtime. The "other" computations include potential calculation and a number of collective communication calls to aggregate book-keeping quantities across processors. The time taken for potential calculation remains unchanged due to parallelization since every processor computes the entire potential array, which takes constant time irrespective of the number of processors used. However, there might be potential to improve the scaling by interleaving the collective communication calls with computation.

5. Conclusions

We presented a new parallel code, CMC, for simulating collisional N-body systems with up to $N \sim 10^7$. In order to maintain a platform independent implementation, we adopt the Message Passing Interface (MPI) library for communication. The parallelization scheme uses a domain decomposition that guarantees a near-equal distribution of data among processors to provide a good balance of workload among processors, and at the same time minimizes the communication requirements by various modules of the code. Our code is based on the Hnon Monte Carlo method , with algorithmic modifications including a parallel random number generation scheme, and a parallel sorting algorithm. We present the first ever self-consistent N-body simulations of star clusters with N spanning three orders of magnitude, from 10^5 to 10^7. The core collapse times obtained in our simulations are in good agreement with previous studies, confirming the correctness of our code. We also test our implementation for the previously considered N values on 1 to 1024 processors. The code scales linearly up to 64 processors for all cases considered, after which it saturates, which we find to be characteristic of the parallel sorting algorithm. The overall performance of the code is impressive, delivering maximum speedups of up to 220$\times$ for $N = 10^7$.

Interesting future lines of work may include reducing the communication overhead by overlapping communication with computation. Running simulations with even higher N values will allow us to test how the code scales, and to identify any bottlenecks. In addition, we can employ GPUs to accelerate these bottlenecks, and develop with a hybrid code which can run on heterogeneous distributed architectures with GPUs. With progress on these lines, we may be able to reach the domain of nuclear star clusters for the first time. With their much larger masses and escape velocities, nuclear star clusters are likely to retain many more stelllar-mass black holes than globular clusters, and, thus, might significantly contribute to the black hole binary merger rate, as well as to the gravitational wave detection rate of advanced LIGO (Laser Interferometer Gravitational wave Observatory) (Miller & Lauburg 2009). Therefore, the study of nuclear star clusters with a fully self-consistent dynamical code such as CMC has the potential to make strong predictions for future gravitational wave detection missions.

This work was supported by NASA Grant NNX09A036G to F.A.R. We also acknowledge partial support from NSF Grants PHY05-51164, CCF-0621443, OCI-0724599, CCF-0833131, CNS-0830927, IIS- 0905205, OCI-0956311, CCF-0938000, CCF-1043085, CCF-1029166, and OCI-1144061, and from DOE Grants DE-FC02-07ER25808, DE-FG02-08ER25848, de-sc0001283, de-sc0005309, and de-sc0005340.

A. GPU Implementation of the Orbit Computation

An optional feature of the CMC code is the GPU acceleration of the orbit computation that consists of finding peri- and apastron of each stellar orbit and sampling a new orbital position (see Section 2). As we have shown, the time complexity of both parts is $N \log N$, and each orbit and new position for one star can be independently determined from the other stars. This makes the orbit computation particularly suited to be calculated on a GPU, not only because of the inherent parallelism of the algorithm, but also for the large number of memory accesses, which also scale as $\sim N \log N$, and, thus, allow to take advantage of the fast GPU memory.

Based on the structure of the algorithm, our implementation assigns one thread on the GPU to do the computations for one star. This ensures minimal data dependency between the threads since the same set of operations are performed on different data, and makes the bisection method and rejection technique implementations naturally suited for SIMD (Single Instruction, Multiple Data) architectures, such as the GPU. In the following we describe the specific adaptations of the serial implementation of the algorithms to the GPU architecture and present performance results.

A.1. Memory Access Optimization

To harness the high computation power of the GPU, it is very essential to have a good understanding of its memory hierarchy in order to develop strategies that reduce memory access latency. The first step towards optimizing memory accesses is to ensure that memory transfer between the host and the GPU is kept to a minimum. Another important factor that needs to be considered is global memory coalescing in the GPU which could cause a great difference in performance. When a GPU kernel accesses global memory, all threads in groups of a half-warp access a bank of memory at the same time (Nvidia 2010). Coalescing of memory accesses happens when data requested by these groups of threads are located in contiguous memory addresses, in which case they can be read in one (or very few number of) access(es). Hence, whether data is coalesced or not has a significant impact on an application's performance as it determines the degree of memory access parallelism. In CMC, the physical properties of each star are stored in a C structure, containing 46 double precision variables. The N stars are stored in an array of such C structures.

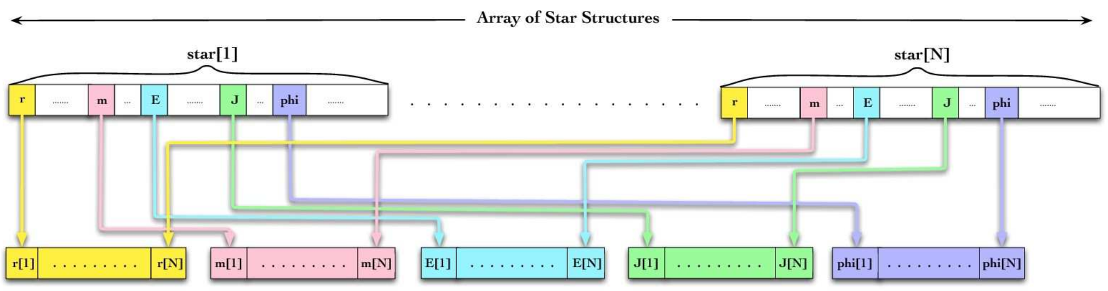

Fig. 9.— Data coalescing strategy used to strip the original star data structure and pack into contiguous arrays before transferring them to the GPU.

Figure 9 gives a schematic representation of the data reorganization. At the top, the original data layout is shown, i.e., an array of structures. The kernels we parallelize only require 5 among the 49 variables present in the star structure: radial distance, r, mass m, energy, E, angular momentum, J, and potential at r, ϕ, which are shown in different color. To achieve coalesced memory accesses, we need to pack the data before transferring it to the GPU in a way that they would be stored in contiguous memory locations in the GPU global memory. A number of memory accesses involve the same set of properties for different stars being accessed together by these kernels since one thread works on the data of one star. Hence, we extract and pack these into separate, contiguous arrays, one for each property. This ensures that the memory accesses in the GPU will be mostly coalesced. Also, by extracting and packing only the 5 properties required by the parallel kernels, we minimize the data transfer between the CPU and GPU.

A.2. Random Number Generation

For the generation of random numbers we use the same combined Tausworthe generator and parallel implementation scheme as described in Section 3.6. That is, for each thread that samples the position of a star, there is one random stream with an initial state that has been obtained by jumping multiple times in a seeded random sequence to ensure statistical independence between streams. As we will see later, to achieve optimal performance, 6000 to 8000 threads, and, thus, streams, are required. This can be easily accomodated by one random sequence, as for our jump distance of 2^{80} and a random number generator period of $\approx 2^{113}$, $\approx 10^{10}$ non-overlapping streams can be generated. Once generated on the host, the initial stream states are transferred to the GPU global memory. Each thread reads the respective starting state from the memory and produces random numbers independently.

A.3. Performance Results

All our simulations are carried out on a 2.6 GHz AMD PhenomTM Quad-Core Processor with 2 GB of RAM per core and an NVIDIA GTX280 GPU, with 30 multiprocessors, and 1 GB of RAM. The algorithms have been written in the CUDA C language, and were compiled with the version 3.1 of the CUDA compiler. All computations are done in double precision, using the only Double Precision Unit (DPU) in each multiprocessor on the GTX280 GPU.

We collect the timing results for 5 simulation timesteps of a single-mass cluster with a Plummer density profile, and sizes ranging from 10^6 to 7×10^6 stars, encompassing nearly all globular cluster sizes (e.g., McLaughlin & van der Marel 2005).

Figure 10 compares the GPU and CPU run-times. Figure 11 shows the speedup of the 'new orbits calculation' part and the bisection and rejection kernels individually. All speedup values are with respect to the code performance on a single CPU. We see that the average speedups for the rejection and bisection kernels are 22 and 31, respectively. This is due to the difference in the number of floating point operations between the two kernels which is a factor of 10. This makes a major difference on the CPU but not on the GPU as it has more arithmetic logic units (ALUs). Indeed, the bisection and rejection kernels take about equal amount of time on the GPU for all N. This also indicates that the performance of these kernels is only limited by the memory bandwidth as they roughly require the same amount of global memory accesses.

We also observe that the total speedup increases slightly as the data size increases.

In general, we obtain very good scalability. Analyzing the dependence of the run-time on N in Figure 10 we find that the GPU run-times follow closely the kernel's complexity of $\mathcal{O}(N \log N)$. The run-times on the CPU, on the other hand, have a steeper scaling with$N$,

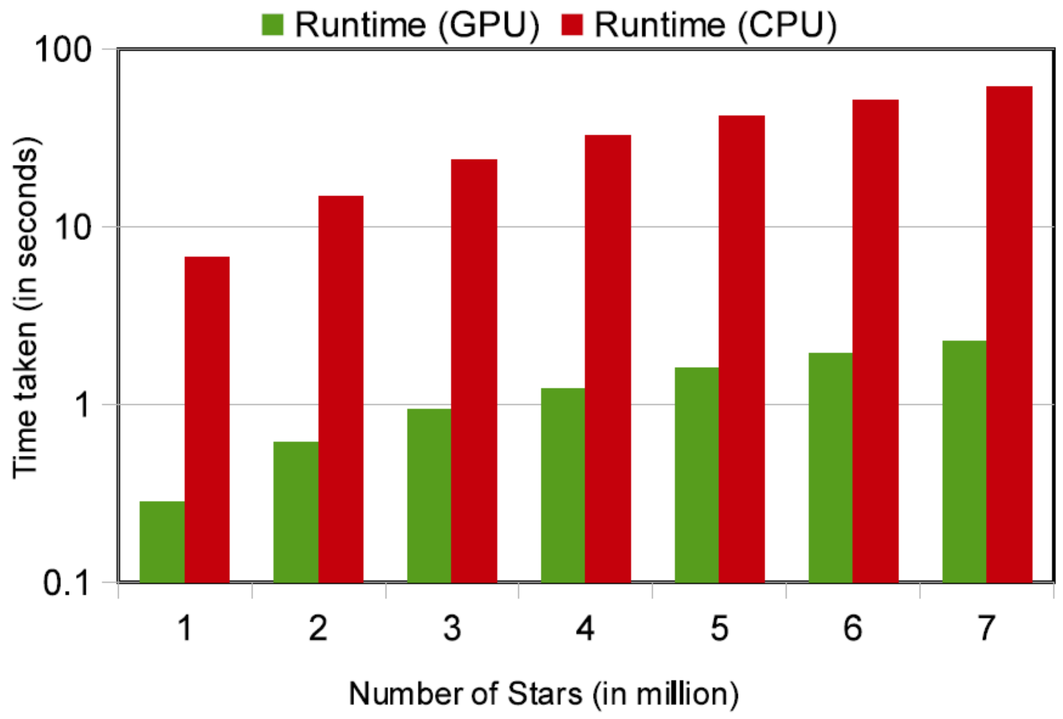

Fig. 10.— Comparison of total run-times of the sequential and parallelized kernels for various N.

such that the run with $N = 7 \times 10^6$ takes a factor of 11 longer than with $N = 10^6$, instead of the expected factor of 8. The reason for the somewhat worse scaling of the run-time on the CPU is not yet clear and remains to be investigated in the future.

Note that as the memory transfer between the CPU and GPU is currently not optimized, our speedup calculations do not include that overhead. However, as we transfer only a subset of the entire data for each star, there is the potential space for improvement to interleave kernel computations with data transfer and substantially reduce this overhead.

Finally, we looked at the influence of GPU specific parameters on the run-time. In order to improve performance and utilization of the 30 multi-processors, we partitioned our data space into a one-dimensional grid of blocks on the GPU. Due to the complexity of the expressions involved in the calculations of orbital positions, our kernels use a significant amount of registers (64 registers per thread). Thus, the block dimension is restricted to 256 threads per block as the GTX280 GPU has only 16384 registers per block. To analyze the performance, we first made a parameter scan in the block and grid dimensions by varying the block sizes from 64 to 256 and the grid sizes from 12 to 72. Figure 12 shows the total run-time of all kernels as a function of the total number of active threads on the GPU. As expected, the runtime decreases with increasing thread number but saturates at around 6000 threads. The saturation is most likely due to the finite memory bandwidth, as we noted before that the run-time of the kernels is mainly determined by the number of memory accesses as opposed to the number of floating point operations. One can also see that the curve shows little scatter, $\approx 0.1\,\mathrm{s}$, which means that the specific size of a single block of threads has a minor effect on performance. We furthermore find that for a given total number of threads, the run-time is shortest when the total number of thread blocks is a multiple of the number of multi-processors, in our case 30.

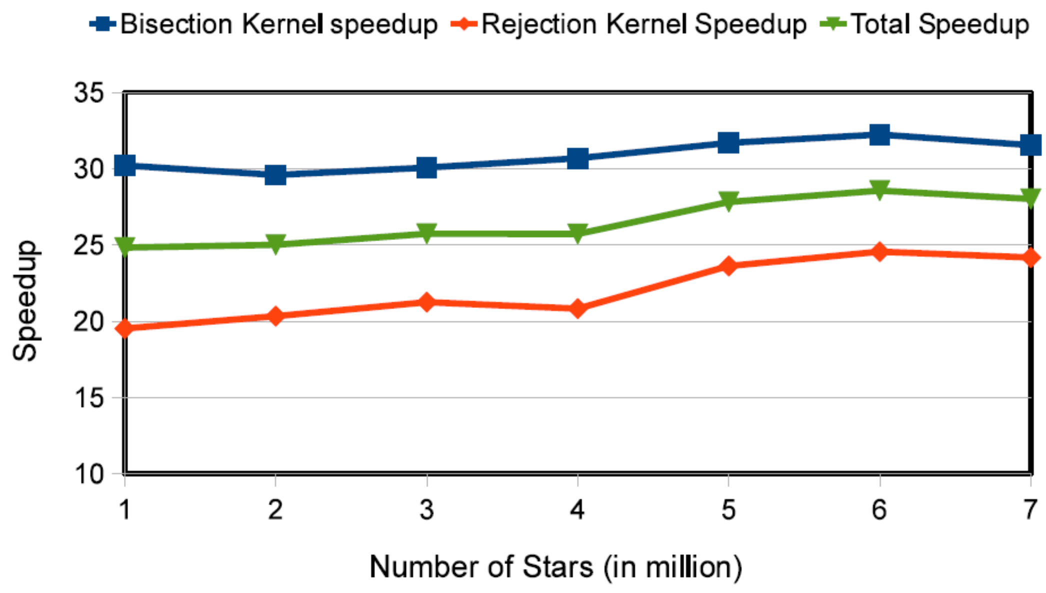

Fig. 11.— Total and individual speedups of the bisection and rejection kernels.

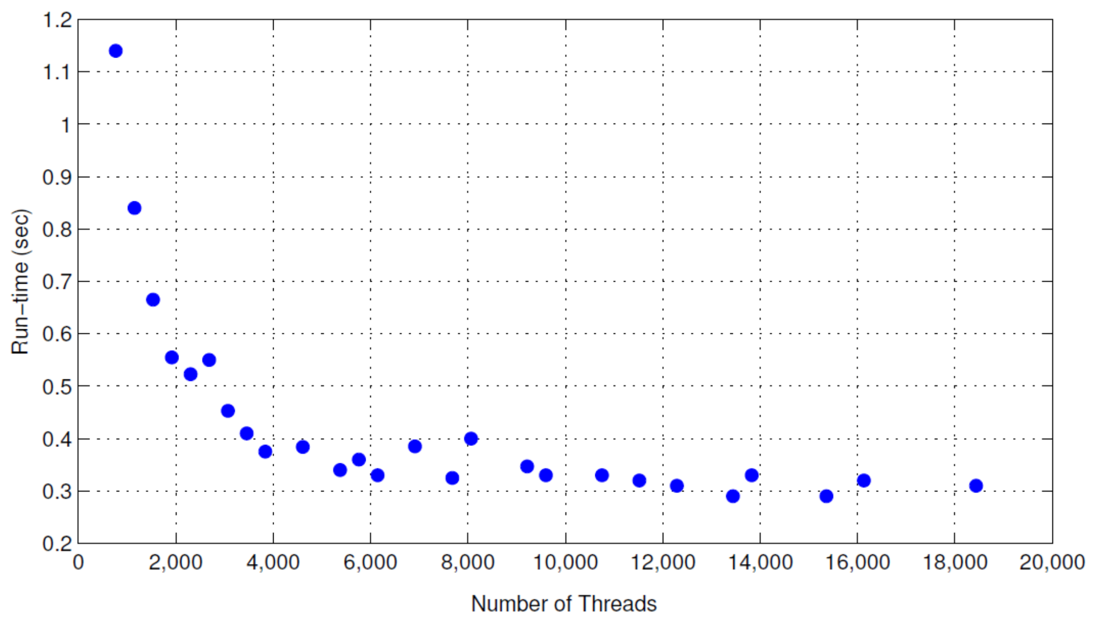

Fig. 12.— Total run-time of all kernels over the total number of threads.

REFERENCES

Binney, J., & Tremaine, S. 2008, Galactic Dynamics: Second Edition (Princeton University Press)

Chatterjee, S., Fregeau, J. M., Umbreit, S., & Rasio, F. A. 2010, ApJ, 719, 915

Cohn, H. 1980, ApJ, 242, 765

Collins, J. C. 2008, Army Research Laboratory, 4, 41

Fregeau, J. M., Cheung, P., Portegies Zwart, S. F., & Rasio, F. A. 2004, MNRAS, 352, 1

Fregeau, J. M., Gürkan, M. A., Joshi, K. J., & Rasio, F. A. 2003, ApJ, 593, 772

Fregeau, J. M., & Rasio, F. A. 2007, ApJ, 658, 1047

Freitag, M., & Benz, W. 2001, A&A, 375, 711

Goswami, S., Umbreit, S., Bierbaum, M., & Rasio, F. A. 2011, ArXiv e-prints

Heggie, D., & Hut, P. 2003, The Gravitational Million-Body Problem: A Multidisciplinary Approach to Star Cluster Dynamics, ed. Heggie, D. & Hut, P.

Heggie, D. C., & Stevenson, D. 1988, MNRAS, 230, 223

Hénon, M. H. 1971, Ap&SS, 14, 151

Hoare, C. A. R. 1961, Commun. ACM, 4, 321

Hurley, J. R., Pols, O. R., & Tout, C. A. 2000, MNRAS, 315, 543

Hurley, J. R., Tout, C. A., & Pols, O. R. 2002, MNRAS, 329, 897

Hut, P., Makino, J., & McMillan, S. 1988, Nature, 336, 31

Jalali, B., Baumgardt, H., Kissler-Patig, M., et al. 2012, A&A, 538, A19

Joshi, K. J., Nave, C. P., & Rasio, F. A. 2001, ApJ, 550, 691

Joshi, K. J., Rasio, F. A., & Portegies Zwart, S. 2000, ApJ, 540, 969

L'Ecuyer, P. 1999, Math. Comput., 68, 261

Li, X., Lu, P., Schaeffer, J., et al. 1993, Parallel Computing, 19, 1079

Lightman, A. P., & Shapiro, S. L. 1978, Reviews of Modern Physics, 50, 437

McLaughlin, D. E., & van der Marel, R. P. 2005, ApJS, 161, 304

Miller, M. C., & Lauburg, V. M. 2009, ApJ, 692, 917

Nvidia. 2010, http://developer.download.nvidia.com, 1

Takahashi, K. 1995, PASJ, 47, 561

Umbreit, S., Fregeau, J. M., Chatterjee, S., & Rasio, F. A. 2012, ApJ, 750, 31

Zonoozi, A. H., Küpper, A. H. W., Baumgardt, H., et al. 2011, MNRAS, 411, 1989

This manuscript was prepared with the AAS LaTeX macros v5.2.