Evaluating the Performance and Energy Efficiency of N-Body Codes on Multi-Core CPUs and GPUs

Ivan Zecena[1], Martin Burtscher[1], Tongdan Jin[2], Ziliang Zong[1]
[1] Department of Computer Science, Texas State University
[2] Ingram School of Engineering, Texas State University
{iz1003, burtscher, tj17, zz11}@txstate.edu

***Abstract*—N-body simulations are computation-intensive applications that calculate the motion of a large number of bodies under pair-wise forces. Although different versions of N-body code have been widely used in many scientific fields, the performance and energy efficiency of various N-body codes have not been comprehensively studied, especially when they are running on newly released multi-core CPUs and GPUs (e.g. Tesla K20). In this paper, we evaluate the performance and energy efficiency of five parallel *n*-body implementations on two different multi-core CPU systems and on two different types of GPUs. Our experimental results show that up to 71% of the energy can be saved by using all cores of a Xeon E5620 CPU instead of only one. We find hyper-threading to be able to further reduce the energy usage and runtime, but not as much as by adding more cores does. Finally, our experiments illustrate that GPU-based acceleration using a Tesla K20c can boost the performance and energy efficiency by orders of magnitude.**

Keywords—performance; energy efficiency; multi-core CPUs; GPUs; hyper-threading; n-body simulation

I. INTRODUCTION

Ever since Newton formulated his theory of gravitation to describe the motion of planets and stars (*i.e.*, bodies) under mutual forces, *n*-body problems have attracted significant interest. In recent decades, researchers have started employing *n*-body simulations in a number of domains outside of astronomy, including for studying elementary particles that induce electric and magnetic forces upon each other.

The *n*-body problem is simple in principle. Given the initial state (mass/charge, position and velocity) of *n* bodies at time *T*, we want to calculate the state of these bodies at a subsequent time *T'*. This is usually done incrementally by computing the evolution of the system in small time steps. For many real-world problems, the number of bodies, *n*, is very large (millions or billions). Hence, direct pair-wise *n*-body simulation may not be feasible due to its $O(n^2)$ algorithmic complexity. To make large problem sizes computationally tractable, several approximate *n*-body algorithms have been proposed, including the Barnes-Hut algorithm [2] and the Fast Multipole Method (FMM) [6]. In this paper, we study two implementations of the direct $O(n^2)$ approach, which we call NB, as well as three implementations of the $O(n \log n)$ Barnes-Hut algorithm, which we call BH. We chose the NB and BH methods because they represent interesting extremes. Of all the *n*-body algorithms we are aware of, NB is the most compute bound and BH is the most memory bound. Other fast algorithms, such as FMM, share similarities with BH but are less memory bound.

Most of the existing work related to *n*-body simulation focuses on reducing the time complexity of the algorithm or on parallelizing it for a specific hardware. However, a comprehensive study on the performance and energy efficiency of various *n*-body codes is missing, which is critical for N-body simulation users to choose the right version of N-body code based on different hardware (e.g. multi-core CPUs and GPUs) available to them. In this paper, we evaluate the performance and energy consumption of five parallel implementations of NB and BH. For BH, we analyze the P-Threads version written by Nicholas Chen at UIUC [12], the OpenMP version developed by Ricardo Alves at the University of Minho in Portugal, and the CUDA version from the LonestarGPU benchmark suite [9]. For NB, we study an OpenMP and a CUDA version we programmed. Our CUDA code outperforms the corresponding *n*-body implementation that ships with the CUDA 5.0 SDK [5] and reaches over two teraflops on some inputs on a K20c GPU. Section III provides more detail about these implementations. Hereafter, we refer to the five codes as BH_{PThr}, BH_{OMP}, BH_{CUDA}, NB_{OMP}, and NB_{CUDA}.

We first evaluate the impact of the thread count on the performance and energy efficiency of the multi-core CPU codes (BH_{PThr}, BH_{OMP}, and NB_{OMP}). We perform this analysis on two systems: one based on an Intel CPU (System 1) and the other based on two AMD CPUs (System 2). Our experiments show that up to 71% of the energy can be saved when all CPU cores are utilized. On the system that supports hyper-threading (System 1), we also evaluate the impact of hyper-threading on performance and energy consumption. Our results show that hyper-threading can improve the performance of BH_{Pthr} and BH_{OMP} by up to 30%, which yields up to 21% energy savings when simulating one million bodies. However, the impact of hyper-threading on the energy efficiency of NB_{OMP} is negligible. Finally, we evaluate the benefit of using GPU acceleration on performance and energy efficiency of NB and BH. Our experiments show that the GPU codes outperform the CPU versions by orders of magnitude. For example, when simulating one million bodies, the BH_{CUDA} code running on a K20c GPU is 45 times faster and 97.6% more energy efficient than the BH_{OMP} code running on a Xeon E5620 CPU. When simulating one million bodies, the NB_{CUDA} code runs 424 times faster on the GPU and only consumes 0.27% of the energy compared to the NB_{OMP} code running on the multi-core CPU. In addition, we find our Kepler-based Tesla K20c GPU to outperform our Fermi-based GeForce GTX 480 GPU in both performance and energy efficiency on all tested programs and inputs.

The remainder of this paper is organized as follows. Section II discusses related work. Section III presents the CPU and GPU *n*-body implementations. Section IV describes our systems and experimental methodology. Section V discusses and analyzes the results. Section VI concludes our study.

II. RELATED WORK

The *n*-body simulation problem has been studied extensively, and a variety of algorithms for it have been proposed. In particular, many previous studies focus on developing fast algorithms to break the O(n^2) complexity boundary.

In 1986, Barnes and Hut proposed the now well-known Barnes-Hut algorithm, which lowers the time complexity to O(*n* log *n*) using approximation [2]. Based on the general principle of this algorithm, researchers have developed many variants to speed up the execution. Salmon implemented a parallel version for distributed-memory machines [13]. Later, Warren and Salmon proposed improved parallel implementations [15], [16]. In 1997, Warren *et al.* exceeded one gigaflop when running their Barnes-Hut code on 16 Intel Pentium Pro processors [14]. Liu and Bhatt developed an algorithm that is based on a dynamic global tree that spans multiple processors [8]. Burtscher and Pingali wrote the first Barnes-Hut implementation that runs the entire algorithm on a GPU [4]. Bedorf *et al.* present a similar GPU implementation [3].

Several algorithms with a linear time complexity have also been proposed. The first was devised by Appel [1] in 1985. Greengard and Rokhlin developed the O(*n*) Fast Multipole Method [6]. Xue proposed another linear-time hierarchical tree algorithm for *n*-body simulations [18].

Very little work has been published on studying the energy efficiency of different *n*-body algorithms. The closest work we can find was published by Malkowski *et al.* [10]. They explore how to use low-power modes of the CPU and caches, and hardware optimization such as a load-miss predictor and data prefetchers, to conserve energy without hurting performance.

III. IMPLEMENTATION DESCRIPTION

We evaluate five different *n*-body codes belonging to two different categories. The NB implementations have O(n^2) complexity and the BH implementations have O(*n* log *n*) complexity. The goal of all five programs is to simulate the time evolution of a star cluster under gravitational forces for a given number of time steps.

A. BH Algorithm

The Barnes-Hut algorithm approximates the forces acting on each body. It hierarchically partitions the volume around the *n* bodies into successively smaller cells and records this spatial hierarchy in an unbalanced octree. Each cell forms an internal node of the octree and summarizes information about all the bodies it contains. The leafs of the octree are the individual bodies. This spatial hierarchy reduces the time complexity to O(*n* log *n*) because, for cells that are sufficiently far away, the algorithm only performs one force calculation with the cell instead of performing one force calculation with each body inside the cell, thus drastically reducing the amount of computation. However, differing parts of the octree have to be traversed to compute the force on each body, making the code's control flow and memory-access patterns quite irregular.

The P-Threads, OpenMP and CUDA codes we study perform six key operations in each time step to implement the BH algorithm. The first is an O(*n*) reduction to find the minimum and maximum coordinates of all bodies. The second operation builds the octree by hierarchically dividing the space containing the bodies into ever smaller cells in O(*n* log *n*) time until there is at most one body per cell. The third operation summarizes information in all subtrees in O(*n*) time. The fourth operation approximately sorts the bodies by spatial distance in O(*n*) time to improve the performance of the next operation. The fifth operation computes the force on each body in O(*n* log *n*) time by performing prefix traversals on the octree. This is by far the most time consuming operation in the BH algorithm. The final operation updates each body's position and velocity based on the computed force in O(*n*) time. Note that the P-Threads code only parallelizes the force calculation. The OpenMP and CUDA codes parallelize all six operations. In case of OpenMP, "parallel for" and "parallel" pragmas are used, in some cases in combination with gcc-specific synchronization primitives, memory fences, and atomic operations to handle data dependencies. The force calculation code uses a block-cyclic schedule whereas the other operations use the default schedule. The CUDA code incorporates many GPU-specific optimizations that are described elsewhere [4].

B. NB Algorithm

The direct NB algorithm performs precise pair-wise force calculations. For *n* bodies, O(n^2) pairs need to be considered, making the calculation quadratic in the number of bodies. However, identical computations have to be performed for all *n* bodies, leading to a very regular implementation that maps well to GPUs. As with BH, all force calculations are independent, resulting in a large amount of parallelism.

Both the OpenMP and the CUDA implementations perform two key operations per time step. The first is the O(n^2) force calculation and the second is an O(*n*) integration where each body's position and velocity are updated based on the computed force. The OpenMP code uses a "parallel for" pragma to parallelize the outer loop of the force calculation and the loop of the integration. The default schedule is used in both cases. The CUDA code is very similar in structure and uses GPU threads to completely eliminate these two loops, *i.e.*, each thread handles a different iteration. In addition, the force calculation code employs data tiling in shared memory (a software controlled L1 data cache) and unrolls the inner loop.

In summary, the NB codes are relatively straightforward, have a high computational density, and only access main memory infrequently because of excellent caching. In contrast, the BH codes are quite complex (they repeatedly build unbalanced octrees and perform various traversals on them), have a low computational density, and perform mostly pointer-chasing memory accesses. Due to the lower time complexity, the BH_{CUDA} implementation is about 33 times faster on a K20c than the NB_{CUDA} code when simulating one million stars.

IV. EVALUATION METHODOLOGY

A. Systems, Compilers, and Inputs

We conducted our experiments on two machines. System 1 is based on a 2.4 GHz Xeon E5620 CPU with four cores running 32-bit CentOS 5.9. It contains two GPUs. The first GPU is a previous generation Fermi-based GeForce GTX 480 with 1.5

GB of global memory and 15 streaming multiprocessors (SMs) with 480 processing elements (PEs) running at 1.4 GHz. The second GPU is a current generation Kepler-based Tesla K20c with 5 GB of global memory and 13 SMXs with 2,496 PEs running at 0.7 GHz. The idle power of the GTX 480 is 54 W, the K20c draws 13 W when idling, and the entire system consumes 165 W in idle mode. System 2 contains two quad-core AMD Opteron 2380 CPUs running at 2.5 GHz. It has no GPUs. The idle power draw of System 2 is 134.5 W.

We compiled the CUDA codes on System 1 with nvcc 5.0 using the -O3, -ftz=true, and -arch=sm_20 or -arch=sm_35 flags. For the P-Threads and OpenMP codes, we used gcc 4.6.3 on System 1 and gcc 4.4.6 on System 2 with the -O3 flag on both systems. We ran the BH programs with 250,000, 500,000 and one million stars and the NB programs with 25,000, 50,000 and 100,000 stars. We used ten time steps for all experiments. The stars' positions and velocities are initialized according to the empirical Plummer model [11], which mimics the density distribution of globular clusters.

B. Energy Profiling

We measure the system-wide energy consumption when running the application codes using a WattsUp power meter [17]. The meter's software runs in the background as a daemon and samples the voltage and current to compute the power. It has a 1 Hz sampling frequency and a power resolution of 0.1 W. The energy measurements are derived from the power readings by integrating the power over the runtime of the codes.

To improve the accuracy of our measurements, we report the median of five runs for each experiment. Additionally, we removed high-energy outliers in the data caused by operating system jitter [7] (a.k.a operating system noise or operating system interference) and external user activities such as logging into the system, which are unavoidable in multi-user systems. Finally, we started our measurements with a four-second head delay before launching the program. This removes the overhead caused by starting the meter and our applications at the same time and allows the system to settle before the application program is launched. We then removed the first four seconds of power readings from our final calculations.

V. EXPERIMENTAL EVALUATION

This section presents our experiments and analyzes the results. Subsection A discusses the impact of increasing the number of threads on performance and energy efficiency of the BH and NB codes. Subsection B studies the impact of hyper-threading on performance and energy consumption. Subsection C evaluates the GPU results and compares them to the CPU results.

A. Impact of Thread Count

BH Algorithm Table I shows the runtime and the energy consumption of BH_{OMP} and BH_{PThr} on System 1 for one, four, and eight threads with 250,000, 500,000 and one million bodies over ten time steps. For all three input sizes and both implementations, we observe that the performance increases almost linearly (96% parallel efficiency) when going from one to four threads but sublinearly (64% parallel efficiency) when going from four to eight threads, *i.e.*, when going from one to two threads per CPU core using hyper-threading. We defer the discussion of the hyper-threading results to Subsection B.

TABLE I. CPU BH RESULTS FOR 10 TIME STEPS ON THE HYPER-THREADED 4-CORE SYSTEM 1

# of Bodies	# of Threads	Algorithm	Runtime [s]	Energy [J]
250,000	1 Thread	OpenMP	183.0	30,776
		P-Threads	225.0	37,934
	4 Threads	OpenMP	47.5	9,035
		P-Threads	58.9	11,065
	8 Threads (HT)	OpenMP	38.3	7,539
		P-Threads	46.4	9,017
500,000	1 Thread	OpenMP	410.6	69,505
		P-Threads	497.6	83,643
	4 Threads	OpenMP	106.3	20,312
		P-Threads	130.1	24,811
	8 Threads (HT)	OpenMP	83.8	16,564
		P-Threads	101.5	19,818
1,000,000	1 Thread	OpenMP	902.7	152,462
		P-Threads	1,085.6	183,693
	4 Threads	OpenMP	232.7	44,723
		P-Threads	284.9	53,847
	8 Threads (HT)	OpenMP	180.7	35,937
		P-Threads	218.5	42,714

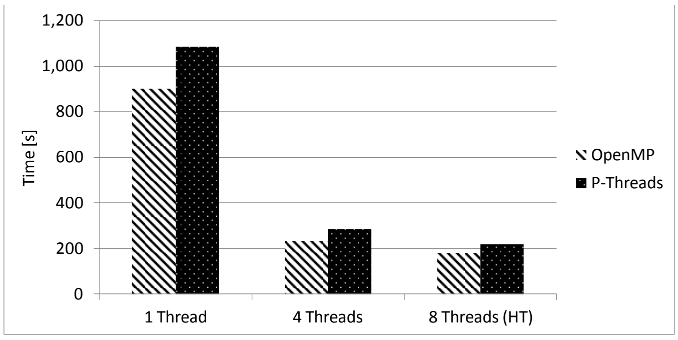

Fig. 1. Runtimes of BH_{OMP} and BH_{PThr} with 1 million bodies and 10 time steps on System 1

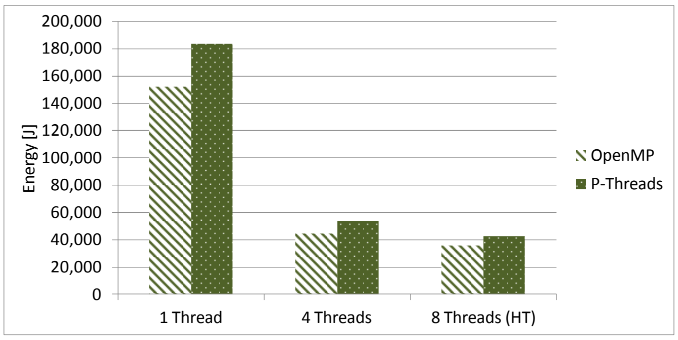

Fig. 2. Energy consumption of BH_{OMP} and BH_{PThr} with 1 million bodies and 10 time steps on System 1

Taking the 500,000-body experiments as an example, we observe that the OpenMP code's runtime is 410.6 seconds with one thread and 106.3 seconds with four threads, which amounts to a speedup of 3.86 when quadrupling the number of cores used. Analyzing the energy consumption, we find that using one thread consumes 69,505 joules whereas using four threads only consumes 20,312 joules to compute the same result, a 71% reduction in energy usage. The P-Threads implementation behaves similarly but the absolute numbers are worse. It runs for 497.6 seconds with one thread and for 130.1 seconds with four threads, which amounts to a speedup of 3.82. Energy-wise, we see a 70% savings when using four threads instead of one. The other two inputs exhibit nearly identical trends. For all three inputs sizes, the OpenMP version of BH consistently outperforms the P-Threads version on System 1 in both performance and energy efficiency. On average, the OpenMP code is 22.2% faster and consumes 21.5% less energy than the P-Threads code. Figures 1 and 2 visualize the runtime and energy consumption, respectively, for the 1,000,000-body runs.

The results for System 2 are provided in Table II. Figures 3 and 4 graphically depict the runtimes and energy consumption for one million bodies. Since this system has eight physical cores, we obtained results for one, eight, and 16 threads and 250,000, 500,000 and one million bodies. On the middle input, the OpenMP implementation achieves a 7.89-fold speedup (98.6% parallel efficiency) and an 86% energy savings when going from one to eight threads. The P-Threads code yields a speedup of 7.17 (89.6% parallel efficiency) and an 85% reduction in energy consumption when going from one to eight threads. The results for the other inputs are again very similar.

TABLE II. CPU BH RESULTS FOR 10 TIME STEPS ON THE 8-CORE SYSTEM 2

# of Bodies	# of Threads	Algorithm	Runtime [s]	Energy [J]
250,000	1 Thread	OpenMP	490.7	66,559
		P-Threads	528.4	71,743
	8 Threads	OpenMP	62.3	9,264
		P-Threads	75.1	11,025
	16 Threads	OpenMP	172.5	25,546
		P-Threads	81.9	11,979
500,000	1 Thread	OpenMP	1,039.6	141,165
		P-Threads	1,117.6	151,816
	8 Threads	OpenMP	131.8	19,498
		P-Threads	155.9	22,995
	16 Threads	OpenMP	344.6	51,134
		P-Threads	162.6	23,942
1,000,000	1 Thread	OpenMP	2,175.1	295,330
		P-Threads	2,325.7	316,155
	8 Threads	OpenMP	276.5	40,887
		P-Threads	321.2	47,221
	16 Threads	OpenMP	695.9	103,739
		P-Threads	324.1	47,598

Interestingly, the BH_{OMP} code scales substantially better on System 2 than the BH_{PThr} code. Whereas the OpenMP code is only 7.4% faster when using one thread, its advantage over the P-Threads code increases to 18.3% when using eight threads. The energy savings closely follow these percentages. In contrast, the two codes scale nearly identically on System 1.

Comparing the single-thread runs across the two systems, we find that System 1 is, on average, 2.54 faster on the OpenMP code and 2.25 times faster on the P-Threads code. However, the energy-efficiency ratios are lower. System 1 outperforms System 2 by a factor of 2.04 on the OpenMP code and by a factor of 1.81 on the P-Threads code on average. Overall, System 1 is much faster and also more energy efficient than System 2, but System 2 has a lower average power consumption of 135.8 watts (compared to 168.7 watts for System 1). Note that these numbers are only a little above the idle power draw for both systems, showing that the one computation thread consumes relatively little extra energy. For reference, when using four threads on System 1 and eight threads on System 2, the average power draw increases to 190.2 watts and 147.6 watts, respectively.

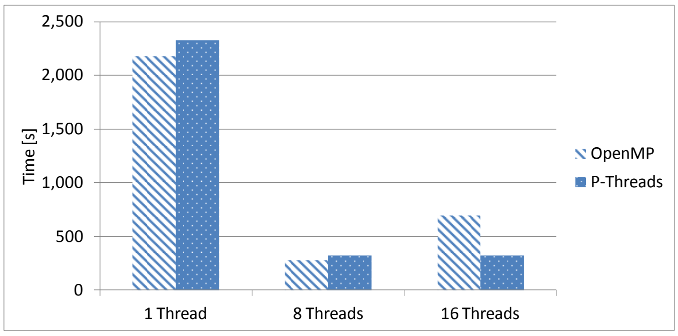

Fig. 3. Runtimes of BH_{OMP} and BH_{PThr} with 1 million bodies and 10 time steps on the 8-core System 2

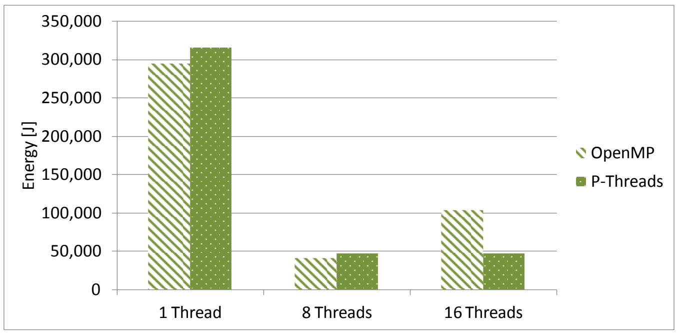

Fig. 4. Energy consumption of BH_{OMP} and BH_{PThr} with 1 million bodies and 10 time steps on the 8-core System 2

In summary, we find the OpenMP implementation to outperform the P-Threads implementation in performance and energy efficiency on both systems for all investigated inputs and thread counts. This is most likely because the P-Threads code only parallelizes the force calculation whereas the OpenMP code parallelizes all six algorithmic steps. Nevertheless, both BH codes scale well on our two multi-core CPU systems. Most importantly, we find on both systems that using all available cores results in large energy savings. The main reason for this is the high idle power. Since its contribution to the overall energy consumption is proportional to the runtime, reducing the runtime through parallelization saves energy. If the idle power were zero, using four cores to run the code, say, 3.86 times faster would actually increase the energy consumption relative to running the code on just a single core assuming all cores are independent (which they are not in current multi-core CPUs) and identical.

As real systems have a non-zero idle power, there is a minimum speedup that must be achieved by parallel code to be more energy efficient than serial code. For example, System 1 must execute the BH_{OMP} code at least 1.13 times faster using four cores to save energy over running the code on one core. The corresponding factor for System 2 is 1.10. Fortunately, a 10% or 13% speedup should be relatively easy to achieve on four or eight cores, meaning that parallelization is likely to be worthwhile to improve a program's energy efficiency on today's multi-core systems.

NB Algorithm Tables III and IV contain the experimental results for our CPU implementation of the NB algorithm running on Systems 1 and 2, respectively. They show how the OpenMP code performs for different input sizes and thread counts. As before, we use 1, 4, and 8 threads on System 1 and 1, 8, and 16 threads on System 2. The input sizes are 25,000, 50,000 and 100,000 bodies on both systems. These sizes are smaller because the NB code is slower than the BH code. The NB_{OMP} results for the largest input are further illustrated in Figures 5 and 6 for System 1 and in Figures 7 and 8 for System 2. We leave the hyper-threading discussion for Subsection B.

TABLE III. CPU NB_{OMP} RESULTS FOR 10 TIME STEPS ON THE HYPER-THREADED 4-CORE SYSTEM 1

# of Bodies	# of Threads	Runtime [s]	Energy [J]
25,000	1 Thread	154.3	25,971
	4 Threads	39.9	7,344
	8 Threads (HT)	39.7	7,388
50,000	1 Thread	616.3	103,670
	4 Threads	159.2	29,563
	8 Threads (HT)	158.7	29,996
100,000	1 Thread	2,465.5	414,248
	4 Threads	637.6	119,447
	8 Threads (HT)	634.3	120,725

TABLE IV. CPU NB_{OMP} RESULTS FOR 10 TIME STEPS ON THE 8-CORE SYSTEM 2

# of Bodies	# of Threads	Runtime [s]	Energy [J]
25,000	1 Thread	280.5	38,251
	8 Threads	35.2	5,157
	16 Threads	35.6	5,253
50,000	1 Thread	1,131.3	153,478
	8 Threads	141.6	20,809
	16 Threads	142.4	20,968
100,000	1 Thread	4,525.3	614,314
	8 Threads	567.9	83,875
	16 Threads	566.9	83,546

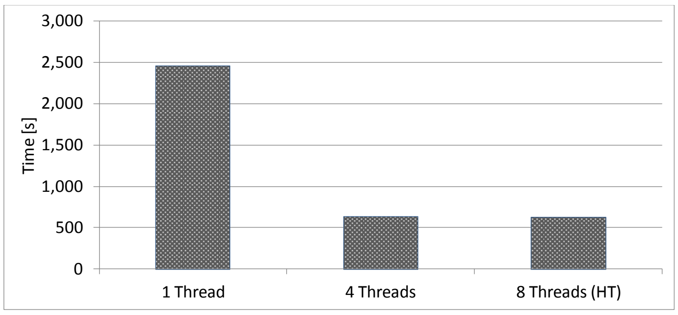

Fig. 5. Runtime of NB_{OMP} with 100,000 bodies and 10 timesteps on System 1

On System 1, the NB code scales nearly identically on all three inputs. Going from one to four threads, we obtain an average speedup of 3.87 (corresponding to 96.7% parallel efficiency) and a 71.5% savings in energy. On System 2, the behavior is also very similar between the three inputs. Increasing the number of threads from one to eight yields a speedup of 7.975 (99.7% parallel efficiency) and energy savings of 86.4%.

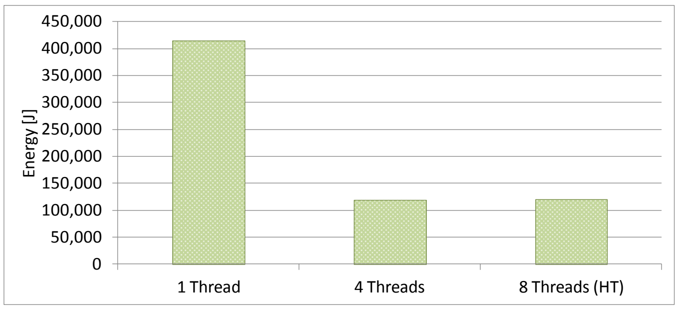

Fig. 6. Energy consumption of NB_{OMP} with 100,000 bodies and 10 time steps on System 1

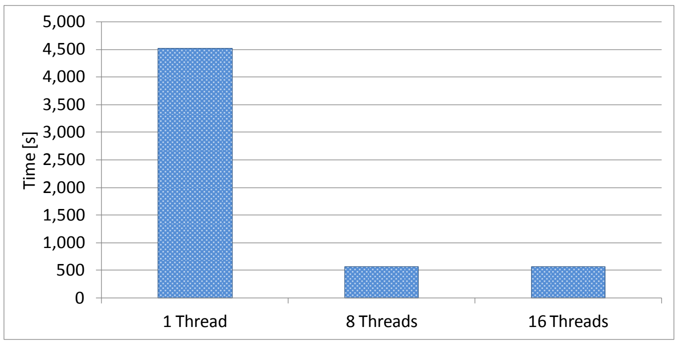

Fig. 7. Runtime of NB_{OMP} with 100,000 bodies and 10 timesteps on the 8-core System 2

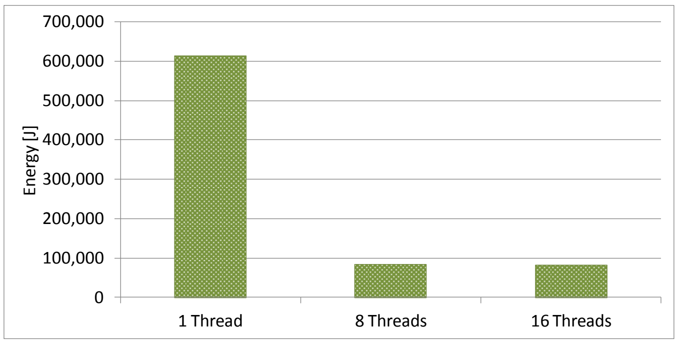

Fig. 8. Energy consumption of NB_{OMP} with 100,000 bodies and 10 time steps on the 8-core System 2

Comparing the single-threaded runs on both systems, we find that System 1 is 1.83 times faster and 1.48 times more energy efficient than System 2 when running NB_{OMP}. The minimum speedup to be energy efficient using four cores on System 1 is 10.5%. System 2 only requires a minimum speedup of 8.2% on eight cores to be more energy efficient.

In summary, the BH and NB CPU codes experience good scaling and boosts in energy efficiency on both tested systems. Clearly, parallelizing the BH and NB codes is a very profitable way to improve both performance and energy efficiency.

B. Impact of Hyper-Threading

In this subsection, we study the impact of hyper-threading (*i.e.*, running two threads per CPU core) on performance and energy efficiency of System 1. System 2 does not support hyper-threading.

BH Algorithm Table I and Figures 1 and 2 above illustrate the results for BH. Whereas hyper-threading clearly provides a benefit, the benefit is smaller than the benefit of using more physical cores. For instance, going from one to two threads per core (*i.e.*, increasing the thread count from 4 to 8) results in a parallel efficiency of about 64%. In contrast, going from one to four threads with one thread per core results in a parallel efficiency of about 96%. The results are nearly the same for both implementations and all inputs we tested. Note that the runtime as well as the energy consumption decrease when using hyper-threading. On average, hyper-threading yields additional energy savings of 18% on BH_{OMP} and 20% on BH_{PThr}.

On System 2, running twice as many threads as there are cores (*i.e.*, increasing the thread count from 8 to 16) hurts the performance and the energy efficiency, as the data in Table II and Figures 3 and 4 above show, because this system does not support hyper-threading. Interestingly, the degradation is small for BH_{PThr} but large for BH_{OMP}. We are unsure what causes this difference in behavior between the two programs when over-subscribing threads to cores in System 2. Running more threads than there are (non-hyper-threading) cores hurts the performance and increases the energy consumption on both codes and all inputs. Hence, we do not recommend it.

NB Algorithm Surprisingly, hyper-threading does not help with NB as shown in Figures 5 and 6. Whereas it does result in a tiny speedup, as Table III reveals, it actually raises the energy consumption. Hence, hyper-threading does not further improve the energy efficiency of the NB code on System 1.

The reason why hyper-threading improves the performance and the energy efficiency of the two BH implementations but not the NB implementation is the following. The BH codes are memory bound, meaning that they do not fully utilize the CPU cores because the memory system is the bottleneck. Hyper-threading enables each core to execute useful instructions from one thread whenever the other thread is stalled waiting for a memory request, thus boosting performance. In other words, hyper-threading helps hide some of the memory access latencies. In contrast, the NB code is compute bound, *i.e.*, the CPU cores are already fully utilized when running one thread per core. Hence, there is no benefit from hyper-threading, but it should be pointed out that it also does not hurt performance.

In summary, we conclude that adding extra cores is more useful than adding hyper-threading support from an energy-efficiency perspective. Yet, we find running two threads per core to save up to 21% energy (20% on average on BH_{PThr}). Moreover, the combination of both approaches results in the largest energy savings and should be used when available.

C. *Impact of GPU Acceleration*

In this subsection, we study the performance and the energy efficiency of our CUDA implementations of BH and NB. We report the runtime over the entire application, including CPU and GPU code, and the energy consumption of the entire system when executing the accelerated code segments on a GTX 480 or a K20c GPU, both of which reside in System 1. As before, we run the BH code with 250,000, 500,000, and one million bodies and the NB code with 25,000, 50,000 and 100,000 bodies. Table V presents the results for the GTX 480 and Table VI for the K20c. Figures 9 to 12 depict the same results graphically.

TABLE V. GTX 480 GPU BH_{CUDA} AND NB_{CUDA} RESULTS FOR 10 TIME STEPS

Algorithm	# of Bodies	Runtime [s]	Energy [J]
BH	250,000	1.3	376
	500,000	2.4	663
	1,000,000	4.8	1,223
NB	25,000	0.4	249
	50,000	1.1	371
	100,000	3.5	906

TABLE VI. K20C GPU BH_{CUDA} AND NB_{CUDA} RESULTS FOR 10 TIME STEPS

Algorithm	# of Bodies	Runtime [s]	Energy [J]
BH	250,000	1.3	330
	500,000	2.2	547
	1,000,000	4.0	865
NB	25,000	0.5	196
	50,000	0.7	198
	100,000	1.5	329

Comparing the runtimes between the two GPUs in Figures 9 and 10, we find that the K20c outperforms the GTX 480 by 1.6% to 20.4% on BH and by up to a factor of 2.35 on NB. However, on the smallest NB input, the GTX 480 is 13.6% faster. Nevertheless, the K20c is typically faster as it is based on the current-generation Kepler architecture whereas the GTX 480 is based on the previous-generation Fermi architecture.

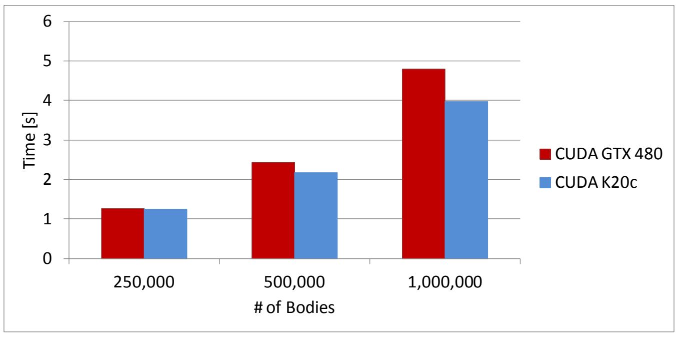

Fig. 9. Runtime of BH_{CUDA} with 10 timesteps

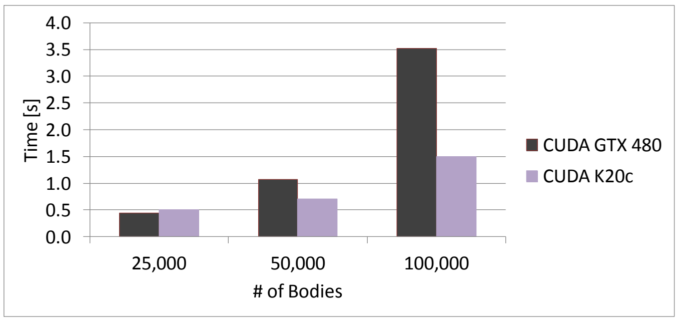

Fig. 10. Runtime of NB_{CUDA} with 10 timesteps

This difference in GPU generations is also the reason for the better energy efficiency of the K20c, which was a primary design goal for the Kepler. Figures 11 and 12 show the energy consumption of BH_{CUDA} and NB_{CUDA}, respectively. The K20c is more energy efficient than the GTX 480 on all three inputs and both codes. On BH, the K20c saves between 12% and 29% energy. On NB, it saves between 21% and 64% energy.

In summary, the K20c generally but not always outperforms the GTX 480. The difference in runtime is more pronounced on compute-bound code like NB. More importantly, the K20c is substantially more energy efficient than the GTX 480, in particular on compute-bound code. Note that the higher performance of the K20c contributes to but is not the primary reason for its better energy efficiency. Rather, the K20c is based on a substantially lower power GPU design.

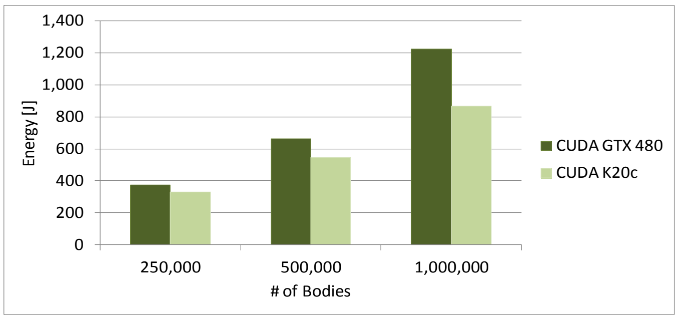

Fig. 11. Energy consumption of BH_{CUDA} with 10 timesteps

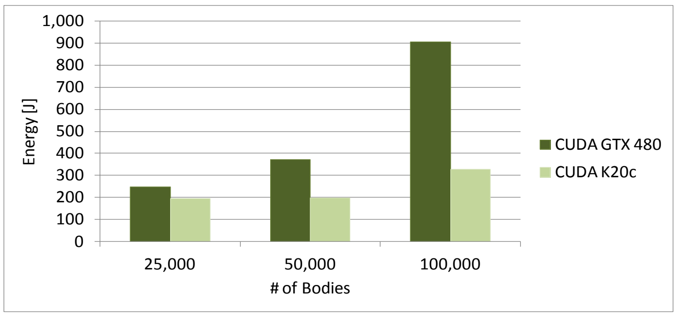

Fig. 12. Energy consumption of NB_{CUDA} with 10 timesteps

To evaluate the benefit of using GPUs, we compare the GPU results to the results of the best-performing CPU code, which is the OpenMP version of BH and NB running on System 1 with hyper-threading. Figures 13 and 14 show the BH comparisons and Figures 15 and 16 show the NB comparisons.

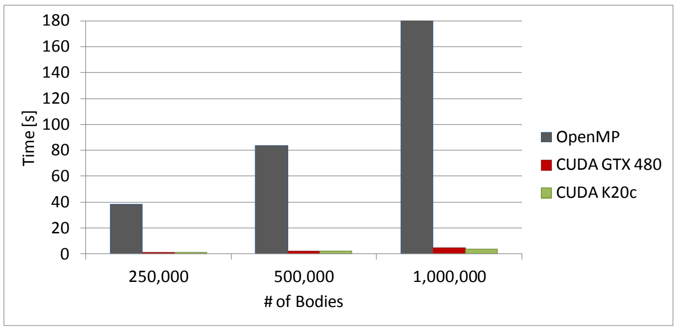

Fig. 13. Runtime comparison between BH_{OMP} and BH_{CUDA} with 10 timesteps

The GPU codes are tremendously faster and much more energy efficient than the multi-core CPU codes on all inputs. Comparing BH_{OMP} running on System 1 to BH_{CUDA} running on the K20c, we find the GPU code to be 31 times as fast for 250,000 bodies, 39 times as fast for 500,000 bodies, and 45 times as fast for 1,000,000 bodies. Moreover, the GPU code consumes only 4.4% of the energy for 250,000 bodies, 3.3% for 500,000 bodies, and 2.4% for 1,000,000 bodies. This amounts to almost two orders of magnitude in energy savings. Clearly, GPUs are not only great at accelerating code but also nearly as effective at saving energy.

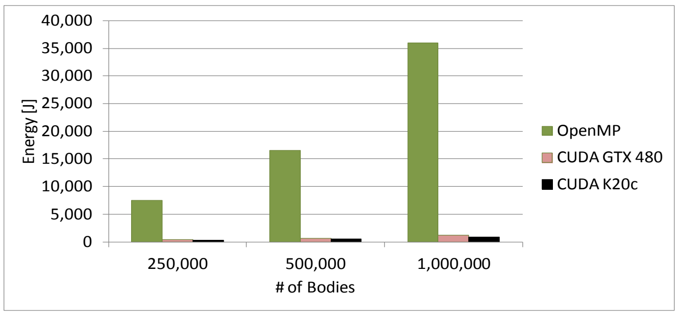

Fig. 14. Energy consumption comparison between BH_{OMP} and BH_{CUDA} with 10 timesteps

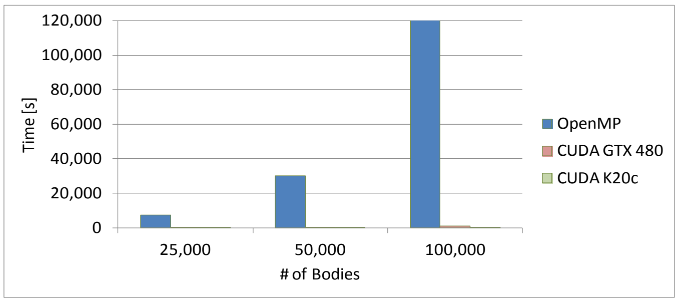

Fig. 15. Runtime comparison between NB_{OMP} and NB_{CUDA} with 10 timesteps

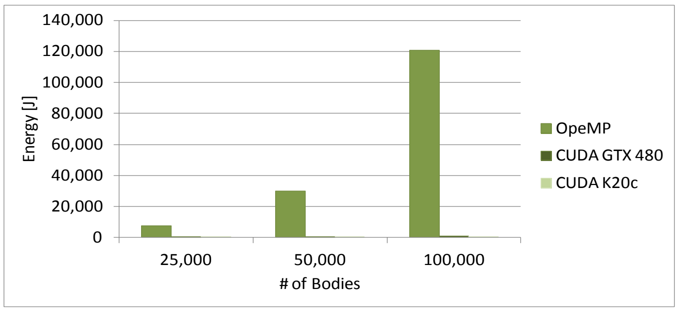

Fig. 16. Energy consumption comparison between NB_{OMP} and NB_{CUDA} with 10 timesteps

The GPU outperforms the CPU by even larger factors on NB. However, this is likely in part due to poor code generation. In particular, the gcc compiler does not exploit vector instructions for this code. Hence, we find the NB_{CUDA} code running on the K20c to be 79 times as fast as the NB_{OMP} code running on System 1 for 25,000 bodies, 225 times as fast for 50,000 bodies, and 424 times as fast for 100,000 bodies. The GPU code consumes only 2.7% of the energy for 25,000 bodies, 0.66% for 50,000 bodies, and 0.27% for 100,000 bodies. This amounts to close to three orders of magnitude in energy savings.

In summary, our results indicate that GPUs can improve the performance of *n*-body codes by one to three orders of magnitude compared to multi-core CPUs. Their energy consumption is lower by almost the same factor, making GPU acceleration also a very promising and effective approach for improving the energy efficiency of *n*-body codes.

VI. CONCLUSION AND FUTURE WORK

This paper studies the energy consumption and performance of five *n*-body codes running on two systems and two GPUs. The first system is based on a 2.4 GHz quad-core Intel Xeon E5620 CPU that supports 2-way hyper-threading. The second system contains two 2.5 GHz quad-core AMD Opteron 2380 CPUs. The two NVIDIA GPUs are a 1.4 GHz GeForce GTX 480 with 480 processing elements and a 0.7 GHz Tesla K20c with 2496 processing elements. Two of the five *n*-body codes we study implement an $O(n^2)$ algorithm and the remaining three programs implement the $O(n \log n)$ Barnes-Hut algorithm. The former category includes an OpenMP and a CUDA version whereas the latter category includes a P-Threads, an OpenMP and a CUDA version. Each code is tested on 3 inputs.

The studied codes scale well on both systems with different multi-core CPUs, including the complex Barnes-Hut implementation. However, there are differences in the scaling of the programs. Unexpectedly, we found some *n*-body codes to scale better on one system than on the other while other codes scale nearly equally on both systems. Hardly surprisingly, the OpenMP implementation that parallelizes every step of the algorithm is faster than the P-Threads implementation that only parallelizes one step.

In general, the CPU power draw is relatively small compared to the system idle power, which makes the energy consumption of a program highly dependent on the runtime. As a consequence, any reduction in runtime results in nearly proportional savings in energy. This is why program parallelization is so important to achieving high energy efficiency. It also explains why our system with the lower power draw is less efficient as it is much slower. We conclude that shortening a program's runtime is paramount to improve its energy efficiency. Hence, it is crucial to parallelize the entire application and to utilize all cores. After all, we found very small speedups due to parallelization to be sufficient to improve the energy efficiency.

Once all cores are used, running multiple threads per core on systems that support hyper-threading can result in significant additional improvements. Interestingly, we found hyper-threading to only help on the memory-bound code we studied but not on the compute-bound code. Nevertheless, since it does not appear to hurt, we recommend using hyper-threading when available. However, running too many threads can be detrimental to performance and energy efficiency. Thus, care must be taken to avoid oversubscribing threads to cores.

Even though the power drawn by the GPUs is quite high (on the order of the system's idle power) and much higher than that of the CPUs, the GPUs are so much faster that they turn out to be very energy efficient. In fact, the GPU-accelerated *n*-body implementations we investigated consume two to three orders of magnitude less energy than the multi-core CPU codes. Whereas the current-generation K20c GPU typically but not always outperforms the previous-generation GTX 480 GPU, the K20c appears to generally consume less energy, in particular on compute-bound code. Overall, we found GPUs to be great at speeding up our programs and nearly as effective at saving energy, making GPU acceleration our number one recommendation for improving energy efficiency.

In future work, we plan to expand our test bed to encompass a wider variety of systems and accelerators, including the new Intel Xeon Phi. On the software side, we want to extend our work to distributed-memory implementations as well as other commonly used *n*-body algorithms such as FMM. Finally, we hope to broaden our investigation to important problem domains such as sorting and FFTs.

ACKNOWLEDGMENTS

The work reported in this paper is supported by the U.S. National Science Foundation under Grants No. CNS-1118043, DUE-1141022, CNS-1217231, the U.S. Department of Agriculture under Grant No. 2011-38422-30803, a grant from the Texas State University Research Enhancement Program as well as grants and equipment donations from NVIDIA Corporation.

REFERENCES

[1] A. Appel, "An Efficient Program for Many-Body Simulation", SIAM J. Scientific and Statistical Computing, vol. 6, 1985.

[2] J. Barnes and P. Hut, "A Hierarchical O(N log N) Force-Calculation Algorithm", Nature, vol. 324, 1986.

[3] J. Bedorf, E. Gaburov, and S.P. Zwart, "A Sparse Octree Gravitational n-Body Code that Runs Entirely on the GPU Processor", J. Comput. Phys., 231(7):2825–2839, 2012.

[4] M. Burtscher and K. Pingali, "An Efficient CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm", Chapter 6 in GPU Computing Gems Emerald Edition, pp. 75-92, 2011.

[5] CUDA SDK: https://developer.nvidia.com/cuda-toolkit

[6] L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations", J. of Comp. Physics, 73:325–348, 1987.

[7] T. Jones, A. Tauferner, and T. Inglett, "Linux OS Jitter Measurements at Large Node Counts Using a Blue Gene/L", Technical Report, Oak Ridge National Laboratory, ORNL/TM-2009/303, 2009.

[8] P.F. Liu and S.N. Bhatt, "Experiences with Parallel N-Body Simulation", IEEE Trans. on Parallel and Distributed Sys., 11(11), 2000.

[9] LonestarGPU: http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu

[10] K. Malkowski, P. Raghavan, and M. Irwin, "Toward a Power Efficient Computer Architecture for Barnes-Hut N-body Simulations", Proc. of the 2006 ACM/IEEE Conference on Supercomputing, 2006.

[11] H. C. Plummer, "On the Problem of Distribution in Globular Star Clusters", Mon. Not. R. Astron. Soc., 71:460, 1911.

[12] P-Threads Barnes-Hut implementation with uniform data distribution: https://wiki.engr.illinois.edu/download/attachments/183271790/BarnesHut_PThreads_Uniform_Distribution_v1.0.zip

[13] J. Salmon, "Parallel Hierarchical N-Body Methods", PhD thesis, 1990.

[14] M. Warren, D. Becker, M. Goda, J. Salmon, and T. Stering, "Parallel Supercomputing with Commodity Components", Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications, 1997.

[15] M. Warren and J. Salmon, "Astrophysical N-Body Simulations Using Hierarchical Tree Data Structures", Proc. Supercomputing, 1992.

[16] M. Warren and J. Salmon, "A Parallel Hashed Oct-Tree N-Body Algorithm", Proc. Supercomputing, 1993.

[17] WattsUp: https://www.wattsupmeters.com/

[18] G. Xue, "An O(n) Time Hierarchical Tree Algorithm for Computing Force Field in n-Body Simulations", Theoretical Computer Science, vol. 197(12):157–169, 1998.