LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-609973

Scalable Load Balancing for Massively Parallel Distributed Monte Carlo Particle Transport

M. J. O'Brien, P. S. Brantley, K. I. Joy

January 14, 2013

International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering
Sun Valley, ID, United States
May 5, 2013 through May 9, 2013

Disclaimer

International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013), Sun Valley, Idaho, USA, May 5-9, 2013, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2013)

SCALABLE LOAD BALANCING FOR MASSIVELY PARALLEL DISTRIBUTED MONTE CARLO PARTICLE TRANSPORT

Matthew J. O'Brien and Patrick S. Brantley
Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA 94550
mobrien@llnl.gov; brantley1@llnl.gov

Ken I. Joy
Institute for Data Analysis and Visualization
Computer Science Department
University of California
One Shields Avenue
Davis, CA 95616
joy@cs.ucdavis.edu

ABSTRACT

In order to run computer simulations efficiently on massively parallel computers with hundreds of thousands or millions of processors, care must be taken that the calculation is load balanced across the processors. Examining the workload of *every* processor leads to an unscalable algorithm, with run time at least as large as $\Omega(N)$, where N is the number of processors. We present a scalable load balancing algorithm, with run time $\Theta(\log(N))$, that involves iterated processor-pair-wise balancing steps, ultimately leading to a globally balanced workload. We demonstrate scalability of the algorithm up to 2 million processors on the Sequoia supercomputer at Lawrence Livermore National Laboratory.

Key Words: Scalability, load balancing, high performance computing, Monte Carlo particle transport.

1. INTRODUCTION

In order to run computer simulations efficiently on massively parallel computers with hundreds of thousands or millions of processors, care must be taken that the calculation is load balanced across the processors. We assume we have a distributed memory parallel supercomputer, using the *Message Passing Interface* (MPI) [1] for inter-process communication. The work described in this paper is aimed at developing a scalable load balancing technique for a massively parallel Monte Carlo particle transport code [2], where the particle workload is distributed across processors. The assumptions for this paper are that the computational cost of all the Monte Carlo particles is the same, and that any processor can process any particle. In our previous load balancing algorithm [3], [4], a description of each processor's workload was gathered to the 0^{th} ranked processor, where a *global* communication graph was constructed to achieve a load balanced state. This algorithm performed efficiently up to thousands of processors, but for larger

processor counts, the load balancing step itself required longer than the computation portion of the calculation. As soon as one array of length proportional to the number of processors is required, the algorithm is already not scalable. We define an algorithm to be ***scalable*** if its run time is at most proportional to the logarithm of the number of processors. This definition rules out any global algorithm that needs to simultaneously know the workload of every processor.

We have developed and implemented a scalable load balancing algorithm in the *Mercury* Monte Carlo particle transport code [2], [5]. Mercury is written in C++ with a Python user interface and uses distributed memory parallelism with MPI. Mercury models dynamic neutron, gamma and light charged particle transport and also solves neutron criticality problems. The geometry information through which the particles are transported is stored redundantly on all of the processors and is not domain decomposed in this case. (Mercury has domain decomposition, but this paper only addresses particle replication). The number of Monte Carlo particles can be very large, and the particles are load balanced and distributed across processors.

The run time of our previous load balancing algorithm [3] was $\Theta(N^2)$ where N is the total number of processors. This algorithm performed efficiently up to several thousand processors, but on 2^{17} = 131,072 processors, the load balancing step itself took 90 times longer than the computation part of the calculation. The load balancing step should take only a small fraction of the computation part of the calculation. The load balancing algorithm was not initially written with scalability in mind, so we are now revisiting load balancing with a focus on scalability.

Romano and Forget [6] address a similar problem with a different set of constraints and assumptions. Their algorithm has the constraint that particles must be processed in a certain order, but in our case each particle has its own random number seed and may be processed on any processor in any order. This is a much less constrained problem and allows for an efficient, scalable, load balancing solution.

The remainder of this paper is organized as follows. In Sec. 2, we describe the need for and goals of load balancing for Monte Carlo calculations. We describe in Sec. 3 the scalable load balancing algorithm that we have developed and implemented in Mercury. In Sec. 4, we present numerical results from weak scaling studies that demonstrate the need for load balancing as well as the scalability of the load balancing algorithms we have developed. We conclude the paper in Sec. 5 and offer suggestions for future work.

2. LOAD BALANCING FOR MONTE CARLO CALCULATIONS

The *load balance efficiency* of a calculation is the average amount of computational work per processor divided by the maximum amount of computational work on any processor. Let w_0, w_1, …, w_{N-1} be the amount of computational work per processor, then the load balance efficiency is:

$$\text{Load Balance Efficiency} = \frac{ave(w_i)}{max(w_i)} = \frac{\frac{1}{N}\sum_{0\le i<N} w_i}{\max\limits_{0\le i<N} w_i} \quad (1)$$

The goal of load balancing is to maximize the load balance efficiency of a calculation. By moving work from one processor to another, we cannot change the *average* amount of work per processor, but we can change the *maximum* amount of work on any processor. The goal of load balancing then becomes trying to minimize the amount of work on the processor that has the most work, thereby maximizing the load balance efficiency.

At the start of each computational physics cycle, each processor starts with some number of particles. The goal of the load balancing problem is for each processor to have the *same* number of particles (or have the maximum difference of particle counts be at most one if the number of particles is not a multiple of the number of processors). The result of the load balancing algorithm is to have particles communicated between processors, so that after the communication the particle counts are balanced. Then the computational physics cycle occurs, which may induce new load imbalance, and the load balance step is repeated.

Running problems *without* load balancing results in the load balance efficiency decreasing as a function of generation in an eigenvalue calculation. The load balance efficiency also decreases as the number of processors increases. Load balancing the problem enforces that all processors have essentially the same number of particles, so the efficiency remains high.

3. SCALABLE PARTNER PROCESSOR ALGORITHM

An iterative load balancing algorithm was developed such that at each iteration, every processor finds a unique *partner* processor. The partner processors send and receive the number of particles they own to each other. Then both processors compute the average of these two numbers. The partner that is above the average sends particles to the partner that is under the average, so both processors end up having the average number of particles. If both processors have the same number of particles, then they are already load balanced and have nothing to do. This algorithm has pair-wise interactions between processors, never knowing what the global workload distribution looks like. We define the processor ***rank*** to be the unique processor number in *{0, 1, 2, …, N-1}*, when there are *N* total processors. After each iteration, the partner processors are *individually* balanced. By choosing the partner processor appropriately, on the k^{th} iteration, all processor ranks with the same binary representation up to the last *k* digits will have exactly the same number of particles, i.e. processors in groups of 2^k are balanced.

We now define how the processors are paired. Processors are paired based upon their processor rank and the current iteration number *k* of the algorithm. We choose the partner processor on the k^{th} iteration of the algorithm by defining the partner function f_k, and the rank of the partner processor is given by: *partner* = $f_k(rank)$, where

$$f_k(rank) = \begin{cases} rank + 2^k & \text{if the } k^{th} \text{ binary digit of } rank \text{ is 0} \\ rank - 2^k & \text{if the } k^{th} \text{ binary digit of } rank \text{ is 1} \end{cases}$$

(2)

Another interpretation of f_k is to flip the k^{th} binary digit of the input argument. If $a_n \ldots a_0$ are the binary digits of the processor rank, then

$$\text{rank} = a_n a_{n-1} \ldots a_{k+1} a_k a_{k-1} \ldots a_1 a_0 = \sum_{i=0}^{n} a_i 2^i \quad \text{where } a_i \in \{0,1\} \tag{3}$$

$$f_k(rank) = f_k(a_n a_{n-1} \ldots a_{k+1} a_k a_{k-1} \ldots a_1 a_0) = a_n a_{n-1} \ldots a_{k+1} \overline{a_k} a_{k-1} \ldots a_1 a_0 \tag{4}$$

$$where \ \overline{a_k} = 1 - a_k \tag{5}$$

f_k has the appealing property that it is self-inverting: $f_k(f_k(rank)) = rank$, i.e. f_k is an *involution*, so $f_k^{-1} = f_k$. As a result of this property,

$$f_k(rank) = partner \text{ and } f_k(partner) = rank. \tag{6}$$

Figure 1 shows a graph of the partner function f_k for k=0,1,2,3,4. Note the functions are symmetric about the identity line ("y=x" line) at integer values, which characterizes involutions (self inverse functions).

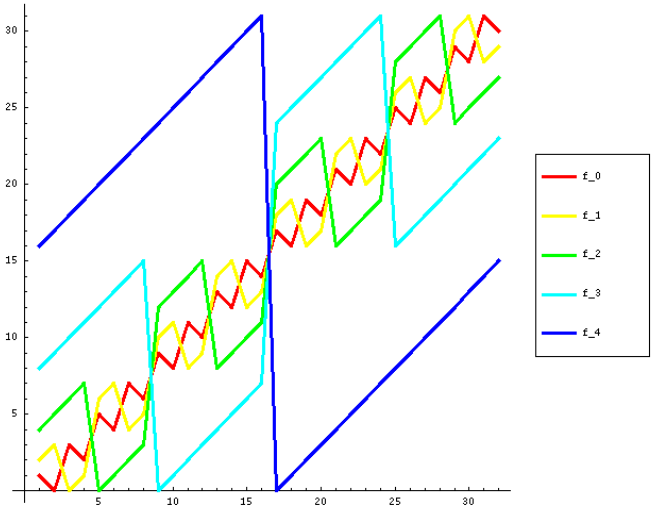

Figure 1: The partner function f_k for k=0,1,2,3,4.

Algorithm 1 shows pseudocode for the implementation of a scalable load balancing algorithm. If N = the number of processors, when $N=2^n$ is a power of 2, then **`BalanceWithPartnerWrapper()`** globally balances the number of particles per processor so that all processors have exactly the same number of particles at the end of the algorithm. (Or the processor with the most number of particles has at most *log(N)* more particles than the processor with the least number of particles). Furthermore the runtime of this algorithm is $\Theta(log(N))$. The proof of these properties has been omitted for brevity but will be published in a future paper.

```
BalanceWithPartnerWrapper()
{
    int NumRounds = ceiling(log2(numProcessors));

    for ( int k = 0; k < NumRounds; k++ )
    {
        BalanceWithPartner(k);
    }
}

BalanceWithPartner(int binaryDigit)
{
    // rank is this processor's MPI rank
    // all binary digits agree except binaryDigit is flipped in partner
    int partner = rank ^ (1 << binaryDigit);

    // Send and Recv with partner processor the number of particles each has
    int aveNumParticles = ( myNumParticles + partnerNumParticles ) / 2;

    if ( myNumParticles > partnerNumParticles ) // I am sending
    {
        // send (myNumParticles – aveNumParticles) particles to partner
    }
    else if ( myNumParticles < partnerNumParticles ) // I am receiving
    {
        // recv (partnerNumParticles – aveNumParticles) particles from partner
    }
}
```

Algorithm 1: Pseudocode showing the implementation of a scalable load balancing algorithm

Figure 2 illustrates the initial workload of each processor *before* load balancing, and what the workload looks like after each round of load balancing. The processor ranks are colored so that processors of the same color are "partners" and are communicating with each other; this partnership is also indicated with an arc drawn between the processor ranks communicating. After k rounds of load balancing, processors in groups of 2^k have the exact same workload.

We have also developed and implemented a generalized version of this load balancing algorithm that groups processors together in groups of size 2^w, where w is a user-settable parameter of the algorithm. The generalized algorithm has the advantage that the larger w is, the fewer rounds of load balancing are required to achieve global load balance (but each round takes longer). For brevity, we omit the detailed description of the general algorithm but will publish it in a future paper.

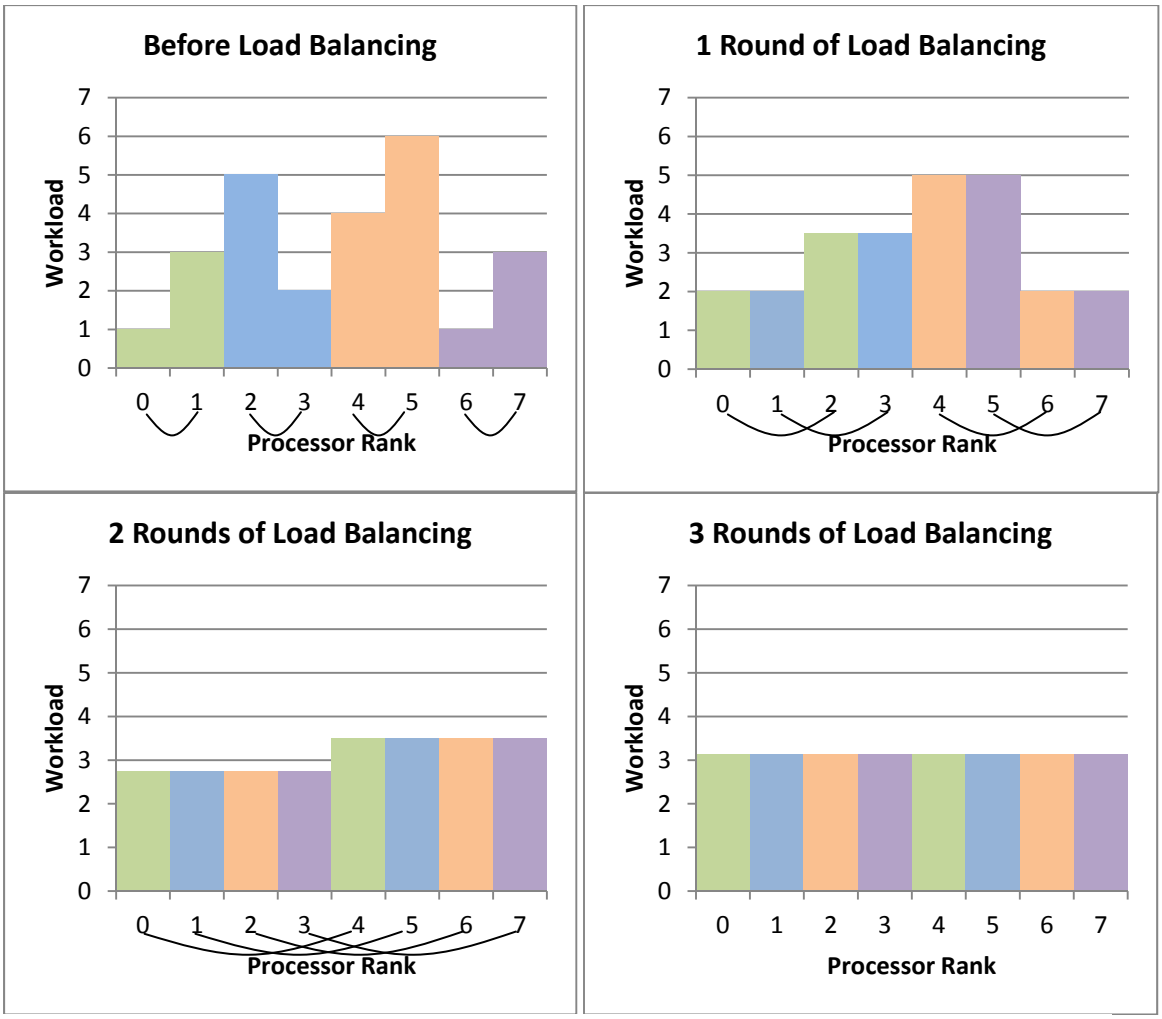

Figure 2: Illustration of processor workloads during load balancing.

4. NUMERICAL SCALING RESULTS

The test problem that we use in this paper to investigate parallel scalability is the Godiva critical assembly test problem (HEU-MET-FAST-001) [7], a uranium sphere of radius 8.7407 cm and density 18.74 g/cm^3. The isotopic atom fractions are U234 = 0.01025002, U235 = 0.9376829 and U238 = 0.05206708. The Godiva problem was modeled using one spherical combinatorial geometry cell.

4.1 Effects of Load Imbalance

In this section, we demonstrate the effects of load imbalance for the Godiva critical sphere problem when run with large numbers of processors. Lawrence Livermore National Laboratory has the Sequoia [8] super computer, which we use to study the processor scaling of the Godiva

Static-K criticality calculation up to 2 million (2^{21}) processors. We run Godiva as a *weak scaling* problem, that is, with fixed work per processor as we increase the number of processors. Each processor has 10,000 particles, so the total number of particles increases as we increase the processor count up to 2^{21} = 2,097,152 processors and 21 billion particles. We demonstrate that running this problem *without* load balancing results in the load balance efficiency decreasing as a function of Static-K generation. The load balance efficiency also decreases as the number of processors increases, because the amount of work on the most worked processor increases as the number of processors increases. If each processor starts with the same number of particles, the number of particles on a processor at the end of an iteration follows a normal distribution [6]. As the number of processors increases, the normal distribution is sampled more often. As a result, the most worked processor continues to increase and the efficiency decreases.

Figure 3 shows the load balance efficiency for 6 Godiva Static-K calculations that are NOT load balanced and run on 2^1, 2^4, 2^8, 2^{16}, and 2^{21} processors. In general, the load balance efficiency decreases as the number of processors increases and decreases with the number of iterations (Static K generations).

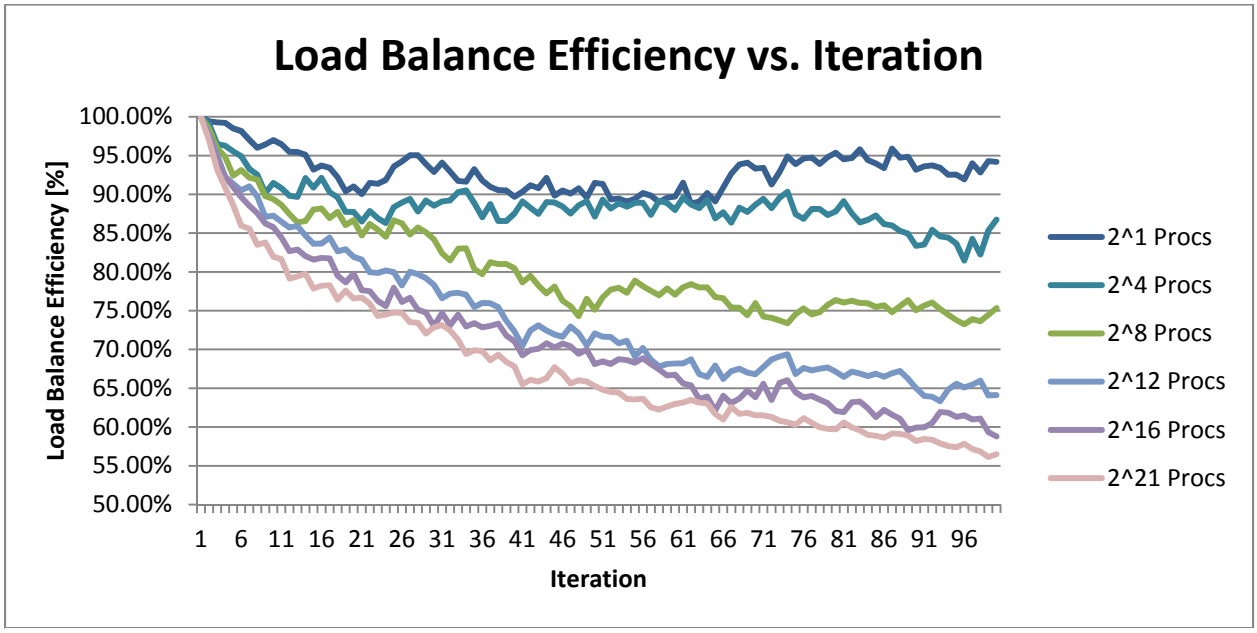

Figure 3: Load balance efficiency for 6 Godiva Static-K calculations that are NOT load balanced.

4.2 Partner Processor Algorithm Results

In this section, we investigate the parallel scalability of the partner processing load balancing algorithm. We ran a weak scaling study for N=1, 2, 4, 8, 16, …, 2^{17} = 131,072 processors on the Dawn supercomputer (IBM Blue Gene/P [9]) at Lawrence Livermore National Laboratory. The test problem is the Godiva critical assembly test problem again run with 10,000 particles per processor. So every time we doubled the number of processors, we doubled the total number of

particles. We ran 5 iterations of a Static-K calculation. The results of the scaling study are shown in Figure 4.

Figure 4 plots the wall time spent executing the load balancing algorithm at processor counts of $2^0, 2^1, 2^2, \ldots, 2^{17} = 131{,}072$. This plot is almost linear on a log-linear scale. The linear regression line through the data has slope 0.0633s/(doubling of processors). Each time we double the number of processors, we pay a fixed cost of 0.0633s, regardless of the number or processors. These calculations confirm that the load balancing algorithm is in fact proportional to the log of the number of processors.

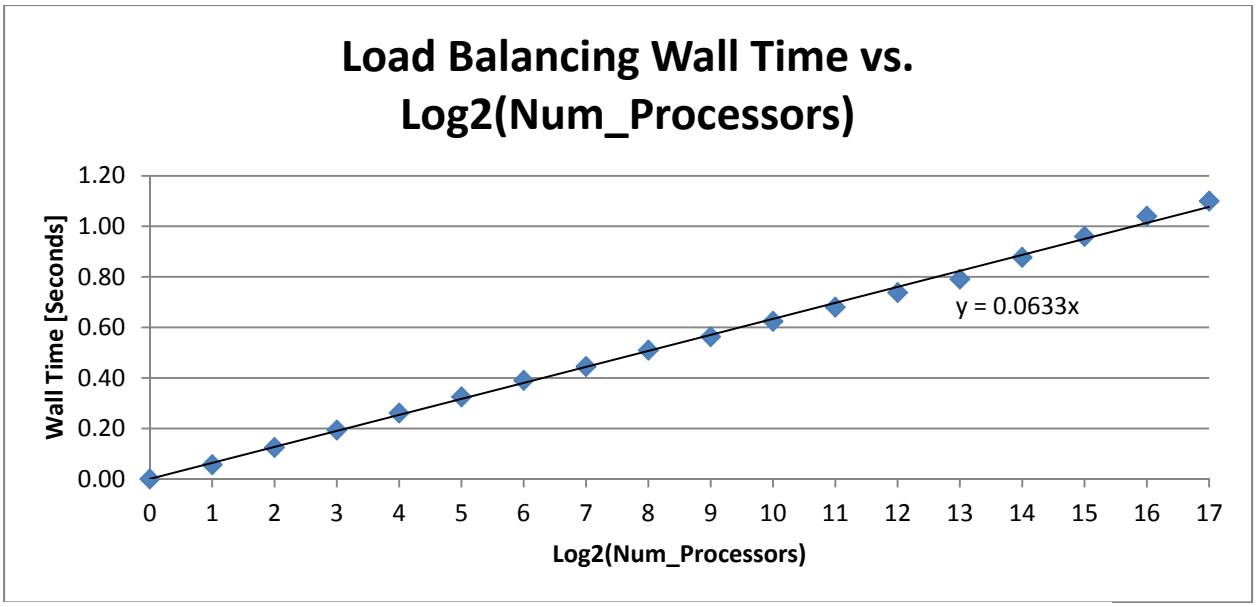

Figure 4: Wall time spent executing the load balancing algorithm.

4.3 Generalized Load Balancing Algorithm Results

We study the processor scaling of the Godiva Static-K criticality calculation up to 2 million (2^{21}) processors using the Sequoia supercomputer. The results in this section used the generalized load balancing algorithm briefly described in Sec. 3. We examine how the load balance efficiency scales, shown in Figure 5, and how the particle tracking time scales, shown in Figure 6. We also examine how the load balancing step itself scales with the number of processors, shown in Figure 7. We have proven the algorithm scales like $\Theta(\log(N))$, which is what we observe when we time the algorithm.

Figure 5 is a plot of the average load balance efficiency vs. $\log_2$(Num_Processors), comparing load balancing against not load balancing. With load balancing, the efficiency remains above 95%, even at 2 million processors. Without load balancing, the efficiency decreases with processor count down to 68% at 2 million processors.

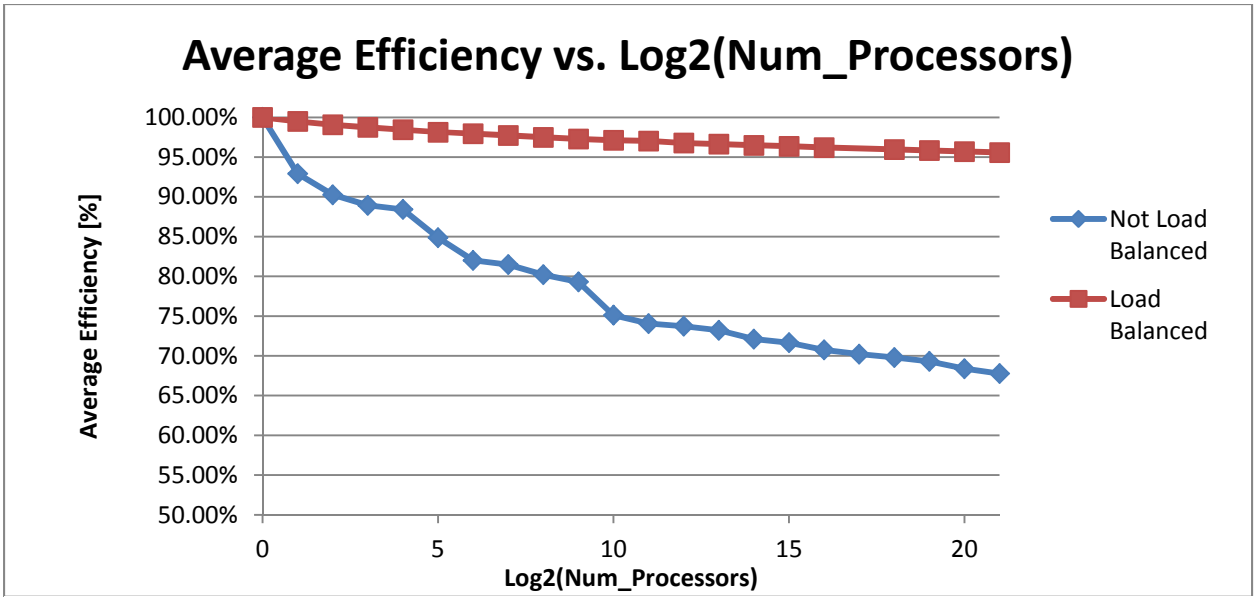

Figure 5: Average load balance efficiency vs. $\log_2$(Num_Processors)

Figure 6 shows the wall time spent tracking particles vs. $\log_2$(Num_Processors), comparing load balancing against not load balancing. With load balancing, we see essentially perfect scaling up to 2 million processors. Since each processor has the same amount of work, the calculation takes the same amount of time at any scale. Without load balancing, dispersion in the number of particles per processor occurs, and the calculation cannot proceed until the most worked processor has finished. This dispersion in processor workload results in an increase in tracking time.

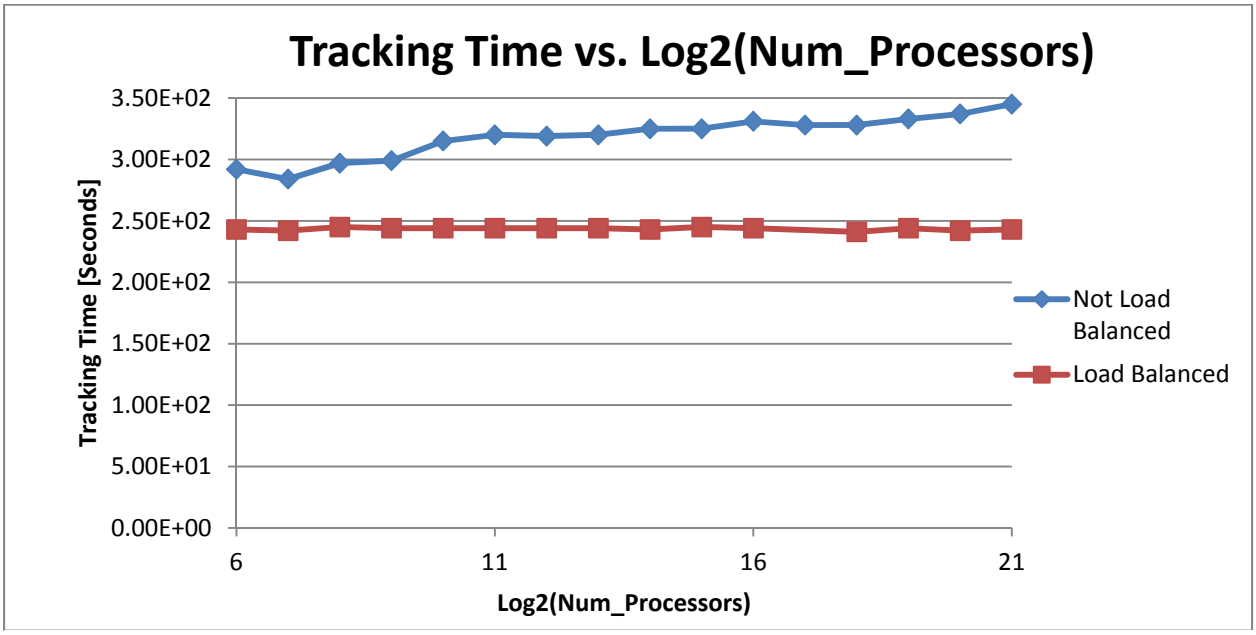

Figure 6: Wall time spent tracking particles vs. $\log_2$(Num_Processors)

Figure 7 shows the scaling behavior of the load balancing algorithm by plotting the load balancing time vs. $\log_2$(Num_Processors). The wall time should scale roughly like a straight line, since the algorithm is $\Theta(\log(N))$. The algorithm balances particle workloads in processors of group size 2^9, so we see an extra added expense at 2^9 and 2^{18}, since the algorithm needs to do an additional round of communication.

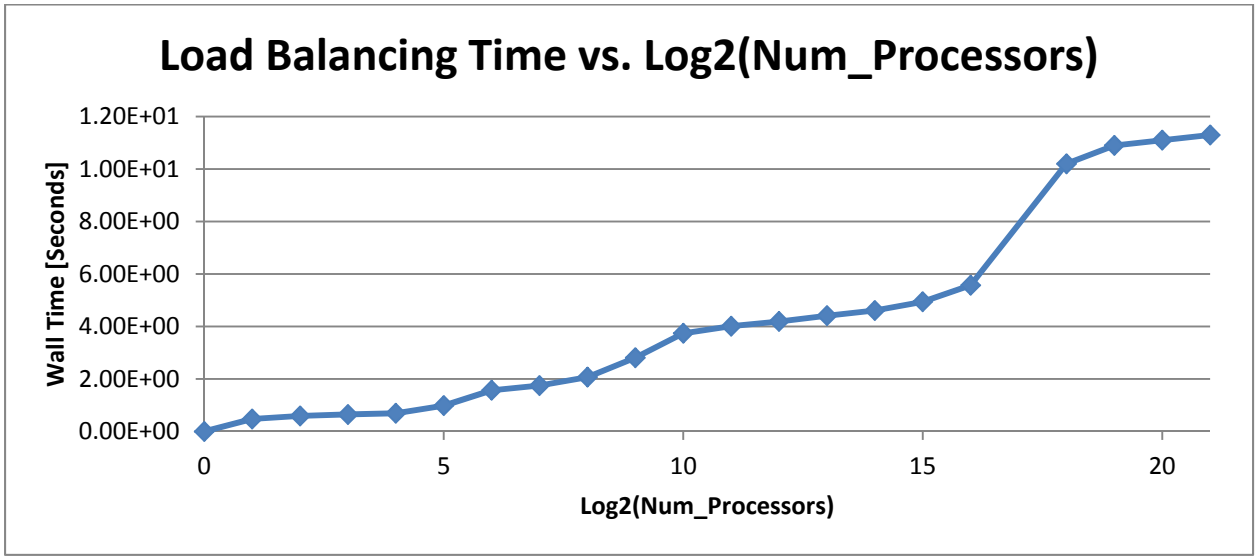

Figure 7: Scaling of the generalized load balancing algorithm

5. CONCLUSIONS

We have developed, implemented, and proven correct a scalable load balancing algorithm that has a computational complexity of $\Theta(\log(N))$. This algorithm globally balances all of the processors' work without globally knowing what the work distribution looks like. Through a sequence of local operations, global load balance can be achieved. We ran a scaling study up to 2^{21}=2,097,152 processors on the Sequoia (IBM BG/Q) supercomputer at Lawrence Livermore National Laboratory, and the observed results agree very well with the theoretical predictions. We believe that this algorithm applies to a broad class of homogeneous load balancing problems where the units of work all cost the same amount and any processor can process any unit of work. This algorithm allows for load balanced computations on the next generation of supercomputers with millions of cores. The new algorithm represents a significant improvement over our previous algorithm which would have taken 17 days on 2 million processors of Sequoia, while this new algorithm takes less than 1 second.

We also implemented a parameterized version of the load balancing algorithm in which the user may select the *Processor Group Size* parameter w which is the number of processors to group together when doing a local load balance step. Instead of load balancing processors 2 at a time, the general algorithm load balances processors in groups of size 2^w. We will publish the full details of the generalized algorithm in a future paper.

In a future paper, we will also consider scalable load balancing for domain decomposed problems, where the geometry information (in addition to the particles) are distributed across processors and allowed to scale with the number of processors.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

REFERENCES

[1] "MPI: A Message-Passing Interface Standard," (2011). [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi.

[2] P. Brantley and M. McKinley, "Mercury Web Site," (2011). [Online]. Available: https://wci.llnl.gov/codes/mercury/.

[3] M. O'Brien, J. Taylor and R. Procassini, "Dynamic Load Balancing of Parallel Monte Carlo Transport Calculations," in *The Monte Carlo Method: Versatility Unbounded In A Dynamic Computing World*, Chattanooga, TN, (2005).

[4] R. Procassini, M. O'Brien and J. Taylor, "Load Balancing of Parallel Monte Carlo Transport Calculations," in *Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Application*, Palais des Papes, Avignon, France, (2005).

[5] G. Greenman, M. O'Brien, R. Procassini and K. Joy, "Enhancements to the Combinatorial Geometry Particle Tracker in the Mercury Monte Carlo Transport Code: Embedded Meshes and Domain Decomposition," in *Proceeding from the ANS Mathematics and Computation 2009 Meeting*, (2009).

[6] P. Romano and B. Forget, "Parallel Fission Bank Algorithms in Monte Carlo Criticality Calculations," *Nuclear Science and Engineering*, vol. 170, no. 2, pp. 125-135, (2012).

[7] "International Handbook of Evaluated Criticality Safety Benchmark Experiments," Nuclear Energy Agency, on CD-ROM (2010).

[8] "Advanced Simulation and Computing- Sequoia," Lawrence Livermore National Laboratory, (2012). [Online]. Available: https://asc.llnl.gov/computing_resources/sequoia/.

[9] B. Barney, "Using the Dawn BG/P System," (2011). [Online]. Available: https://computing.llnl.gov/tutorials/bgp.

[10] G. Zheng, A. Bhatele, E. Meneses and L. V. Kale, "Periodic Hierarchical Load Balancing for Large Supercomputers," *International Journal of High Performance Computing Applications*, (2011).

[11] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction," Ph. D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, (2005).

[12] C. Englemann and A. Geist, "Super-Scalable Algorithms for Computing on 100,000 Processors.," *Proceedings of ICCS*, (2005).

[13] P. Balaji, D. Buntinas, D. Goodell and W. Gropp, "MPI on a Million Processors," in *EuroPVMMPI'09*, Helsinki, Finland, (2009).