Dynamic Load Balancing for Petascale Quantum Monte Carlo Applications: The Alias Method

C.D. Sudheer[a], S. Krishnan[b], A. Srinivasan[b], P. R. C. Kent[c]

[a] *Dept. of Mathematics and Computer Science, Sri Sathya Sai Institute of Higher Learning, Prasanthi Nilayam, India*

[b] *Dept. of Computer Science, Florida State University, Tallahassee, FL 32306*

[c] *Center for Nanophase Materials Sciences and Computer Science and Mathematics Division, Oak Ridge National Lab, Oak Ridge, TN 37831*

Abstract

Diffusion Monte Carlo is a highly accurate Quantum Monte Carlo method for the electronic structure of materials, but it requires frequent load balancing or population redistribution steps to maintain efficiency on parallel machines. This step can be a significant factor affecting performance, and will become more important as the number of processing elements increases. We propose a new dynamic load balancing algorithm, the Alias Method, and evaluate it theoretically and empirically. An important feature of the new algorithm is that the load can be perfectly balanced with each process receiving at most one message. It is also optimal in the maximum size of messages received by any process. We also optimize its implementation to reduce network contention, a process facilitated by the low messaging requirement of the algorithm: a simple renumbering of the MPI ranks based on proximity and a space filling curve significantly improves the MPI Allgather performance. Empirical results on the petaflop Cray XT Jaguar supercomputer at ORNL showing up to 30% improvement in performance on 120,000 cores. The load balancing algorithm may be straightforwardly implemented in existing codes. The algorithm may also be employed by any method with many near identical computational tasks that requires load balancing.

Email addresses: cdsudheerkumar@sssihl.edu.in (C.D. Sudheer), krishnan@cs.fsu.edu (S. Krishnan), asriniva@cs.fsu.edu (A. Srinivasan), kentpr@ornl.gov (P. R. C. Kent)

Keywords: Quantum Monte Carlo, Parallel Computing, Load Balancing

1. Introduction

Quantum Monte Carlo (QMC) is a class of quantum mechanics-based methods for electronic structure calculations [1, 2, 3]. These methods can achieve much higher accuracy than well-established techniques such as Density Functional Theory (DFT) [4] by directly treating the quantum-mechanical many-body problem. However, this increase in accuracy comes at the cost of substantially increased computational effort. The most common QMC methods are nominally cubic scaling, but have a large prefactor, making them several orders of magnitude more costly than the less-accurate DFT calculation. On the other hand, QMC can today effectively use the largest parallel machines, with $O(10^5)$ processing elements, while DFT can not use these machines routinely. However, with the expected arrival of machines with orders of magnitude more processing elements than common today, it is important that all the key algorithms of QMC are optimal and remain efficient at scale.

Diffusion Monte Carlo (DMC) is the most popular modern QMC technique for accurate predictions of materials and chemical properties at zero temperature. It is implemented in software packages such as CASINO [5], CHAMP [6], QMCPack [7], and QWalk [8]. Unlike some Monte Carlo approaches, this method is not trivially parallel and requires communications throughout the computation.

The DMC computation involves a set of random walkers, where each walker represents a quantum state. At each time step, each walker moves to a different point in the configuration space, with this move having a random component. Depending on the energy of the walker in this new state relative to the average energy of the set of walkers (or a reference energy related to the average), the walker might be either terminated or new walkers created at the same position. Alternatively, weights may be associated with each walker, and the weights increased or decreased appropriately. Over time, this process creates a load imbalance, and the set of walkers (and any weights) must be rebalanced. For optimum statistical efficiency, this rebalancing should occur every single move [9].

DMC is parallelized by distributing the set of walkers over the available compute cores. The relative cost of load balancing the walkers as well as the

inefficiency from the statistical fluctuations in walker count can be minimized if the number of walkers per compute element is kept large. This approach has typically been used on machines with thousands to tens of thousands of cores. However, with increasing core count the total population of walkers is increased. This is undesirable since (1) there is an equilibration time for each walker that does not contribute to the final statistics and physical result, (2) it is usually preferable to simulate walkers for longer times, enabling any long term trends or correlations to be determined, (3) the amount of memory per compute element is likely to reduce in future, necessitating smaller populations per compute element. On the highest-end machines, it is desirable to use very few walkers (one or two) per compute element and assign weights to the walkers, instead of adding or subtracting walkers, to avoid excessively large walker counts and to avoid the large fluctuations in computational effort that would result from even minor fluctuations in walker count.

Since load balancing is in principle a synchronous, blocking operation, requiring communication between all compute elements, it is important that the load balancing method is highly time efficient and makes very effective use of the communications network, minimizing the number and size of messages that must be sent. It is also desirable that the algorithm is simple to enable optimization of the messaging for particular networks, and to simplify use of latency hiding techniques through overlap of computation and communications. We note that CASINO [5] recently transitioned [10] to using asynchronous communications and suspect that other codes may use some of these techniques, but apart from [10], they have not been formally described.

In this paper we discuss a new load balancing algorithm which can be used to load balance computations involving near identical independent tasks such as those in DMC (we consider each random walker a task in the description of our load balancing algorithm). The algorithm has the interesting feature that each process needs to receive tasks from at most one other process. We optimize this algorithm on the peta-flop Cray XT5 supercomputer at Oak Ridge National Laboratory and show, using data from the simulation of the Cr_2 molecule, that it improves performance over the existing load balancing implementation of the QWalk code by up to 30% on 120,000 cores. Moreover, due to the optimal nature of the algorithm we expect its utility and effectiveness to increase with the multiple orders of magnitude increase in compute elements expected in the coming years.

1.1. Load Balancing Model Definitions

Dynamic load balancing methods often consist of the following three steps. (i) In the *flow computation* step, we determine the number of tasks that need to be sent by each process to other processes. (ii) In the *task identification* step, we identify the actual tasks that need to be sent by each process. (iii) In the *migration* step, the tasks are finally sent to the desired processes. Since we deal with identical independent tasks, the second step is not important; any set of tasks can be chosen. Our algorithm determines the flow (step (i)) such that step (iii) will be efficient, under certain performance metrics.

We assume that a collection of P processes need to handle a set of T identical tasks (that is, each task requires the same computation time), which can be executed independently. Before the load balancing phase, the number of tasks with process i, $1 \leq i \leq P$, is T_i. After load balancing, each process will have at most $\lceil T/P \rceil$ tasks (we are assuming that the processors are homogeneous, and therefore process tasks at the same speed). This redistribution of tasks is accomplished by having each process i send t_{ij} tasks to processes j, $1 \leq i, j \leq P$, where non-zero values of t_{ij} are determined by our algorithm for flow computation, which we describe in § 3. Of course, most of the t_{ij}s should be zero, in order to reduce the total number of messages sent. In fact, at most $P-1$ of the possible $P(P-1)$ values of t_{ij} will be non-zero in our algorithm.

The determination of t_{ij}s is made as follows. The processes perform an "all-gather" operation to collect the number of tasks on each process. Each process k independently implicitly computes the flow (all non-zero values of t_{ij}, $1 \leq i, j \leq P$) using the algorithm in § 3, and then explicitly determines which values of t_{kj} and t_{jk} are non-zero, $1 \leq j \leq P$.

The algorithm to determine non-zero t_{ij}s takes $O(P)$ time, and is fast in practice. We wish to minimize the time taken in the actual migration step, which is performed in a decentralized manner by each process. In some load balancing algorithms [11], a process may not have all the data that it needs to send, and so the migration step has to take place iteratively, with a process sending only data that it has in each iteration. In contrast, the t_{ij}s generated by our algorithm never require sending more data than a process initially has, and so the migration step can be completed in one iteration. In fact, no process *receives* a message from more than one process, though processes may need to *send* data to multiple processes.

The outline of the rest of the paper is as follows. We summarize related work in § 2. In § 3, we describe our algorithm for dynamic load balancing. We first describe the algorithm when T is a multiple of P, which is the ideal case, and then show how the algorithm can be modified to deal with the situation when T is not a multiple of P. In § 4, we define a few metrics for the time complexity of the load re-distribution step, and theoretically evaluate our algorithm in terms of those. In particular, we show that it is optimal in the maximum number of messages received by any process and in the maximum size of messages received by any process. We then report results of empirical evaluation of our method and comparisons with an existing QMC dynamic load balancing implementation, in § 5. We finally summarize our conclusions in § 6.

2. Related Work

Load balancing has been, and continues to be, an important research issue. Static partitioning techniques try to assign tasks to processes such that the load is balanced, while minimizing the communication cost. This problem is NP-hard in most reasonable models, and thus heuristics are used. Geometric partitioning techniques can be used when the tasks have coordinate information, which provide a measure of distance between tasks. Graph based models abstract tasks as weighted vertices of a graph, with weights representing computational loads associated with tasks. Edges between vertices represent communication required, when one task needs information on another task. A variety of partitioning techniques have been studied, with popular ones being spectral partitioning [12, 13] and multi-level techniques [14, 15, 16, 17], and have been available for a while in software such as Chaco and Metis.

Dynamic load balancing schemes start with an existing partition, and migrate tasks to keep load balance, while trying to minimize the communication cost of the main computation. A task is typically sent to a process that contains neighbors of the task in the communication graph, so that the communication cost of the main computation is minimized[1]. Other schemes make larger changes to the partitions, but remap the computation such that the cost of migration is small [18].

[1]When tasks are fairly independent, as in QMC, it is reasonable to model it as a complete graph, indicating that a task can be migrated to any process.

The diffusion scheme is a simple and well-known scheme sending data to neighboring processes [19, 20, 21, 22]. Another scheme, proposed in [11], is also based on sending tasks to neighbors. It is based on solving a linear system involving the Laplacian of the communication graph. Both these schemes require the tasks to be arbitrarily divisible for the load balancing to work. For example, one should be able to send 0.5 tasks, 0.1 tasks, etc. Modified versions of diffusive type schemes have also been proposed which remove restrictions on arbitrary divisibility [23]. Multi-level graph partitioning based dynamic schemes are also popular [24]. Hyper-graphs generalize graphs using hyper-edges, which are sets of vertices with cardinality not limited to two. Hyper-graph based partitioning has also been developed [25]. Software tools, such as, JOSTLE [26], ParMetis, and Zoltan are available, implementing a variety of algorithms.

Apart from general purpose algorithms, there has also been interest in the development of algorithms for specific applications, such as [27]. There has also been work performed on taking factors other than communication and computation into account, such as IO cost [28].

There has been much work performed on load balancing independent tasks (bag of tasks) in the distributed and heterogeneous computing fields [29, 30, 31, 32]. Many of the scheduling algorithms try to minimize the makespan, which can be considered a type of load balancing. They consider issues such as differing computing power of machines, online scheduling, etc.

Within the context of QMC and DMC, we are not aware of any published work specifically focusing on the algorithms used for load balancing, although optimizations to existing implementations have been described [10]. Since all QMC codes must perform a load balancing step, each must have a reasonably efficient load balancing implementation, at least for modest numbers of compute elements. However, the methods used have not been formally described and we do not believe any existing methods share the optimality features of the algorithm described below.

3. The Alias Method Based Algorithm for Dynamic Load Balancing

Our algorithm is motivated by the alias method for generating samples from discrete random distributions. We therefore refer to our algorithm as the Alias method for dynamic load balancing. There is no randomness in our algorithm. It is, rather, based on the following observation used in a

deterministic pre-processing step of the alias method for the generation of discrete random variables. If we have P bins containing kP objects in total, then it is possible to re-distribute the objects so that each bin receives objects from at most one other bin, and the number of objects in each bin, after the redistribution, is exactly k. Walker [33] showed how this can be accomplished in $O(P \log P)$ time. This time was reduced to $O(P)$ by [34] using auxiliary arrays. In Algorithm 1 below, we describe our in-place implementation that does not use auxiliary arrays, except for storing a permutation vector.

We assume that the input to Algorithm 1 is an integer array A containing the number of objects in each bin. Given A, we can compute k easily in $O(P)$ time, and will also partition it around k in $O(P)$ time so that all entries with $A[i] < k$ occur before any entry with $A[j] > k$. We will assume that $A[i] \neq k$, because other bins do not need to be considered – they have the correct number of elements already, and our algorithm does not require redistribution of objects to or from a bin that has k objects. If we store the permutation while performing the partitioning, then the actual bin numbers can easily be recovered after Algorithm 1 is completed. This algorithm runs only with $P \geq 2$, because otherwise all the bins already have k elements each. We assume that a pre-processing step has already accomplished the above requirements in $O(P)$ time.

Algorithm 1:

Input: An array of non-negative integers $A[1 \cdots P]$ and an integer $k > 0$, such that $\sum_{i=1}^{P} A[i] = kP$, entries of A have been partitioned around k, and $P \geq 2$. $A[i]$ gives the number of objects in bin i, and $A[i] \neq k$.

Output: Arrays $S[1 \cdots P]$ and $W[1 \cdots P]$, where $S[i]$ gives the bin from which bin i should get $W[i]$ objects, if $S[i] \neq 0$.

Algorithm:

1. Initialize arrays S and W to all zeros.
2. $s \leftarrow 1$.
3. $l \leftarrow \min\{j | A[j] > k\}$.
4. while $l > s$
   (a) $S[s] \leftarrow l$.
   (b) $W[s] \leftarrow k - A[s]$.
   (c) $A[l] \leftarrow A[l] - W[s]$.
   (d) if $A[l] < k$ then
      i. $l \leftarrow l + 1$.

(e) $s \leftarrow s + 1$.

It is straightforward to see the correctness of Algorithm 1 based on the following loop invariants at the beginning of each iteration in step 4: (i) $A[i] \geq k,\ l \leq i \leq P$, (ii) $0 \leq A[i] < k,\ s \leq i \leq l-1$, and (iii) $A[i] + W[i] = k,\ 1 \leq i \leq s-1$. Since bin l needs to provide at most k objects to bin s, it has a sufficient number of objects available, and also as a consequence of the same fact, $A[l]$ will not become negative after giving $W[s]$ objects to bin s. The last clause of the loop invariant proves that all the bins will have k objects after the redistribution. We do not formally prove the loop invariants, since they are straightforward.

In order to evaluate the time complexity, note that in the while loop in step 4, l and s can never exceed P. Furthermore, each iteration of the loop takes constant time, and s is incremented once each iteration. Therefore, the time complexity of the while loop is $O(P)$. Step 3 can easily be accomplished in $O(P)$ time. Therefore the time complexity of this algorithm is $O(P)$.

Load balancing when T is a multiple of P. Using Algorithm 1, a process can compute t_{ij}s as follows, if we associate each bin with a process[2] and the number of objects with the number of tasks:

$$t_{S[i]i} \leftarrow W[i],\ S[i] \neq 0. \tag{1}$$

All other t_{ij}s are zero. Of course, one needs to apply the permutation obtained from the partitioning before performing this assignment. Note that the loop invariant mentioned for Algorithm 1 also shows that a process always has sufficient data to send to those that it needs to; it need not wait to receive data from any other process in order to have sufficient data to send, unlike some other dynamic load balancing algorithms [11].

Load balancing when T is not necessarily a multiple of P. The above case considers the situation when the total number of tasks is a multiple of the total number of processes. We can also handle the situation when this is not true, using the following modification. If there are T tasks and P processes, then let $k = \lceil T/P \rceil$. For balanced load, no process should have more than

[2]In our algorithm, processes that already have a balanced load do not participate in the redistribution of tasks to balance the load. Therefore, we use P to denote the number of processes with *unbalanced loads* in the remainder of the theoretical analysis.

k tasks. We modify the earlier scheme by adding $kP - T$ fake "phantom" tasks. This can be performed conceptually by incrementing $A[i]$ by one for $kP - T$ processes before running Algorithm 1 (and even before the pre-processing steps involving removing entries with $A[i] = k$ and partitioning). The total number of tasks, including the phantom ones, is now kP, which is a multiple of P. So Algorithm 1 can be used on this, yielding k tasks per process. Some of these are phantom tasks, and so the number of tasks is at most k, rather than exactly k. We can account for the phantom tasks by modifying the array S as follows, after completion of Algorithm 1. Let F be the set of processes to which the fake phantom tasks were added initially (by incrementing their A entry). For each $j \in F$, define $r_j = \min\{i | S[i] = j\}$. If r_j exists, then set $W[r_j] \leftarrow W[r_j] - 1$. This is conceptually equivalent to making each process that initially had a phantom task to send this task to the first process to whom it sends anything. Note that on completion of the algorithm, no process has more than two phantom tasks, because in the worst case, it had one initially, and then received one more. So the total number of tasks on any process after redistribution will vary between $k-2$ to k. The load is still balanced, because we only require that the maximum load not exceed $\lceil T/P \rceil$ after the redistribution phase[3]. This modified algorithm can be implemented with the same time complexity as the original algorithm.

4. Theoretical Analysis

We next analyze the performance of the migration step of the load balancing algorithm, when using the t_{ij}s as computed by Algorithm 1 in § 3. We define a few performance metrics, and give the approximation ratio of our algorithm (that is, an upper bound on the ratio of the time taken by our algorithm to that of an optimal one, in the metric considered). The results are summarized in Table 1. We assume that $P \geq 2$ in Algorithm 1; otherwise no load redistribution is performed, either by our algorithm or an optimal one, and the approximation ratio, 0/0 under any of our metrics, is undefined. The analyses below consider the case where T is a multiple of P. If T is not a multiple of P, then all the bounds still hold, except in the Maximum-Tasks-Received metric, where our algorithm may have a value one more than the optimal.

[3]In our implementation, the phantom tasks are not actually sent, and they do not even exist in memory.

Maximum-Receives:. In this metric, the time taken in the migration step is the maximum number of messages received by any process. Formally, it is given by $\max_j |\{i|t_{ij} \neq 0\}|$. This metric is reasonable to use if sending can be performed asynchronously, and if the latency overhead of sending a message is very high, as is common in many distributed environments. The receive operation still blocks until the message is received, and so this cost can dominate if the data size is not very large.

In the alias algorithm, any process receives at most one message. An optimal algorithm too requires at least one message to be received on some process, since the load is unbalanced. So the *approximation ratio is 1*.

Maximum-Tasks-Received:. In this metric, the time taken is estimated as the largest number of tasks received by any process. That is, it is $\max_j \sum_{i=1}^{P} t_{ij}$, which is the largest total sizes of messages received by any process. This metric is reasonable to use if sends are asynchronous as above, receives blocking, and message sizes fairly large.

Let $d = \max_i T/P - T_i$. That is, it is the largest deficit in number of tasks initially on any process. The optimal algorithm needs to have the process with this deficit receive at least d tasks. So the optimal solution is at least as large as d.

We will next show that the alias based scheme can redistribute the load with the maximum-tasks-received being d, thereby being optimal in this metric. Assume that this is not true, and that the alias-based algorithm takes greater than d. Let i be the smallest indexed process with $W[i] > d$. Let $W[i] = \hat{d}$. Process i cannot be one that initially had a deficit, because from Algorithm 1, one can see that a process with an initial deficit never sends tasks to any process. So its deficit will never increase beyond its initial one, which is at most d, because $W[i] = k - A[i] \leq d$ for such processes, where $k = T/P$. So i must be a process that initially had an excess (that is, i is at least as large as the initial value of l in Algorithm 1). Process i must have contributed to the $W[j]$ of some process j in order to later experience a deficit of $\hat{d}$. Consider the situation just before the last time it contributed to some $W[j]$. $A[i] \geq k$ at that time. After the contribution, $W[j] \leq d$ because i is the smallest indexed process with W entry greater than d, and in Algorithm 1, processes contribute only to lower indexed processes. So, $A[i] \geq k - d$ after the contribution. The deficit it experiences is thus at most d, and so $W[i]$ must be at most d. Thus the assumption is false, and $W[i] \leq d$ for all i. This shows optimality of the alias method-based algorithm in this

metric, and so its *approximation ratio is 1*.

Total-Messages:. In this metric, we count the total number of messages, $\sum_j |\{i|t_{ij} \neq 0\}|$ or $\sum_i |\{j|t_{ij} \neq 0\}|$. That is, we count the total number of sends or the total number of receives. This can be a reasonable metric if many messages are being sent. In that case, we want to reduce congestion on the network, which may be reduced by reducing the total number of messages.

The optimal solution sends at least $P/2$ messages for the following reason. Let us partition the processes into sets D and E, where D consists of all processes that have an initial deficit, and E consists of all processes that have an initial excess. Each process in D must receive at least one message to balance its load, and each process in E must send at least one message to balance its load. Thus $\max\{|D|, |E|\}$ is a lower bound on the cost of the optimal solution. But $|D| + |E| = P$. Therefore $\max\{|D|, |E|\} \geq P/2$, and so the cost of the optimal solution is at least $P/2$. The alias-based scheme has each process receive at most 1 message. So the total number of messages received (or, equivalently, sent) is at most P (in fact, it is $P - 1$, if we note, from Algorithm 1, that process P cannot receive a message). So the *approximation ratio is 2*.

Example 1: We next show that the above bound is tight. Consider $P = 2n$ processes for some sufficiently large n, with $T_1 \cdots T_{n-1} = n-1, T_n = 2, T_{n+1} = 2n - 2, T_{n+2} \cdots T_{2n} = n + 1$. It is possible to balance the load by having process $n+1$ send $n-2$ tasks to process n, and having processes $n + 2 \cdots 2n$ send one tasks each to processes $1 \cdots n - 1$ respectively. The total number of messages sent will be $n = P/2$, which is also the best possible, as shown above. In the alias method-based algorithm, process $n + 1$ sends one task each to processes $1 \cdots n - 1$, process $n + 2$ sends $n - 2$ tasks to process n, process $n + 3$ sends one task to process $n + 1$ and $n - 3$ tasks to process $n + 2$, and each process $n + i$ sends $n - (i - 1)$ tasks to process $n + i - 1$, $4 \leq i \leq n$. Each process other than process $2n$ receives one messages, and so the total number of messages is $2n - 1 = P - 1$. The approximation factor is $(P - 1)/(P/2) = 2(1 - 1/P)$. Since P can be arbitrarily large, the bound is tight.

Total-Tasks-Sent:. In this metric, we count the total number of tasks, $\sum_i \sum_j t_{ij}$ sent. This is a reasonable metric to use if we send several large messages, because these can then congest the network.

The *approximation factor is unbounded*, as can be seen from Example 1 above. In that example, there are $n-1$ messages of length 1 each from process $n+1$, one message of length $n-2$ from process $n+2$, one message of length 1 and one message of length $n-3$ from process $n-3$, and messages of length $1, 2, \cdots, n-3$ from processes $2n, 2n-1, \cdots, n+4$ respectively. So, the total number of tasks sent is $3n-5+(n-3)(n-2)/2$. An optimal flow for this example in the Maximum-Tasks metric sends $2n-3$ tasks, as explained above, and so the optimal schedule for the Total-Tasks metric sends at most $2n-3$ tasks. The approximation ratio is then $\Omega(n) = \Omega(P)$, which is unbounded, because the number of processes, P, is unbounded.

Maximum-Sends and Maximum-Tasks-Sent:. We have earlier defined metrics that use the number and sizes of messages received by a process. In analogy with those, we now discuss metrics that are based on the number and sizes of messages sent. In the Maximum-Sends metric, we count the maximum number of messages sent by any process. The approximation factor is unbounded in this metric, as can be seen from Example 1. An optimal schedule for the Total-Messages metric sends a maximum of 1 message from each process for this example, as described above. The alias-based algorithm has process $n+1$ send $n-1 = P/2-1$ messages. Since P can be arbitrarily large, the *approximation factor is unbounded* in this metric.

In the Maximum-Tasks-Sent metric, we count $\max_i \sum_j t_{ij}$, which is the largest total sizes of messages sent by any process. The *approximation factor is unbounded* here too, as can be seen from the following example.

Example 2: Let $T_1 = 1,\ T_2 \cdots T_P = P+1$. It is possible to balance the load by having processes $2, \cdots, P$ send one task each to process 1, for this metric to have value 1. It is easy to see that this is also optimal. In the alias method-based algorithm, process 2 sends a message with $P-1$ tasks to process 1, and each process $3, 4, \cdots, P$ sends messages of size $P-2, P-3, \cdots, 1$ to process $2, 3, \cdots, P-1$ respectively. The maximum size is sent by process 2 with value $P-1$, yielding an approximation factor of $P-1$, which is unbounded, because P can be arbitrarily large.

5. Empirical Results

5.1. Experimental Setup

The experimental platform is the Cray XT5 Jaguar supercomputer at ORNL. It contains 18,688 dual hex-core Opteron nodes running at 2.6GHz

Method	Approximation factor
Maximum-Receives	1
Total-Messages	2
Maximum-Tasks-Sent	∞
Maximum-Sends	∞
Total-Tasks-Sent	∞
Maximum-Tasks-Received	1

Table 1: *Approximation factors under different metrics.*

with 16GB memory per node. The peak performance of the machine is 2.3 petaflop/s. The nodes are connected with SeaStar 2+ routers having a peak bandwidth of 57.6GB/s, with a 3-D torus topology. Compute Node Linux runs on the compute nodes.

In running the experiments, we have two options regarding the number of processes per node. We can either run one process per node or one process per core. QMC software packages were originally designed to run one MPI process per core. The trend now is toward one MPI process per node, with OpenMP threads handling separate random walkers on each core. Qmcpack already has this hybrid parallelization implemented, and some of the other packages are expected to have it implemented in the near future. We assume such a hybrid parallelization, and have one MPI process per node involved in the load balancing step.

In our experiments, we consider a granularity of 24 random walkers per node, that is, 2 per core. This is a level of granularity that we desire for QMC computations in the near future. Such scalability is currently limited by the periodic collective communication and load balancing that is required.

Both these are related in the following manner. The first step leads to termination or creation of new walkers, which in turn requires load balancing. There is some flexibility in the creation and termination of random walkers. Ideally, the load balancing results both in reduced wall clock time per step (of all walkers) and an improved statistical efficiency.

We note that there is some flexibility in the creation and termination of random walkers. Ideally the load balancing is performed after every time step, to obtain the best statistical error and to minimize systematic errors due to the finite sized walker population. However, the overhead on large parallel machines can hinder this, and so one may perform them every few

iterations instead. Our goal is to reduce these overheads so that these steps can be performed after every time step. For large physical systems where the computational cost per step is very high, these overheads may be relatively small compared with the computation cost. However, for small to moderate sized physical systems, these overheads can be large, and we wish to efficiently apply QMC even to small physical systems on the largest parallel computational systems.

We consider a small system, a Cr_2 molecule, with an accurate multideterminant trial wavefunction. The use of multideterminants increases computational time over the use of a single determinant. However, it provides greater accuracy, which we desire when performing a large run. The computation time per time step per walker is then around 0.1 seconds. The two collective steps mentioned above consume less than 10% of the total time on a large machine (the first step does not involve just collective communication, but also involves other global decisions, such as branching). Even then, on 100,000 cores, this is equivalent to wasting 10,000 cores. We can expect these collective steps to consume a larger fraction of time at even greater scale.

In evaluating our load balancing algorithm, we used samples from the load distribution observed in a long run of the above physical system. Depending on the details of the calculations, the amount of data to be transferred for each random walker can vary from 672B to 32KB for Cr_2. We compared our algorithm against the load balancing implementation in QWalk. The algorithm used in QWalk is optimal in the maximum number of tasks sent by any processor and in the total number of tasks sent by any processor, but not on the maximum or total number of messages sent; these are bounded by the maximum imbalance and the sum of load imbalances respectively. One may, therefore, expect that algorithm to be more efficient than ours for a sufficiently large task sizes, and ours to be better for small sizes. Also, the time taken for the flow computation in QWalk is O(P + total_load), where P is the number of nodes.

Each experiment involved 11 runs. As we show later, inter-job contention on the network can affect the performance. In order to reduce its impact, we ignore the results of the run with the largest total time. In order to avoid bias in the result, we also drop the result of the run with the smallest time. For a given number of nodes, all runs for all task sizes for both algorithms are run on the same set of nodes, with one exception mentioned later.

5.2. Results

Our testing showed that the time taken for the alias method is linear in the number of nodes, as expected theoretically (not shown). The maximum time taken by any node can be considered a measure of the performance the algorithm, because the slowest processor limits the performance. Figure 1 shows the average, over all the runs, of the maximum time for the following components of the algorithm. (We refer to it in the figure caption as the 'basic alias method', in order to differentiate it from a more optimized implementation described later.) Note that the maximum for each component may occur on different cores, and so the maximum total time for the algorithm over all the cores may be less than the sum of the maximum times of each component. We can see that communication operations consume much of the time, and the flow computation is not the dominant factor, even with a large number of nodes. The MPI_Isend and MPI_Irecv operations take little time. However MPI_Waitall and MPI_Allgather consume a large fraction of the time. It may be possible to overlap computation with communication to reduce the wait time. However, the all-gather time is still a large fraction of the total time.

In interpreting the plot in Figure 1, one needs to note that it is drawn on a semi-log scale. The increase in time, which appears exponential with the number of cores, is not really so. A linear relationship would appear exponential on a semi-log scale. On the other hand, one would really expect a sub-linear relationship for the communication cost. The all-gather would increase sub-linearly under common communication cost models. In the absence of contention, the cost of data transfer need not increase with the number of cores for the problem considered here; the maximum imbalance is 4, each node has 6 communication links, and so, in principle, if the processes are ideally ordered, then it is possible for data to travel on different links to nearby neighbors which would be in need of tasks. The communication time can, thus, be held constant. We can see from this figure that the communication cost (essentially the wait time) does increase significantly. The communication time for 12,000 cores is 2-3 times the time without contention, and the time with 120,000 cores is 4-6 times that without contention. The cause for contention is that the routing on this machine uses fixed paths between pairs of nodes, and sends data along the x coordinate of the torus, in the direction of the shortest distance, then in the y direction, and finally

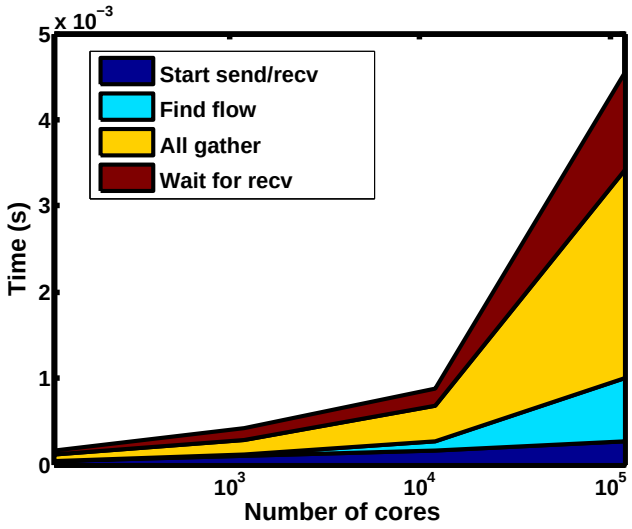

Figure 1: Maximum time taken for different components of the basic Alias method with task size 8KB.

in the z direction[4]. Multiple messages may need to share a link, which causes contention.

In fig. 2, we consider the mean value of the different components in each run, and plot the average of this over all runs. We can see that the wait time is very small. The reason for this is that many of the nodes have balanced loads. The limiting factor for the load balancing algorithm is the few nodes with large work.

We next optimize the alias method to reduce contention. We would like nodes to send data to nearby nodes. We used a heuristic to accomplish this. We obtained the mapping of node IDs to x, y, and z coordinates on the 3-D torus. We also found a space-filling Hilbert curve that traverses these nodes. (A space-filling curve tries to order nodes so that nearby nodes are close by on the curve.) At run time, we obtain the node IDs, and create a new communicator that ranks the nodes according to their relative position on the space-filling curve. We next changed the partitioning algorithm so that it preserves the order of the space-filling curve in each partition. We also made slight changes to the alias algorithm so that it tries to match nodes based on

[4]Personal communication from James Buchanan, OLCF, ORNL.

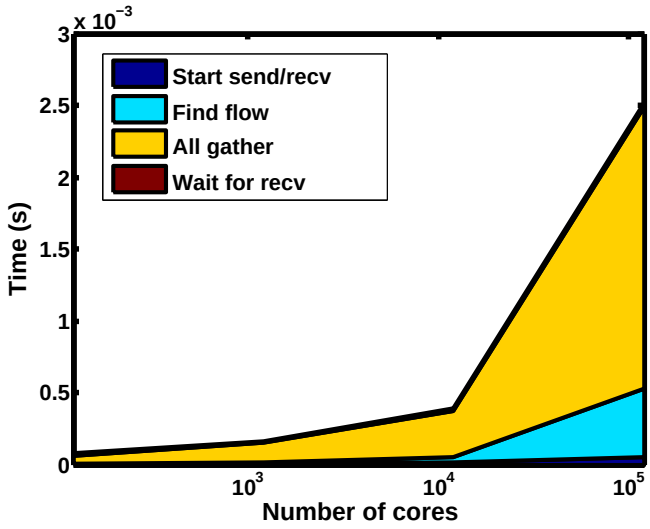

Figure 2: Mean time taken for different components of the basic Alias method with task size 8KB.

their order in the space filling curve. The creation of a new communicator is performed only once, and the last two steps don't have any significant impact on the time taken by the alias method. Thus, the improved algorithm is no slower than the basic algorithm. Figure 3 shows that the optimized algorithm has much better performance than the basic algorithm for large core counts. It is close to 30% better with 120,000 cores, and 15-20% better with 1,200 and 12,000 cores.

We next analyze the reason for the improved performance. Figure 4 considers the average over all runs for the maximum time taken by different components of the algorithm. As with the analysis of the basic algorithm, the total maximum time is smaller than the sum of the maximum times of each component. We can see that the wait time is smaller than that of the basic algorithm shown in fig. 1, which was the purpose of this optimization. The improvement is around 60% with 120,000 cores and 20% on 12,000 core. Surprisingly, the MPI_Allgather time also reduces by around 30% on 120,000 cores and 20% on 12,000 cores. It appears that the MPI implementation does not optimize for the topology of the nodes that are actually allocated for a run, and instead uses process ranks. The ranks specified by this algorithm happens to be good for the MPI_Allgather algorithm. This improvement

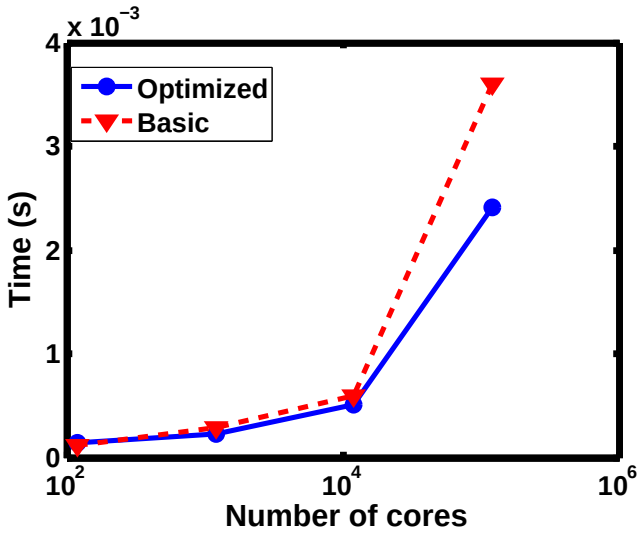

Figure 3: Comparison of optimized Alias method against the basic method with task size 8KB.

depends on the nodes allocated. In the above experiment, with 120,000 cores, the set of allocated nodes consisted of six connected components. In a different run, we obtained one single connected component. The use of MPI_Allgather with the optimized algorithm did not provide any benefit in that case. It is possible that the MPI implementation optimized its communication routines under the assumption of a single large piece of the torus. When this assumption is not satisfied, perhaps its performance is not that good.

The performance gains are smaller with smaller core counts, which can be explained by the following observations. Figure 5 shows the node allocation for the 12,000 core run. We can see that we get a large number of connected components. Thus, inter-job contention can play an important role. Each component is also not shaped close to a cube. Instead, we have several lines and 2-D planes, long in the z direction. This makes it hard to avoid intra-job contention, because each node is effectively using fewer links, making contention for links more likely. It is perhaps worthwhile to consider improvements to the node allocation policy. For 120 and 1,200 cores, typically each connected component is a line (or a ring, due to the wrap-around connections), which would lead to contention if there were several messages

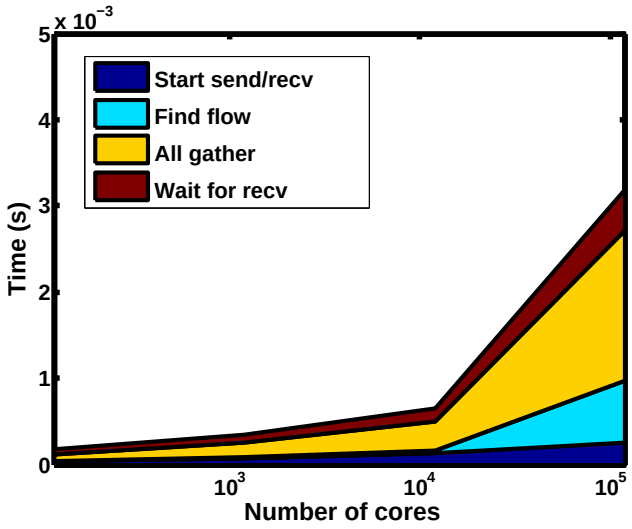

Figure 4: Maximum time taken for different components of the Optimized Alias method with task size 8KB.

sent. However, the number of nodes with imbalance is very small, and contention does not appear to affect performance in the load migration phase. Consequently, improvement in performance is limited to that obtained from the all-gather operation.

We next compare the optimized alias implementation against the QWalk implementation in Fig. 6, Fig. 7, and Fig. 8. The new algorithm improves the performance by up to 30-35% in some cases, and is typically much better for large numbers of cores. The improved performance is often due to

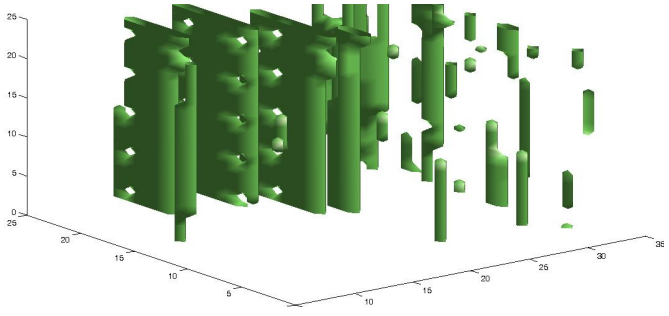

Figure 5: Allocation of nodes on the grid for a run with 12,000 cores.

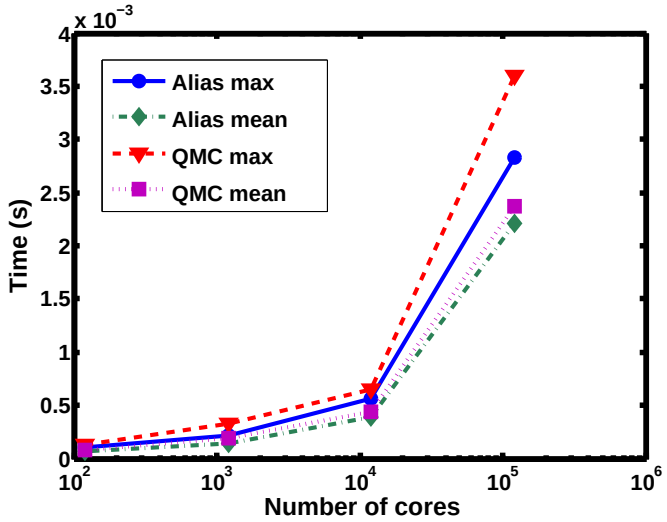

Figure 6: Comparison of the optimized Alias algorithm against the existing QWalk implementation with task size 672B.

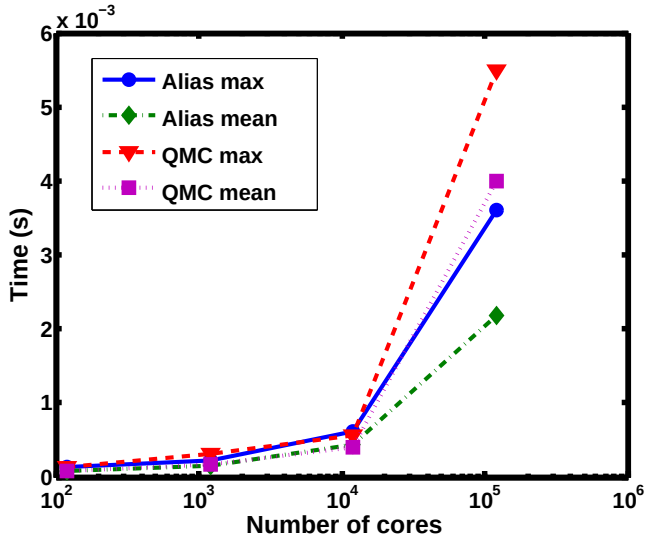

Figure 7: Comparison of the optimized Alias algorithm against the existing QWalk implementation with task size 2KB.

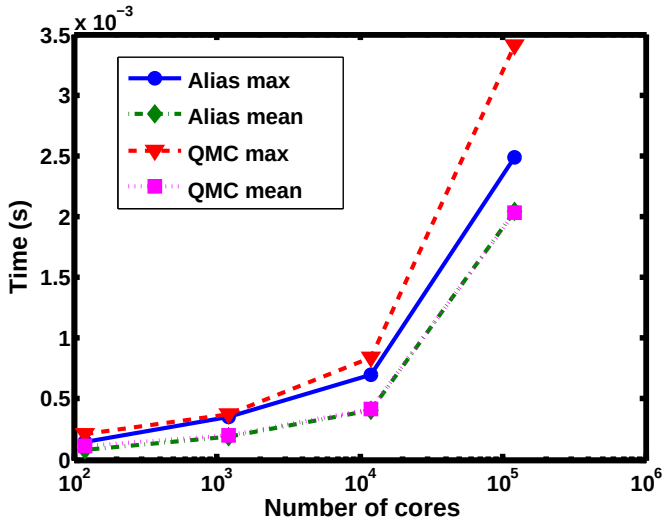

Figure 8: Comparison of Alias algorithm with the existing QWalk implementation with task size 32KB.

improvement in different components of the algorithm and its implementation: all-gather, task migration communication cost, and to a smaller extent, time for the flow computation. We can see from these figures that the time for 2KB tasks is higher on 120,000 cores than that for larger messages, especially with the QWalk algorithm. This was a consistent trend across the runs with QWalk. The higher time with the Alias method is primarily the result of a couple of runs taking much larger time than the others. These could, perhaps, be due to inter-job contention. We did not ignore this data as an outlier, because if such a phenomenon occurs 20% of the time, then we believe that we need to consider it a reality of the computations in realistic conditions.

We know that the alias method is optimal in the maximum number of messages received by any node, and find (not shown) that QWalk requires an increasing maximum number of receives with increasing core count. However, when we measure the mean number of tasks sent per core, Fig.9, we find that QWalk is optimal. The alias method is approximately a factor of two worse in terms of the number of messages sent. Although we do not see this in tests with realistic message sizes, for sufficiently large messages it is clear that there must be a cross-over in the preferred load balancing algorithm. At some point

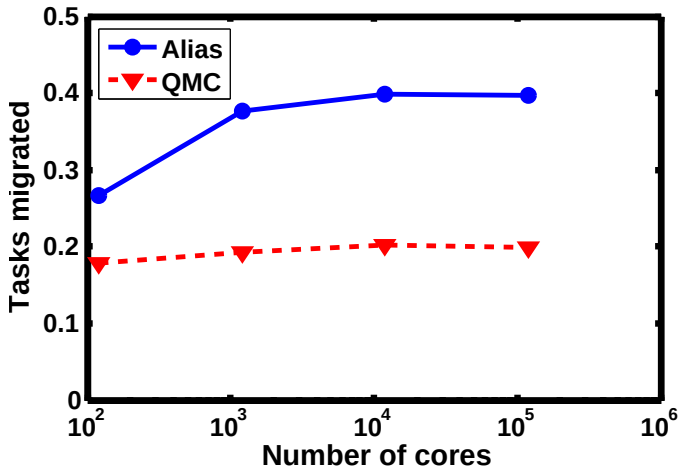

Figure 9: Mean number of tasks sent per core.

the existing QWalk algorithm will be preferred since the communications will be bandwidth bound.

6. Conclusions

We have proposed a new dynamic load balancing algorithm for computations with independent identical tasks, which has some good theoretical properties. We have shown that it performs better than the current methods used in Quantum Monte Carlo codes in empirical tests. We have also optimized the implementation and demonstrated that it has better performance due to reduced network contention. The relative performance of the algorithm to existing methods is expected to increase with the increased compute element count of upcoming machines.

Our future work will be along two directions: (i) developing better algorithms and (ii) improving performance of the current implementation. We note that our algorithm is optimal in terms of the maximum number of messages received by a process and also in terms of the maximum size of messages received. However, it is not optimized in terms of the total sizes of the messages on the network. This limitation can reduce performance by increasing the likelihood of network congestion. However, the simplicity of the messaging in the current algorithm and implementation demonstra-

bly allows for communications network contention to be reduced, by using a ranking based on a space filling curve or based on specific knowledge of the hardware layout. In future, we expect the simplicity of the messaging will allow for straightforward overlap of communication and computation, allowing much of the communications overhead to be hidden, while using very simple code. The data migration cost can also be hidden, provided that at least two walkers are used per compute element and that the transfer/communications time remains less than the computation cost of a single walker. Finally, we note that the flow computation can itself be parallelized, a step that will be required as machines with millions of compute elements become available.

Acknowledgments

We acknowledge the ORAU/ORNL HPC program for partial funding, and the INCITE program for computing time. Research by PRCK was conducted at the Center for Nanophase Materials Sciences, which is sponsored at Oak Ridge National Laboratory by the Scientific User Facilities Division, U.S. Department of Energy.

References

1. Lester WA, Reynolds P, Hammond BL. Monte Carlo methods in ab initio quantum chemistry. Singapore: World Scientific; 1994. ISBN 9789810203214.

2. Foulkes WMC, Mitas L, Needs RJ, Rajagopal G. Quantum Monte Carlo simulations of solids. *Reviews of Modern Physics* 2001;73(1):33.

3. Luchow A. Quantum Monte Carlo methods. *Wiley Interdisciplinary Reviews: Computational Molecular Science* 2011;1(3):388–402.

4. Martin RM. Electronic Structure: Basic Theory and Practical Methods. Cambridge University Press; 2004. ISBN 0521782856.

5. Needs RJ, Towler MD, Drummond ND, Rios PL. Continuum variational and diffusion quantum Monte Carlo calculations. *Journal of Physics: Condensed Matter* 2010;22:023201.

6. CHAMP . http://pages.physics.cornell.edu/ cyrus/champ.html. 2011.

7. QMCPACK . http://qmcpack.cmscc.org/. 2011.

8. Wagner LK, Bajdich M, Mitas L. Qwalk: A quantum Monte Carlo program for electronic structure. *Journal of Computational Physics* 2009;228:3390–3404.

9. Nemec N. Diffusion Monte Carlo: Exponential scaling of computational cost for large systems. *Physical Review B* 2010;81(3):035119.

10. Gillan MJ, Towler MD, Alfe D. Petascale computing open new vistas for quantum Monte Carlo. In: *Psi-K Newsletter*; vol. 103. 2011: 32.

11. Hu YF, Blake RJ, Emerson DR. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice and Experience* 1998;10:467–483.

12. Pothen A, Simon HD, Liou K. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications* 1990;11:430–452.

13. Barnard ST, Simon HD. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience* 1994;6:101–107.

14. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 1998;20:359–92.

15. Karypis G, Kumar V. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing* 1998;48:96–129.

16. Karypis G, Kumar V. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing* 1998;48:71–95.

17. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. Tech. Rep. 95-035; University of Minnesota; 1995.

18. Oliker L, Biswas R. PLUM: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing* 1998;51:150–177.

19. Cybenko G. Journal of parallel and distributed computing. *Dynamic load balancing for distributed memory multiprocessors* 1989;7:279–301.

20. Diekmann R, Frommer A, Monien B. Efficient schemes for nearest neighbor load balancing. *Parallel Computing* 1999;25(7):789–812.

21. Hu YF, Blake RJ. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing* 1999;25:417–444.

22. Ghosh B, Muthukrishnan S, Schultz MH. First and second order diffusive methods for rapid, coarse, distributed load balancing. *Theory of Computing Systems* 1998;31:331–354.

23. Elsasser R, Monien B, Schamberger S. Distributing unit size workload packages in heterogeneous networks. *Journal of Graph Algorithms and Applications* 2006;10:51–68.

24. Schloegel K, Karypis G, Kumar V. A unified algorithm for load-balancing adaptive scientific simulations. In: *Proceedings of the IEEE/ACM SC2000 Conference*. IEEE Computer Society; 2000:.

25. Catalyurek U, Boman E, Devine K. A repartitioning hypergraph model for dynamic load balancing. *Journal of Parallel and Distributed Computing* 2009;69:711–724.

26. Walshaw C, Cross M. Dynamic mesh partitioning and load-balancing for parallel computational mechanics codes. In: Topping BHV, ed. *Computational Mechanics Using High Performance Computing*. Edinburgh: Saxe-Coburg Publications; 1999:.

27. Li X, Parashar M. Hierarchical partitioning techniques for structured adaptive mesh refinement applications. *The Journal of Supercomputing* 2003;28:278.

28. Qin X. Performance comparison of load balancing algorithms for i/o-intensive workloads on clusters. *Journal of Network and Computer Applications* 2008;31:32–46.

29. Iosup A, Sonmez O, Anoep S, Epema D. The performance of bags-of-tasks in large-scale distributed systems. In: *Proceedings of the HPDC*. 2008:.

30. Fujimoto N, Hagihara K. Near-optimal dynamic task scheduling of independent coarse-grained tasks onto a computational grid. In: *Proceedings of the International Conference on Parallel Processing*. 2003:391–398.

31. Maheswaran M, Ali S, Siegal H, Hensgen D, Freund R. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. In: *Proceedings of the Heterogeneous Computing Workshop*. 1999:30–44.

32. Dhakal S, Hayat M, Pezoa J, Yang C, Bader D. Dynamic load balancing in distributed systems in the presence of delays: A regeneration-theory approach. *IEEE Transactions on Parallel and Distributed Systems* 2007;18:485–497.

33. Walker AJ. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software* 1977;3:253–256.

34. Kronmal RA, Peterson AV. On the alias method for generating random variables from a discrete distribution. *The American Statistician* 1979;33:214–218.