Scalable Work Stealing*

James Dinan, D. Brian Larkins, P. Sadayappan
Dept. Computer Science and Engineering
The Ohio State University
Columbus, OH 43221
dinan, larkins, saday@cse.ohio-state.edu

Sriram Krishnamoorthy, Jarek Nieplocha
Pacific Northwest National Laboratory
Richland, WA 99352
sriram@pnl.gov

ABSTRACT

Irregular and dynamic parallel applications pose significant challenges to achieving scalable performance on large-scale multicore clusters. These applications often require ongoing, dynamic load balancing in order to maintain efficiency. Scalable dynamic load balancing on large clusters is a challenging problem which can be addressed with distributed dynamic load balancing systems. Work stealing is a popular approach to distributed dynamic load balancing; however its performance on large-scale clusters is not well understood. Prior work on work stealing has largely focused on shared memory machines. In this work we investigate the design and scalability of work stealing on modern distributed memory systems. We demonstrate high efficiency and low overhead when scaling to 8,192 processors for three benchmark codes: a producer-consumer benchmark, the unbalanced tree search benchmark, and a multiresolution analysis kernel.

Keywords

Dynamic Load Balancing, Work Stealing, Task Pools, PGAS, ARMCI, Global Arrays

1. INTRODUCTION

Many applications exhibit irregularity in the dynamic generation of units of parallel computation, especially with nested or recursive parallelism. Irregularity often arises due to sparsity present in the data. For example, in scientific simulations spatial sparsity of the system often translates into sparsity in the numerical model [38]. Recursive spatial decomposition is a convenient and natural means for expressing many such algorithms. However, recursive codes often dynamically expose parallelism by splitting spatial units into smaller tasks requiring a system to support dynamic scheduling of evolving parallelism. For these schemes, the quantity and slackness of available parallelism may have complex dependencies on input data, further confounding static scheduling approaches.

Randomized work stealing is a distributed dynamic load balancing scheme, popularized by the runtime system for the Cilk [16] parallel programming language. Under work stealing, idle processors select victim processors at random and attempt to steal work from them. Work stealing has been proven to be optimal for a large class of problems and has tight space bounds [5, 23]. However, the scalability of work stealing has not been well explored on modern large-scale systems. In particular, concerns exist that the randomized nature of work stealing can lead to long idle times and poor scalability on large-scale clusters.

The introduction of RDMA, or Remote Direct Memory Access, high performance interconnect networks has facilitated efficient implementations of Partitioned Global Address Space (PGAS) parallel programming models [8, 9, 30, 34, 36, 39]. PGAS models provide the programmer with a convenient, partitioned global view of memory across all nodes in the computation, with efficient one-sided access to data on remote nodes. This one-sided remote access model is a natural implementation vehicle for implementing work stealing's one-sided steal operations. Such programming models both support and motivate our work, as discussed in Section 6.

In this work, we investigate the design of a scalable runtime system to support dynamic load balancing through work stealing using the PGAS programming model provided by the Aggregate Remote Memory Copy Interface (ARMCI) [29]. We propose techniques to reduce locking on the critical path and reduce contention; schemes to reduce the overhead of task creation; a work splitting scheme that enhances scalability by maximizing the availability of work; and a customized locking scheme that allows for early aborting to avoid waiting on stale resources. We observe that while the techniques employed to achieve scalability turned out to be simple, the process of identifying the problems and the solution was non-trivial. In particular, the issues identified and addressed were amplified due to the scale of the systems we were targeting, and have not been reported or tackled in the literature.

We evaluate the efficiency and scalability of our system on a modern multicore cluster equipped with an rDMA interconnect that supports efficient execution of PGAS programs. The implementation was evaluated on three bench-

*This work was supported by the U.S. Department of Energy through awards #DE-FC02-06ER25755 and #DE-AC05-00OR22725 and the National Science Foundation through award #0403342. We thank the Molecular Sciences Computing Facility (MSCF) at the Pacific Northwest National Laboratory (PNNL) for the use of their computing facilities.

SC09 November 14-20, 2009, Portland, Oregon, USA.

mark codes that required effective dynamic load balancing to scale to large process counts. Using our system, we demonstrate the scalability of randomized work-stealing on up to 8192 processors. We are not aware any demonstrations of effectiveness of work stealing at this scale on a distributed memory machine.

The rest of the paper is organized as follows. In Section 2 we present an overview of the task parallel programming model and dynamic load balancing problem; in Sections 3 and Section 4 we explore design issues and optimizations to reduce overhead and increase efficiency; and in Section 5 we present an experimental evaluation. Related work is discussed in Section 6 and Section 7 concludes the paper.

2. DYNAMIC LOAD BALANCING MODEL

In this section we discuss the dynamic load balancing and task execution model that we consider in this work. The dynamic load balancing work in the literature can be expressed and understood through the use of *task pools*. Task pools provide a convenient abstraction that allows the programmer to express a parallel computation as a set of dynamic *tasks*. A task is the basic unit of work and each task is executed in the context of a process participating in the parallel computation. In the task pool programming model, the programmer first *seeds* the task pool with an initial set of tasks and then *processes* the task pool in parallel as shown in Algorithm 1.

Algorithm 1 Task pool execution model

```
{ Let T_p be a distributed task pool }
T_p ← {initial_tasks...}
while t ← next_task(T_p) do
  execute_task(t)
end while
```

Each task is represented by its *task descriptor* which provides the task's arguments, including references to locations in the global address space where the task will fetch its inputs and store its results. Tasks execute with respect to data stored in the partitioned global address space, making them portable and enabling them to be executed on any process in the computation. Tasks may also create new subtasks and enter them into the task pool. This allows for the implementation of recursive parallelism and allows the programmer to capture parent-child data dependencies. Tasks are processed from the task pool in LIFO order (Last In First Out), yielding a depth-first traversal of the task tree and bounding the space requirements of the task pool as proportional to $O(T_{depth})$.

We restrict our task execution model by requiring that all tasks enqueued in the task pool are independent. Parent-child dependencies can be expressed through dynamic task creation. However, once enqueued, tasks must be able to execute to completion without blocking. Tasks are allowed to communicate through data stored in the global address space, however tasks must not wait for results produced by any concurrently executing tasks. These restrictions allow us to relax fairness constraints on task scheduling and make it possible to avoid the need for migrating partially executed tasks to balance the load. This model captures the execution style of many significant applications as demonstrated in Section 5 and allows us to optimize for their efficient execution.

2.1 Distributed Task Pools

Task pools can be implemented using either a centralized or distributed approach. Centralized task pools utilize a work server that provides a central location for storing the set of available tasks. This scheme is convenient for small systems, however it provides poor scaling to larger machines. Distributed task pools store patches of the task pool over a set of work queues that are distributed across the processes in the computation. The ratio of work queues to processors can be varied, allowing for a variety schemes with different properties.

In this work we focus on a $1:1$ scheme where each process maintains its own work queue, which allows for efficient local access. Under this scheme, all processes perform work with respect to their own task queue, pushing new tasks on the head of the queue and popping tasks from the head to get the next task to execute in LIFO order.

2.2 PGAS Task Pools

Partitioned Global Address Space (PGAS) programming models provide new opportunities for the efficient and scalable implementation of task pools on distributed memory systems. These programming models provide a global view of physically distributed data along with mechanisms for performing efficient, one-sided access. The ability to perform one-sided access to remote data is especially useful for the implementation of distributed task pools where load balancing requests are generated asynchronously in response to local work conditions. By storing distributed task queues in the global address space, we gain the ability to perform one-sided work stealing where, in spite of distributed memory, steal operations can proceed without interrupting a working remote process.

For our implementation, we focus on the PGAS programming model provided by ARMCI, the Aggregate Remote Memory Copy Interface [29]. ARMCI gives the benefit of interoperability with multiple parallel programming models including MPI [28], the industry standard message passing interface, and the Global Arrays toolkit [30] which provides a PGAS model for distributed shared multidimensional arrays.

ARMCI is a portable, low level PGAS library that allows the programmer to allocate regions of memory that are available for remote access using one-sided *get* and *put* operations. Under this model, computations take on the *get-compute-put* model where data must be first fetched into a local buffer before processing and then copied back into a remote location. In addition, ARMCI provides portable support for a number of one-sided atomic operations aiding in the design of distributed data structures that support direct remote access.

2.3 Work Stealing Algorithm

Under work stealing, each process maintains a double-ended work queue, or *deque*, of tasks. Processes execute tasks from the head of their deque and when no work is available they steal tasks from the tail of another process' deque. The process that initiates the *steal* operation is referred to as the *thief* and the process that is targeted by the steal is referred to as the *victim*. Because the thief is responsible for initiating load balancing requests, work stealing is

a *receiver-initiated* load balancing algorithm and the total volume of load balancing operations performed will therefore be proportional to the load imbalance. In addition, given an appropriate work division scheme, it has been proven that the load imbalance under workstealing is bounded making it a *stable* load balancing algorithm [27, 3].

A PGAS version of the work stealing algorithm is given in Algorithm 2. When performing a steal operation, the thief must first select its victim. Many schemes are possible, however random victim selection has been proven to be optimal [5]. Once a victim has been selected, the thief must then fetch the metadata for the victim's work queue to determine if they have work available. If they do, the thief locks the victim's queue and checks the metadata again to ensure the work is still available. If so, it transfers one or more tasks from the tail of the victim's queue to its own queue. If the victim has no work available, the thief selects a new victim at random and repeats this process until either work is found or global termination is detected.

Algorithm 2 Work stealing algorithm in the *get-compute-put* PGAS style.

```
while ¬have_work() ∧ ¬terminated do
  v ← select_victim()
  m ← get(v.metadata)
  if work_available(m) then
    lock(v)
    m ← get(v.metadata)
    if work_available(m) then
      w ← reserve_work(m)
      m ← m − w
      put(m, v.metadata)
      queue ← get(w, v.queue)
    end if
    unlock(v)
  end if
end while
```

2.4 Termination Detection

In order to determine when the computation has completed, processes must actively detect when all processes are idle and no more work is available. The process of detecting this stable global state is referred to as *termination detection*. Many schemes are possible, ranging from centralized schemes using shared counters and termination detection servers to fully distributed schemes.

In this work, we utilize a distributed voting-tree scheme where a binary tree is mapped onto the process space similar to the scheme originally proposed in [15]. When a process becomes idle, it combines the votes from its two children with its own vote and passes the vote to its parent in the tree. If a process has been the victim of a steal operation it must call for a re-vote by passing a negative vote up the tree. Once the vote reaches the root, the root process broadcasts the result back down the tree. If any processes submitted a negative vote, a new round of voting is started. Otherwise the root broadcasts a message stating that termination has been detected. Because of its tree structure, this termination detection algorithm detects termination in $O(logN)$ time where N is the number of processes.

2.5 Benchmark Applications

For this study, we have selected three benchmarks to evaluate our runtime system for work stealing: a multiresolution analysis kernel targeted at high performance computational chemistry codes, an unbalanced tree search benchmark, and a producer-consumer benchmark. Our experiments were conducted on a 2,310 node cluster with two quad-core processors per node and Infiniband interconnect.

2.5.1 MADNESS Tree Creation Kernel

The MADNESS (Multiresolution ADaptive NumErical Scientific Simulation) project is a collaborative effort to develop a framework for scientific simulation using adaptive multiresolution analysis methods in multiwavelet bases [38]. The first step in every MADNESS execution is to *project* a user-supplied analytic function into its numerical representation through adaptive spatial decomposition. For our kernel we consider functions in 3-dimensions, resulting in an oct-tree based numerical representation of the analytic function.

Projection begins with a fixed 3d volume over which the analytic function is sampled to derive the numeric representation. The accuracy of the representation is then examined and if it is not high enough, the 3d space is split into 8 subspaces and each subspace is recursively processed until the desired accuracy has been reached. The size and shape of the resulting task tree is highly dependent on the user-supplied analytic function making MADNESS a good candidate for dynamic load balancing.

2.5.2 The Unbalanced Tree Search Benchmark

The Unbalanced Tree Search benchmark (UTS) models the parallel performance of state space exploration and combinatorial search problems by measuring the performance achieved when performing an exhaustive parallel depth-first search on a parametrized, unbalanced tree [13, 14, 31, 32]. The tree is constructed using a splittable, deterministic random stream generated using the SHA-1 secure hash algorithm. Each node is represented by a 20-byte SHA-1 digest and its children are found by applying the SHA-1 algorithm to the parent node's digest combined with the child id. There is a high degree of variation in the size of each subtree rooted at any given node in a UTS tree. Thus, if each node is taken as a task in a UTS execution there is a high degree of variation in the amount of work contained within each task. These properties make UTS a challenging problem that is effective for evaluating dynamic load balancing schemes.

2.5.3 Bouncing Producer-Consumer Benchmark

The Bouncing Producer-Consumer (BPC) benchmark is a producer-consumer benchmark with a twist: the producer process may change as the result of a steal operation. This benchmark is intended to create a scenario where locating work is challenging due to the migrating producer.

BPC is a task-parallel benchmark that dynamically produces two kinds of tasks: producer tasks and consumer tasks. In the example task tree shown in Figure 1 producer tasks are shown in black and consumer tasks are shown in white. Each producer task produces one producer task followed by n consumer tasks. The corresponding dequeue has the producer task at its tail making it the first task that will be stolen and the process that steals it will become the new

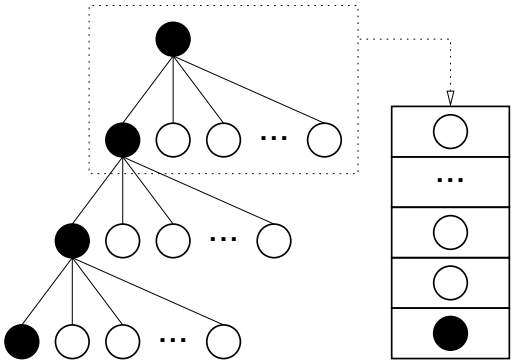

Figure 1: Bouncing producer consumer example tree and corresponding task queue. The next producer node resides at the tail of the queue, available to be stolen.

producer. This process is repeated until a maximum depth d is reached.

3. RUNTIME INFRASTRUCTURE

In this section we present the design of our work stealing runtime system. We propose and explore the performance impact of split task queues that eliminate locking on the local process' critical path and reduce contention; schemes to reduce the overhead of task creation; a work splitting scheme that enhances scalability by maximizing the availability of work; and we discuss the impact of data server responsiveness on scalability.

3.1 Split Task Queues

It is essential that the task queue provide efficient local access, as enqueue and dequeue operations are executed on the critical path. A fully shared task queue would require expensive locking for every access to the task queue. This locking can be eliminated by using two queues per process: one for storing a portion of the work reserved by the local process and a second queue to store shared work. However, whenever work is moved between the local and shared queues a memory copy must be performed. This memory copy can be eliminated by using a single task queue that is split into private and public portions as described in our prior work [12].

The split task queue is shown in Figure 2. This queue splits a single shared queue into a local access only portion and a shared portion. The local portion is located between the `head` and `split` pointers and the shared portion is located between the `tail` and `split` pointers. The local portion of the queue can be accessed by the local process without locking and the shared portion can be accessed by any process and accesses are synchronized via a lock. The local process only needs to take the lock when adjusting the distribution of work between the public and private portions of the queue.

When using the split queue, the local process must periodically check to ensure that it is exposing enough surplus work in the public portion of its queue by moving the split toward the head of the queue. We refer to the operation of advancing the split toward the head of the queue as a *release*

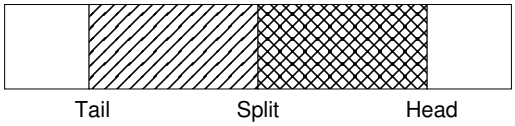

Figure 2: Split Queue: Allows lock-free local access to the private portion of the queue and provides zero-copy transfers between public and private portions of the queue.

operation. Releases must be performed periodically to ensure that enough work is available for idle processes to steal. Likewise, when the local process has exhausted the work in the private portion of its queue, it performs a *reacquire operation* to move the split toward the tail of the queue and reclaim public work into the private portion of the queue.

3.2 Lockless Release Operations

We have further improved on the split queue presented in [12] by reorganizing the queue metadata. In our prior work we maintained the queue using the metadata `{tail, npublic, split, head, nprivate}`, where `npublic` and `nprivate` maintain the number of tasks in the public and private portions of the queue, respectively. Under this scheme, modifications to `tail`, `npublic` and `split` (due to its dependence on `npublic`) had to be performed under lock. Thus, the queue must be locked when performing both release and reacquire operations.

If we further distill the queue metadata to `{tail, split, nlocal}` then we can remove the need to lock when performing release operations. Using this new metadata, `npublic` and the `head` pointer are calculated on demand through simple arithmetic rather than stored. Using this new scheme, only updates to the `tail` pointer need to be synchronized, allowing data to be moved from the private portion of the queue to the public portion of the queue without locking. Because a release operation modifies the `split` and `nlocal` metadata and a steal operation modifies only the `tail` metadata, this scheme makes it possible to support a release operation concurrently with a steal operation. If the process performing the steal operation sees the old value for `split`, it simply appears that less work is available than is actually present in the public portion of the queue, which does not impact correctness. Thus, this metadata scheme enables *lockless release operations* to be performed.

Figure 3 compares the performance of the split queue with lockless release with the performance of a shared queue that requires locking for all accesses to the queue. In this experiment we ran the bouncing producer consumer benchmark with producers that produce $n = 64$ tasks that each perform 1 msec of work and a total depth $d = 32{,}768$. At low processor counts, the parallel slack is relatively high and contention remains low. However, as the processor count is increased, producers have to contend with thieves to acquire the lock for their queue and enqueue new tasks. Performance begins to degrade past 128 processors as the time required to enqueue new tasks grows due to lock contention.

3.3 Efficient Task Creation

The overhead of task creation is an important factor affecting task pool performance as many applications perform

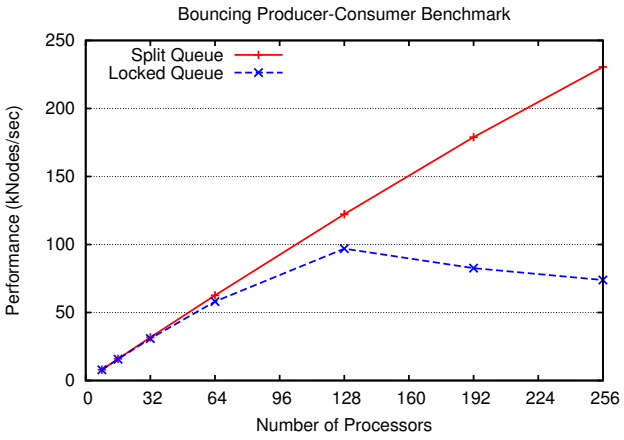

Figure 3: Performance of split vs locked task queues for the BPC benchmark with $n = 64$ and $d = 32768$.

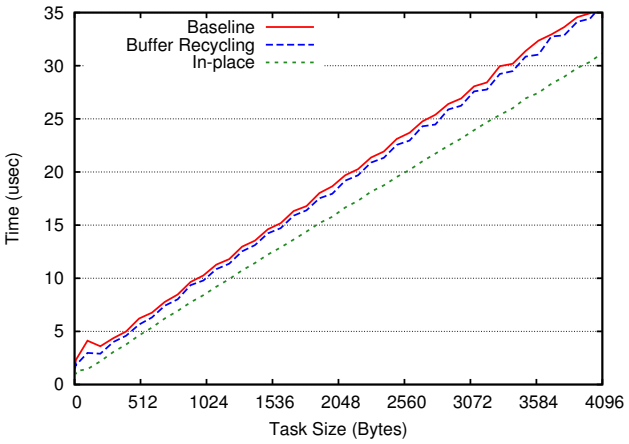

Figure 4: Task creation times for baseline, buffer recycling, and in-place task creation schemes.

dynamic task creation on their critical path. As shown in Table 1, the UTS benchmark has a very short average task length but generates very large numbers of tasks. For applications that exhibit such characteristics, the efficiency of task creation can have a significant impact on performance.

The process of task creation involves three steps:

1. Create a task descriptor
2. Populate the descriptor with task arguments
3. Enqueue the task

In step 1, a temporary buffer is allocated to hold the task descriptor. In step 2, data is copied into the task descriptor, and in step 3 the data is copied from the user's buffer into the task queue.

Since task descriptor buffers are generally short lived, a simple optimization is to recycle buffers to eliminate calls to the memory allocator in step 1. Furthermore, if the programmer knows that the head of the queue does not need to be accessed while the task is being created, further optimization is possible by allocating the new task descriptor directly from the head of the queue. We call this optimization *in-place task creation* and it eliminates both the memory allocation in step 1 and the copy operation in step 3.

We compare the performance of these three schemes in Figure 4. Adding buffer recycling provides a significant benefit over the baseline approach for very small tasks. However, for larger tasks the cost of allocation is amortized by the cost of performing the copy operations. In-place task creation provides roughly a 15% performance improvement, which may seem smaller than expected, as it eliminates both the memory allocation and a copy. The reason that the gain from eliminating the second copy operation is not larger is that the data copy performed in step 3 is less expensive than the data copy performed in step 2, due to cache effects - the copy in step 2 brings the data into cache and the copy in step 3 is performed with the input data already in cache.

3.4 Work Splitting

When performing a steal operation, a thief must determine how much work to steal. In the general work stealing algorithm, the thief attempts to balance the load evenly between itself and the victim [3, 23, 27]. Work stealing load balancers that focus on recursive parallelism, such as Cilk's runtime system, take the approach of stealing the task from the tail of the victim's queue. These systems reason that since the task stolen was at the tail of the queue it is likely to be of the largest granularity and roughly half of the victim's work [5]. For cilk-style applications this approach yields good performance, however for non-strict computations this strategy can yield an unstable load imbalance [3].

Many applications of interest do not follow a recursive parallel style or yield tasks where the amount of work present in each task varies widely. In our system, we wish to apply a strategy that will apply to a broad class of computations so we must not rely on a particular structure or task granularity. In addition we must satisfy the stability constraint given by Berenbrink et al in [3]. This constraint states that given a victim with l work and a work splitting function f:

$$f = \omega(1) \tag{1}$$

and

$$0 \leq f(l) \leq l/2 \tag{2}$$

In our system we use the work splitting function, f, that selects half of the tasks on the victim's public queue for stealing. This approach satisfies the stability constraint for arbitrary l and also allows us to maximize the number of work sources in the computation. In the common case, this strategy leaves the victim with half of the work still available in its public queue and provides the thief with enough work so that it too can put surplus tasks in its public queue. By maximizing the number of work sources, we aim to enhance scalability by decreasing the average time required by any process to locate and steal work.

We evaluate the performance of this scheme in Figure 5 in the context of the Unbalanced Tree Search benchmark. In this figure, we present the performance of UTS performing parallel exploration of a highly unbalanced tree for three schemes: steal-half, steal-1, and steal-2. UTS generates tasks recursively and thus has a workload where the steal-

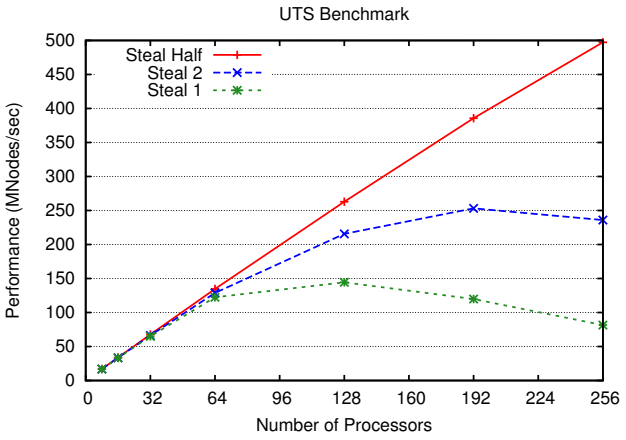

Figure 5: Impact of steal-half over steal-1 (Cilk-style) and steal-2.

n strategy is stable. We see that all schemes scale well to 64 processors, however as the processor count is increased, work becomes harder to find and the steal-1 scheme loses performance as most processors are idle searching for work. Stealing two tasks during each steal operation helps to better distribute the work, but it still does not scale past 128 processors. Performing steal-half gives the best performance, yielding over 95% efficiency on 256 processors. For this experiment, steal-half transferred an overall average of 3.8 tasks per load balancing operation.

3.5 ARMCI Runtime System

On many platforms, ARMCI's runtime system spawns one data server thread per SMP node to efficiently support primitives such as accumulate and strided/vector RDMA, and to guarantee progress for one-sided operations. Additionally, the data server has the job of ensuring portable support for the variety of atomic operations provided by ARMCI. The default behavior of the data server thread is to block on an incoming message and yield its processor to the application. However, we found that this mode of operation introduced long latencies for the atomic operations required to implement work stealing. In this mode, when thieves perform a steal they must not only wait for the operating system to schedule the data server thread, but also for the data server to receive a signal from the networking layer indicating that a message has arrived. These combined overheads often caused an operation that should take tens of microseconds to take tens of milliseconds or more, depending on how heavily loaded the remote node was. Due to these overheads and because the data server is responsible for servicing requests for the entire SMP node, we found that with an unpinned data server even small amounts of contention could result in very long wait times.

In order to eliminate these overheads, ARMCI provides an alternative mode of operation where the data server thread remains bound to a core and actively polls the network for incoming requests. This eliminates latencies due to OS scheduler noise and blocking on a signal from the networking layer. On our test system which is configured with eight cores per node, we found that dedicating a core to the data server is essential to lowering the latency of steal operations, tolerating contention, and achieving scalability.

4. SCALABLE WORK DISCOVERY

The performance and scalability of work stealing depends heavily on the cost of ensuring mutually exclusive access to the shared portion of each process' queue and the performance of this mechanism under contention.

4.1 ARMCI Locks

ARMCI provides a locking mechanism that uses a simplified version of the bakery (aka deli counter) algorithm for mutual exclusion [25]. It uses two shared global counters: *next_ticket* and *current_ticket*. The locking mechanism is analogous to the ticket scheme used in many bakeries: new customers take a ticket when they arrive in the bakery and they are waited on when the number on their ticket comes up.

Under this scheme, when a process executes a lock operation, it first performs an atomic fetch-and-increment operation on *next_ticket* and stores the value fetched in t. It then enters a a loop where it waits until $current\_ticket = t$. Once $current\_ticket = t$ the process may enter its critical section. When the process has finished executing its critical section, it increments *next_ticket*, allowing the next process in line to enter its critical section. This algorithm is considered fair because it ensures FIFO access to the resource guarded by the lock.

4.2 Spinlocks

An alternative locking scheme is to create spinlocks using ARMCI's atomic swap operation, as shown in Algorithm 3. Spinlocks require a single communication operation to obtain a lock, whereas ARMCI's locks require an atomic increment followed by a get operation. Thus, if the lock is uncontended, spinlocks may provide a more efficient locking mechanism.

In comparison with ARMCI's locks which ensure fair, FIFO access, spinlocks don't guarantee access and can lead to starvation. In addition, spinlocks spin using an atomic operation that must be implemented using ARMCI's data server, whereas ARMCI locks spin using a one-sided *get()* operation that is supported by RDMA interconnects. Hence spinlocks may also perform worse under contention.

Algorithm 3 Spinlock *lock*() operation using atomic swap

```
{ Let l_v be a lock on victim v }
oldval ← LOCKED
while oldval ≠ UNLOCKED do
  atomic_swap(oldval, l_v)
  if oldval ≠ UNLOCKED then
    { Perform linear backoff to avoid flooding the victim
    with lock traffic }
  end if
end while
```

4.3 Aborting Steals

ARMCI's fair locks offer better performance under contention and are starvation free. However, in the case of work stealing they can cause a process to wait to steal from a queue that has been emptied by processes ahead of it in

line. Because the process attempting the lock has already acquired a resource by taking a ticket, it must wait for its turn to release the resource even if it realizes that the queue it is waiting on has been emptied.

Spinlocks, on the other hand, allow the process attempting the lock to abort at any time. This provides the process the ability to abort a steal operation if it determines that the victim no longer has any work available. While waiting on the lock for a remote queue, the thief can periodically fetch the remote queue's metadata to determine if it still has work available. If no more work is available, the thief can simply abort the lock operation and select a different victim. In comparison, when using ARMCI locks, even if the thief determines that the victim has no more work available, it must continue to wait until its ticket is up so it can increment the *current_ticket* counter, signaling the next process in line that it may access the victim's queue. This requires waiting $O(T)$ steps, where T is the number of thieves in line, for each thief to release the lock.

5. EXPERIMENTAL RESULTS

We evaluated the performance of our scheme at scale using three benchmarks: MADNESS, UTS, and BPC described in Section 2.5. Experiments were conducted on a 2,310 node Hewlett Packard cluster located at Pacific Northwest National Laboratory. Each node is configured with two quad-core AMD Barcelona processors running at 2.2 GHz and Infiniband interconnect. Compute nodes run the GNU/Linux operating system and the message passing layer used was HP-MPI.

5.1 Benchmark Workloads

In this section we conduct strong scaling experiments to evaluate the scalability of our system. For these experiments, the Madness kernel performed tree creation on a 128 element soft body system; the Unbalanced Tree Search benchmark performed exhaustive search on an unbalanced 270,751,679,750 node geometric tree with depth 18; and the Bouncing Producer Consumer (BPC) benchmark was run on a problem where each producer produced $n = 8192$ 10msec tasks with a total depth $d = 4096$. The total number of tasks and average task execution time these each workload is given in Table 1.

Benchmark	Total Tasks	Avg. Task Time
Madness	1,801,353	102.28073 ms
BPC	33,558,529	9.96720 ms
UTS	270,751,679,750	0.00066 ms

Table 1: Benchmarking workload characteristics.

All three benchmarks start from a single task and dynamically generate subtasks. In the case of Madness, each task corresponds to a 3-dimensional region of space which may be subdivided to get finer precision for the numerical representation where needed. Each adaptive refinement results in 8 new subtasks. In BPC, each producer task produces 1 producer task and 4096 consumer tasks which are leafs and do not produce any subtasks. Finally, UTS was run to produce an unbalanced trees where the expected number of children per node is geometric in the depth of the tree and proportional to the branching factor, $b_0 = 4$.

5.2 Scalability Study

In Figure 6 we show strong scaling experiments for each of the schemes discussed in Section 4. For each benchmark, we show the performance achieved from 512 to 8192 cores. Performance is reported as the total throughput in terms of thousands of nodes visited per second (kNodes/sec) to billions of nodes visited per second (GNodes/sec). We also report the efficiency of each execution as the percent of the total execution time spent executing tasks. This efficiency measurement separates the total execution time into active time, or time when a user task is executing, and idle time, or time spent searching for work. With this data we wish to characterize the load balance and have not subtracted any performance lost due to overheads, such as time lost when enqueueing tasks or balancing the queues. In general these overheads are low since we have created schemes to eliminate locking from the critical path and perform in-place task creation.

The benchmark data presented in Figure 6 is organized from left to right in decreasing task granularity with comparable runs of Madness processing tens of thousands of tasks per second; BPC, hundreds of thousands of tasks per second; and UTS, billions of tasks per second. As the number of cores is increased, Madness and BPC expose parallelism more slowly leading to limited parallel slack and creating the potential for long search times and high contention. UTS, in comparison, produces new work very rapidly and has a high degree of parallel slack. However, due to imbalance in the search space processes frequently run out of work as the subtree they are exploring terminates. For this reason, UTS requires lightweight load balancing, as the need to perform frequent load balancing can lead to high overheads.

The ARMCI and spin lock schemes scale to 4096 processors, however past this point limited parallel slack leads to contention to steal from the limited number of processes that have surplus work and long waits for thieves that frequently result in no work being stolen. By comparing the performance of ARMCI's locks with spinlocks, we clearly see that for Madness and BPC ARMCI's locks offer better performance. However, for UTS which exhibits a high degree of parallel slack, there is relatively little contention and spinlocks outperform ARMCI locks because they require only a single communication operation to take a lock, whereas ARMCI locks require two communication operations.

When contention is high and parallel slack is limited, steal operations performed using the ARMCI and spin lock schemes can often result in a long wait where the victim's work dries up before all the thieves targeting it get work. In order to address this problem, we have implemented aborting steals which periodically check if the victim still has work available to avoid waiting on a stale queue. We see in Figure 6 that this scheme addresses the critical performance problem for Madness and BPC. For Madness, the aborting steals scheme achieves 88% efficiency on 8,192 cores in spite of severely limited parallel slack and for BPC it achieves 97% efficiency. For the UTS benchmark which has a high degree of parallel slack, the aborting steals scheme achieves the best performance with an efficiency of 99% on 8,192 cores. This is because it is built using spinlocks which are less expensive than ARMCI locks when the lock is uncontended.

5.3 Failed and Aborted Steals

We present the total number of failed and aborted steals

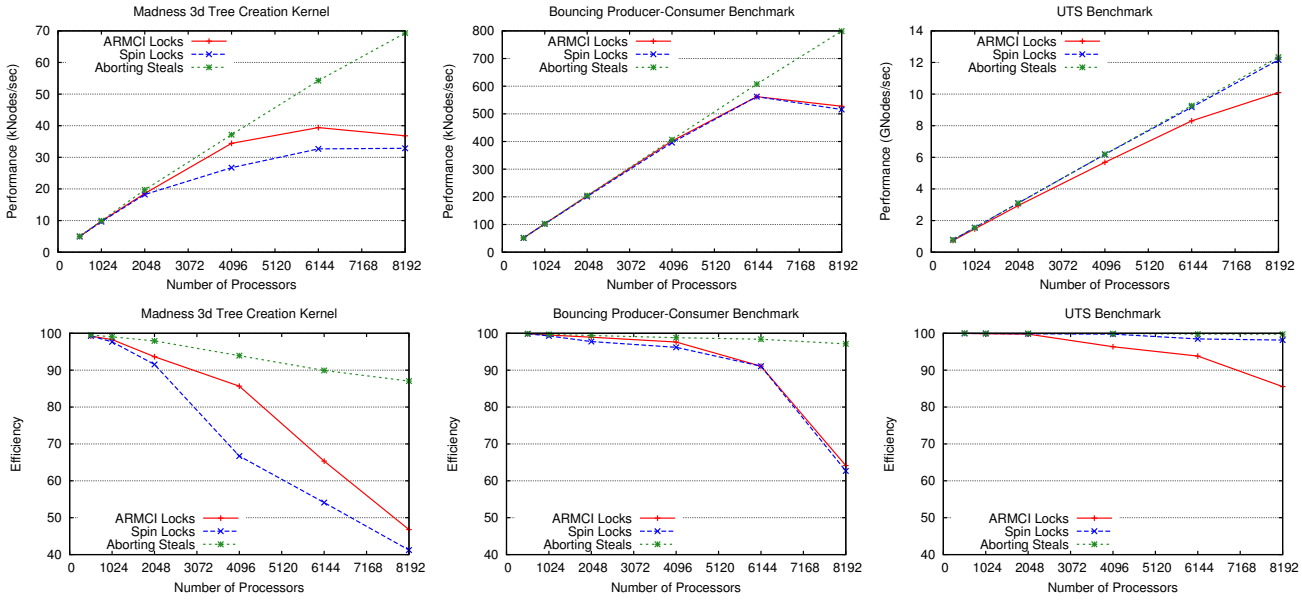

Figure 6: Performance and percent efficiency for Madness, the BPC benchmark, and UTS on 8,192 cores.

for each scheme in Figure 7. *Failed steals* are steal operations where the thief has acquired a lock on the victim, only to discover that the victim has no work available. Failed steals are an indication of the amount of contention present when accessing the shared portion of the queue. This contention can arise due to remote access by multiple processes or due to the local process performing reacquire operations concurrently with remote processes attempting to perform a steal.

Aborted steals are steal operations where the thief aborts the current victim without acquiring the lock, because it has determined that the victim has no work available. Aborted steals can occur under all schemes during victim selection when the thief initially fetches the victim's metadata to determine if it has any work available before attempting to lock the shared portion of the victim's queue. In the aborting steals scheme, this check is performed periodically even after a thief has decided to follow through with a steal on the current victim. If the thief determines that the victim no longer has work available, it can abort its current steal operation and move on to a different victim. Thus, the number of aborted steals reflects the parallel slack or the general availability of work present in the computation.

For Madness and BPC, we see that the performance loss past 4,096 processors is due to the large number of failed steals. The ARMCI and Spin locking schemes experienced between 2.5 to 3 billion failed steals compared with 0.25 billion for the Aborting steals scheme. If we look at the number of aborted steals for Madness, we see that the Aborting steals scheme performed twice as many aborts as the ARMCI locking scheme. This indicates that during execution of the Madness benchmark, idle processes experienced long searches for work due to limited parallel slack, which accounts for its 12% loss of efficiency at scale.

For BPC, we see that, surprisingly, the number of aborted steals is similar for all three schemes. This indicates that the primary cause for performance loss is due to a small number of pileups where multiple consumers were all waiting on a single producer. These pileups can be a significant performance problem for the ARMCI and Spin locks schemes because multiple thieves will have to wait for others to acquire the lock on the remote process, fail on the steal operation, and unlock before they themselves can move on to another victim. Thus, allowing consumers to abort when they detect that a producer's work has been exhausted eliminates a possibly linear time wait to abort the steal on the current victim.

In addition to this, ARMCI and Spin locks both spin on the lock which can cause scalability issues as the networking layer for the process that owns the lock may become saturated by remote lock operations. In the case of ARMCI locks, the remote process issues repeated one-sided *get()* operations to fetch the *current_ticket* counter. On our experimental system, these requests are truly one-sided and are performed using Infiniband RDMA operations. Spin locks perform repeated atomic swap operations which are serviced by ARMCI's remote data server. From the performance data, we can see that ARMCI's locks scale better under contention as the data server becomes more quickly saturated with requests.

If we compare the number of failed steals for ARMCI and Spin locks in Figure 7, an interesting trend emerges: in the case of Madness and BPC, spin locks cause an increase in the number of failures, while in the case of UTS they cause a decrease in the number of failures. This phenomenon is due to how spin locks influence the behavior of the reacquire operation. As described in Section 3.1, a reacquire operation is performed when a process transfers work from the public portion of its queue to the private portion of its queue. When using ARMCI locks, this means that the local process must take a ticket and wait in line behind any other processes to acquire the lock on the public portion of its own queue. When using spin locks, the local process issues native atomic exchange instructions to acquire its lock. The local process can issue these instructions orders of magnitude more quickly than remote processes can issue atomic swap

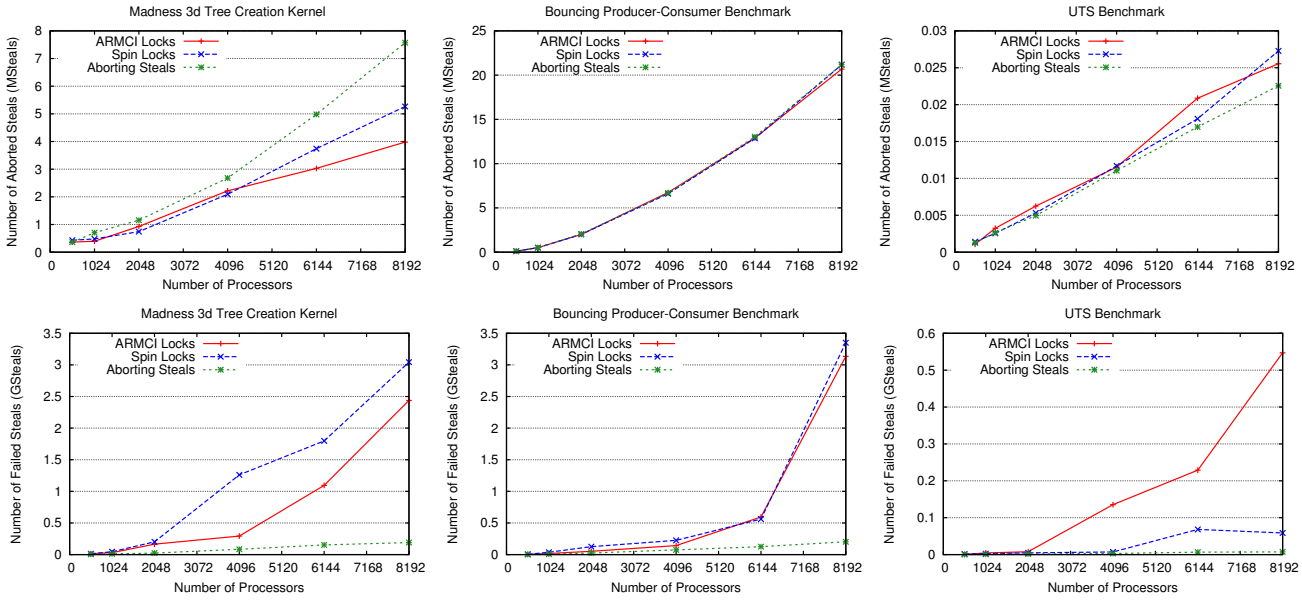

Figure 7: Total number of aborted and failed steals for Madness, UTS, and the BPC benchmark.

operations over the network. This effectively prioritizes the local process during a reacquire operation. This allows the local process to jump ahead of the queue and reclaim any shared work, causing many of the processes attempting to steal from it to fail. Thus, the total number of failures incurred when using spin locks is higher for both Madness and BPC.

In comparison with Madness and BPC, the duration of a UTS task is roughly three orders of magnitude shorter. Because of this, UTS also generates tasks orders of magnitude more rapidly that Madness and BPC. For this application, we observe that prioritizing the local process during reacquire operations gives a significant drop in the number of failed steals. This is because the local process is able to quickly reacquire work from the shared portion of its queue and generate new tasks to satisfy the thieves waiting for the lock on its queue before their steal operation fails.

6. RELATED WORK

Load balancing is a challenging problem and has been widely studied in the literature. Research on task graph scheduling [24] focuses on optimizing the execution time given a set of tasks organized as a task graph. Graph and hypergraph partitioning techniques [7, 11, 21, 35] have been employed to schedule tasks onto processors to balance load while taking into account data locality. We have addressed the problem of locality-aware load balancing for a set of independent tasks [2, 22]. Unlike in this paper, these works assumed knowledge of all the tasks to be executed before any tasks was executed.

OpenMP exploits parallelism at the loop level by distributing different iterations to different processors. While most of the work on OpenMP has focused on shared memory machines, recent efforts address the problems of optimizing OpenMP programs for distributed memory machines [19]. HPCS languages such as X10 [9], Chapel [8] and Fortress [34], and PGAS languages such as Titanium [39] and UPC [36] extend OpenMP's parallelism constructs and are focussed on scaling to the largest distributed memory machines available. These languages motivate our focus on dynamic load balancing at scale. The scalable implementation of the powerful task models of these new languages is still a challenging problem and we believe that the implementation and evaluation in this paper can offer useful insights to their implementers.

Charm++ [20] employs a phase-based dynamic load balancing scheme facilitated by virtualization. The computation is monitored for load imbalance and computation objects are migrated between phases to restore balance. While our idea of migrateable tasks is similar in spirit to virtualization in Charm++, the tasks in our context are much more fine grained and the load balancing is more dynamic and within each phase. Our implementation can be combined with a phased based load balancer, such as in Charm++, to provide hierarchical load balancing.

Cilk [16] supports load balancing of *fully strict* computations based on work stealing. X10 work stealing [10] extends the Cilk algorithm to support *terminally strict* computations and several other extensions. Recent work [26] has investigated execution models in which each task is executed *at least* once rather than exactly once. NESL [4] allows for locality aware work stealing. All these efforts focus on work stealing in the context of shared memory (or distributed shared memory) systems. Guo et al. [17] introduce and evaluate the *help first* scheduling policy on shared memory machines to increase parallelism in the application faster than Cilk-style *work first* execution. This approach is complementary to schemes we have evaluated to optimize the amount of work stolen and the cost of finding work.

Cilk NOW [6] extends Cilk to networks of workstations and adds support for adaptive parallelism and fault tolerance. Dynamic load balancing based on work stealing has been studied in various application contexts on distributed memory machines [23, 33] and on wide area networks [37].

Some of the techniques presented in the paper were ini-

tially explored in prior work on optimizing the Unbalanced Tree Search Benchmark (UTS) [13, 14, 31, 32] and the Scioto [12] infrastructure. Non-blocking work stealing [1, 18] has been studied in the literature. However, the implications of such a design on the performance of work stealing at scale has not been evaluated.

Our work in scaling work stealing to over 8000 processes has brought forth new issues, which we have discussed in this paper. We are not aware of any prior demonstrations of scalable load balancing through work stealing at such scale.

7. CONCLUSION

We have presented a runtime system for supporting work stealing on 8,192 processing cores on a distributed memory cluster. Scalability of work stealing on such a system was achieved through techniques such as split task queues and in-place task creation that reduce contention and optimize critical operations, a work splitting strategy that maximizes the number of work sources, and an efficient work discovery scheme that allows thieves to abort stale victims. We evaluated the performance of this runtime system on three benchmarks: a multiresolution analysis kernel, a producer-consumer benchmark, and a state space exploration benchmark. On these benchmarks our system was able to achieve 88%, 97%, and 99% efficiency, respectively.

8. REFERENCES

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proc. 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, 1998.

[2] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. J. Harrison, S. Hirata, S. Krishanmoorthy, S. Krishnan, C.-C. Lam, Q. Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, Feb. 2005.

[3] P. Berenbrink, T. Friedetzky, and L. Goldberg. The natural work-stealing algorithm is stable. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 178–187, 2001.

[4] G. E. Blelloch and J. Greiner. A provable time and space efflcient implementation of NESL. In *Proc. 1st ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP)*, pages 213–225, Philadelphia, Pennsylvania, May 1996.

[5] R. D. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. 35th Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Nov. 1994.

[6] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proc. USENIX Annual Technical Conference (ATEC)*, pages 10–10, 1997.

[7] Ü. V. Çatalyürek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and L. A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Proc. 21st Intl. Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11. IEEE, 2007.

[8] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Performance Computing Applications (IJHPCA)*, 21(3):291–312, 2007.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. Conf. on Object Oriented Prog. Systems, Languages, and Applications (OOPSLA)*, pages 519–538, 2005.

[10] G. Cong, S. Kodali, S. Krishnamoorty, D. Lea, V. Saraswat, and T. Wen. Solving irregular graph problems using adaptive work-stealing. In *Proc. 37th Int Conf. on Parallel Processing (ICPP)*, Portland, OR, Sept. 2008.

[11] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New challanges in dynamic load balancing. *J. Appl. Numer. Math.*, 52(2-3):133–152, 2005.

[12] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. In *Proc. 37th Intl. Conf. on Parallel Processing (ICPP)*, pages 586–593, 2008.

[13] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *Proc. of 6th Intl. Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS)*, pages 1–8, 2007.

[14] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. A message passing benchmark for unbalanced applications. *J. Simulation Modelling Practice and Theory*, 16(9):1177 – 1189, 2008.

[15] N. Francez and M. Rodeh. Achieving distributed termination without freezing. *IEEE Trans. on Software Engineering*, SE-8(3):287–292, May 1982.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. Conf. on Prog. Language Design and Implementation (PLDI)*, pages 212–223. ACM SIGPLAN, 1998.

[17] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for terminally strict parallel programs. In *Proc. 23rd Intl. Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[18] D. Hendler, Y. Lev, M. Moir, and N. Shavit. A dynamic-sized nonblocking work stealing deque. *J. Distributed Computing*, 18(3):189–207, 2006.

[19] Intel Corporation. Cluster OpenMP user's guide v9.1. (309096-002 US), 2005-2006.

[20] L. V. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proc. Conf. on Object Oriented Prog. Systems, Languages, and Applications (OOPSLA)*, pages 91–108, 1993.

[21] G. Karypis and V. Kumar. *MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0*, Sept. 1998.

[22] S. Krishnamoorthy, Ü. Çatalyürek, J. Nieplocha, A. Rountev, and P. Sadayappan. Hypergraph partitioning for automatic memory hierarchy management. In *Proc. ACM/IEEE Conference Supercomputing (SC)*, page 98, 2006.

[23] V. Kumar, A. Y. Grama, and N. R. Vempaty. Scalable load balancing techniques for parallel computers. *J. Parallel Distrib. Comput.*, 22(1):60–79, 1994.

[24] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[25] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[26] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 45–54, Feb. 2009.

[27] M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *Proc. 10th Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 212–221. ACM, 1998.

[28] MPI Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[29] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *Lecture Notes in Computer Science*, 1586:533–546, 1999.

[30] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *Proc. ACM/IEEE Conference Supercomputing (SC)*, pages 340–349, 1994.

[31] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An unbalanced tree search benchmark. In *Proc. 19th Intl Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 235–250, 2006.

[32] S. Olivier and J. Prins. Scalable dynamic load balancing using UPC. In *Proc. of 37th Intl. Conference on Parallel Processing (ICPP)*, Sept. 2008.

[33] A. Sinha and L. V. Kalé. A load balancing strategy for prioritized execution of tasks. In *Proc. 7th Intl. Parallel Processing Symposium (IPPS)*, pages 230–237, 1993.

[34] G. L. Steele Jr. Parallel programming and parallel abstractions in fortress. In *Proc. 14th Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, page 157, 2005.

[35] A. Trifunović and W. J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *J. Parallel Distrib. Comput.*, 68(5):563–581, 2008.

[36] UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[37] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proc. of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2001.

[38] T. Yanai, G. I. Fann, Z. Gan, R. J. Harrison, and G. Beylkin. Multiresolution quantum chemistry in multiwavelet bases: Analytic derivatives for hartree–fock and density functional theory. *J. Chemical Physics*, 121(7):2866–2876, 2004.

[39] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.