Lifeline-based Global Load Balancing

Vijay Saraswat
IBM TJ Watson Research Center
vijay@saraswat.org

Prabhanjan Kambadur
IBM TJ Watson Research Center
pkambadu@us.ibm.com

Sreedhar Kodali
IBM Systems and Technology Group
srkodali@in.ibm.com

David Grove
IBM TJ Watson Research Center
groved@us.ibm.com

Sriram Krishnamoorthy
Pacific Northwest National Laboratory
sriram@pnl.gov

Abstract

On shared-memory systems, Cilk-style work-stealing [5] has been used to effectively parallelize irregular task-graph based applications such as Unbalanced Tree Search (UTS) [24, 28].

There are two main difficulties in extending this approach to distributed memory. In the shared memory approach, thieves (nodes without work) constantly attempt to asynchronously steal work from randomly chosen victims until they find work. In distributed memory, thieves cannot autonomously steal work from a victim without disrupting its execution. When work is sparse, this results in performance degradation. In essence, a direct extension of traditional work-stealing to distributed memory violates the *work-first principle* underlying work-stealing. Further, thieves spend useless CPU cycles attacking victims that have no work, resulting in system inefficiencies in multi-programmed contexts. Second, it is non-trivial to detect *active distributed termination* (detect that programs at all nodes are looking for work, hence there is no work). This problem is well-studied and requires careful design for good performance. Unfortunately, in most existing languages/frameworks, application developers are forced to implement their own distributed termination detection.

In this paper, we develop a simple set of ideas that allow work-stealing to be efficiently extended to distributed memory. First, we introduce *lifeline graphs*: low-degree, low-diameter, fully-connected directed graphs. Such graphs can be constructed from k-dimensional hypercubes. When a node is unable to find work after w unsuccessful steals, it quiesces after informing the outgoing edges in its lifeline graph. Quiescent nodes do not disturb other nodes. A quiesced node is reactivated when work arrives from a lifeline, and itself shares this work with those of its incoming lifelines that are activated. Termination occurs precisely when computation at all nodes has quiesced. In a language such as X10, such passive distributed termination can be detected automatically using the **finish** construct – no application code is necessary.

Our design is implemented in a few hundred lines of X10. On the binomial tree described in [26], the program achieve 87% efficiency on an Infiniband cluster of 1024 Power7 cores, with a peak throughput of 2.37 GNodes/sec. It achieves 87% efficiency on a Blue Gene/P with 2048 processors, and a peak throughput of 0.966 GNodes/s. All numbers are relative to single core sequential performance. This implementation has been refactored into a reusable *global load balancing framework*. Applications can use this framework to obtain global load balance with minimal code changes.

In summary, we claim: (a) the first formulation of UTS that does not involve application level global termination detection, (b) the introduction of lifeline graphs to reduce failed steals (c) the demonstration of simple lifeline graphs based on k-hypercubes, (d) performance with superior efficiency (or the same efficiency but over a wider range) than published results on UTS. In particular, our framework can deliver the same or better performance as an unrestricted random work-stealing implementation, while reducing the number of attempted steals.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Algorithms, Design

Keywords UTS, global load balancing, distributed computing, X10, work-stealing, parallel programming

1. Introduction

The emergence of new architectures that emphasize distributed memory — clouds and commodity clusters, P7, and Blue Gene – provides significant new opportunities for application developers. New application areas such as business analytics, data mining are presented with unparalleled opportunities to deal efficiently with large workloads. However, these exciting opportunities bring new challenges for parallel system designers.

The APGAS programming model [30] provides a useful and convenient framework for stating these problems and their solutions. The parallel system is viewed as a collection of *places*. Data is partitioned across the places with support for selective replication. In addition to remote data access through one-sided communication primitives, *activities* can be launched on remote places. This allows complementary approaches to move the data or the computation as they match the application domain (e.g., linear algebra vs tree traversal). The APGAS framework also subsumes other newer programming models such as Map-Reduce [8]. Map-Reduce frameworks such as Hadoop allow parallel processing of large files using user-defined map and reduce operations. However, it is challenging to rewrite well-established parallel algorithms to fit the map-reduce model.

A fundamental problem in achieving the promise of scale-out computing in such a programming model is the *global load-balancing problem*. Many problems can be conceptualized in terms of distributed task collections, with dependences between them expressed as data or control dependences. In this paper, we focus on task collections with no dependences. In addition, the tasks encapsulate all data needed for their processing and do not take advantage of the global address space.

Dynamically load balancing such a computation assumes these tasks to be mobile — they can be executed anywhere and do not exhibit affinity to a place. This paper presents a solution to the global load balancing problem for such a task collection.

This problem has been studied in the context of shared-memory systems. In a typical design, the tasks to be processed at each place are placed in a deque. The worker at a given place processes tasks from the top, while thieves steal from the bottom of the deque. Workers continues to processes it local tasks until they run out of work. Thieves attempt to minimize their idle time by looking for work to steal. If T_1 is the time to run a program with one worker and T_∞ is the time required to run the same program with an infinite number of workers, then T_p, the time to run this program with P workers is $T_1/P + T_\infty$. Simultaneously, the space required to run a program with P workers is bounded by $P \times S(1)$, where $S(1)$ is the space required to run the program with one worker. This approach is usually referred to as "work-stealing".

Although work-stealing appears simple at the outset and has attractive space and time bounds, efficient parallelization of applications using the work-stealing scheduler requires careful engineering. The key factor in getting optimal performance in a work-stealing scheduler is reducing the critical path overhead; that is, a worker busy with work must continue with this work and not be interrupted to help a thief. Implementations, such as Cilk's "THE" protocol [15] have been carefully designed so that a worker thread does not pay more than the cost of a volatile write in all those cases where it is not a victim. In fact, the THE protocol also ensures that victims do not pay the cost of locking and unlocking the deque unless there is a single task to be wrestled for. However, if the worker is a thief, it has to acquire a lock before stealing a task from the victim. In this scheduling scheme, contention arises when: (1) multiple thieves try to steal from the same victim, (2) there is a single task on the victim's deque, or (3) both.

In a distributed memory setting, local and remote data accesses incur different costs — the cost of communication cannot be ignored. Additionally, different communication operations might incur very different costs (eg., lock operations vs data access). While lock contention can be expensive in shared memory machines, it has been shown to dramatically impact parallel efficiency at scale in distributed memory contexts [11]. On many architectures, the operations involved in stealing work are not supported in hardware. As a result, steals interrupt the remote worker, incurring additional cost. In particular, the time to process a given set of tasks depends not just on the tasks being processed, but potentially also on the number interruptions due to incoming steal requests.

One central problem is termination detection [4, 9, 10, 14]. In the usual formulation, computation must terminate once it is the case that *every* worker is looking for work; for then no worker has work. In the shared memory case, this can be implemented with a simple barrier algorithm. When a worker finishes its work, it checks into the barrier. If it finds it is the last worker to check into the barrier, it signals termination to all other workers. Otherwise it looks for work randomly, periodically checking to see if termination has been signaled (and terminating if it is). If it finds work, it checks out of the barrier before claiming the work, thus guaranteeing correctness.

As the number of workers scales, repeatedly checking into and out of the barrier can cause performance problems, and more scalable algorithms have to be designed. [26] proposes that workers enter the barrier only when they are "nearly certain" that there is no work in the system. This heuristic decision is made as follows. The worker randomly selects a victim. If the victim has no work, the thief moves to the next worker. Once it completes a circuit of all workers it makes the decision that there is likely no work in the system and enters the barrier. These additional traversals are of size $O(P)$ and can increase the latency to termination.

On scale out, the single location can become a bottleneck. Instead a combining tree must be used [11]. A node forwards a termination signal only when it has processed all its work. The algorithm involves phases of signaling up and down the tree. In the up phase, each node signals its parent for successful termination only if such a decision is reported by this node and its children. In the down phase, each node forwards the signal to its children and exit task processing if a terminate signal was received. The root of the tree broadcasts a terminate signal through its children only if its children and itself voted for termination. When a victim of a steal since the last vote becomes idle, it votes not to terminate. Thus termination is successfully detected by the root node only when all nodes participate in the up phase with no steals since the last up phase. This results in multiple termination detection phases during the computation, which is also slowed down due to active stealing by the nodes.

We contrast the problem with the (somewhat dual) *passive termination detection problem* in X10. In X10 any activity may spawn ("push") more activities on any node in the system. A **finish** operator must detect when all activities spawned within its scope have terminated. In contrast with the active version of the problem, worker threads at each place are passive, they wait for the arrival of messages containing work rather than actively searching for work themselves.

X10 implements a particular version of *vector counts* to detect passive termination. Each worker maintains a vector of counts, one per place.[1] This vector tracks how many asyncs this place has created remotely, and how many asyncs have terminated at this place. Once a place has quiesced (there is no activity running at that place), it sends its vector to the place that spawned the finish. This place simply sum-reduces the count (component-wise). Termination is detected once (and only once) the reduced vector is zero. For computations in which a place spawns computations at only a few other places, say bounded by a constant z, only vectors of

[1] This vector is usually very sparse and is maintained sparsely.

size z need to be communicated between places. Note that these design does not involve speculative waves of termination detection, rather it simply monotonically accumulates a set of vector counts, until the set reduces to zero.

When attempting to solve the UTS problem in X10 the natural question is whether **finish** can be used to implement UTS termination detection.

The central contribution of this paper is that it can. We show that global work stealing can be elegantly formulated as a simple X10 program using **async**, **at** and **finish**. The details are in the next section, we summarize the basic principles here.

Initially one **async** is launched at every place, under the control of a single **finish**. The termination of this **finish** will signal global termination. These asyncs will initially attempt to look for work by guessing a random id, and looking for work at the place with that id. This can be done using the **at** operator, without spawning any new asyncs. Now in "pure" work-stealing, if the async did not find any work at this victim, it will continue looking for work at other victims. This leads immediately to the active termination detection problem: when should the async know to stop looking for work, because there is no work?

Our key insight is to perform such random steals at most w times, where w is some pre-determined bound. If no work has been found after w attempts, the async *terminates*. However, before it does so, it establishes one or more *lifelines*. A lifeline is simply another place. Establishing a lifeline means checking if that place has work, and if so performing a steal, as usual. But if that place does not have work then the id of the thief is recorded at that place as an "incoming" lifeline. Since that place does not have any work, it must itself, recursively, have established a lifeline. Once a place finds some work, it checks to see if any incoming lifelines have been recorded. If so, it *distributes* a portion of its work to this lifeline q, and clears it. Distribution is performed by spawning an async at place q, which *re-initiates* activity at q. Now it can be seen that there is no more work left to do precisely when all the asyncs in the system have terminated – and this condition will, of course, be detected by the single top-level **finish**.

1.1 Lifeline graphs

While the lifeline scheme is correct, it may not be efficient. How are lifelines determined? A natural possibility is that a lifeline is simply another place chosen at random. But it is easy to see this would not work. Place p may randomly choose to make q its lifeline, and simultaneously q could choose to make p its lifeline. As a result both places will be dead – they will not have any running activity and will never get one. Hence throughput and scalability will suffer. A good lifeline graph must have the property that as long as there is a place which has work, there must be a path from that place to every other place in the system.

Another alternative is to base the lifeline graph on a permutation with cycle of length P. For instance, each place p can be mapped to place $(p+1)\%P$. This guarantees that the only cycle is of length P: thus a place will be involved in a cycle only if all places are involved in it, i.e. only when there is no work.

This scheme is correct and works reasonably well for small P. The problem is that it takes on the average $P/2$ hops for work to reach a place from another place. During this time the target place is idle.

An alternative would be to consider the fully connected graph, with an edge from every vertex to every other vertex. While this ensures that tasks have to take only one hop to reach an idle worker, it creates the problem that tasks at one site have to be divided up into many pieces, one for each incoming lifeline. Workers with tasks on their hand may waste cycles moving tasks to their (many!) incoming lifelines, instead of executing them.

We are interested therefore in (directed) graphs that have the following properties. (a) Each vertex must has bounded out-degree. (b) The graph must be connected, (c) The graph must have a low diameter (there should be short paths from every vertex to every other vertex).

A parametric family of graphs that satisfy this property are the *cyclic hypercubes* defined as follows. Choose a radix h and a power z such that $h^{z-1} < P \leq h^z$ (P is the number of processes/places). Each vertex p is represented as a number in base h with z digits. It has an outgoing edge to every vertex a distance $+1$ from it in the Manhattan distance (in modulo h arithmetic). That is, the vertex p labeled $(a_1, \ldots, a_z)$ has an outgoing edge to every vertex q such that for some $i \in 1..z$, $q = (a_1, \ldots, (a_i+1)\%h, \ldots, a_z)$. In two dimensions ($z = 2$), we have a square of size h^2 with all the elements in a row connected in a cycle, and all the elements in a column connected in a cycle.

In such a graph the average number of hops for work to travel from one place to another is $(h \times z)/2$.

One final point is worth mentioning. Usually $P < h^z$, and so P must be embedded in a graph with h^z nodes. Care must be taken to ensure that each place is connected to the next node in that dimension that represents a real, distinct place. This may mean that in a particular dimension a node has no neighbor (i.e. it has less than z lifelines). It is not difficult to show that for every $P > 1$ every node must have at least one neighbor.

1.2 Results

The lifeline scheme has been implemented in X10. The core implementation for binomial trees (presented in the next section) is around 150 lines of code (not counting the implementation of SHA1Rand). [2] The full implementation – with libraries for w-adic numbers, support for geometric trees, command-line processing, timing, data-collection and comments – is about 1500 lines.

The same code has been timed on three different platforms – a small x86-Infiniband cluster (Triloka), a Blue Gene/P machine and a Power7-Infiniband cluster. On the 157 billion node binomial tree reported on in [26], the implementation shows an efficiency of 87% for 1024 cores on BG/P (483.2 vs 0.54 M Nodes/sec), and 86% for 1024 cores on Power7 (2371 vs 2.7 M Nodes/sec), and 94% for 128 cores on x86 (253.4 vs 2.1 M Nodes/sec). All comparisons are with sequential performance. These numbers contrast with the 80% efficiency reported in [26] on 1024 cores of an x86 cluster. [11] reports 99% efficiency at 8192 cores as compared to a baseline of 512 cores on an x86 cluster. These results involved dedication of 1 out of the 8 cores in each SMP node to assist in work stealing. As compared to the approach in this paper, which utilizes all available cores to perform work, the effective efficiency reported in [11] is 87%. The efficiency numbers reported here are over a larger range of process counts (sequential run vs 128 and 1024 cores, thus $128x$ and $1024x$) as compared to $8k/512 = 16x$ as reported in [11].

[2] Due to space constraints, we refer you to [25, 26] for the exact description of both binomial and geometric trees.

On a larger binomial tree, of size 416b, the implementation shows an efficiency of 87% for 2048 cores of BG/P (966.11 vs 0.54 MNodes/sec).

We have also benchmarked the implementation on a geometric tree of size 109B. The implementation shows an efficiency of 89% (1683 vs 1.85 M Nodes/sec) on the Power7 cluster and an efficiency of 92% (357.6 vs 0.38 M Nodes/sec) on the Blue Gene/P cluster. [3]

1.3 Rest of this paper

In the next section we present the scheme in more detail, illustrating with the relevant fragments of X10 code. The code is released under the Eclipse Public Licence and is available from the X10 repository on Sourceforge.[4]

We present the results in more detail in Section 3. Finally in Section 4 we discuss more details of related work.

2. Basic lifeline scheme

We now present the basic scheme by discussing the actual X10 code for the implementation.

To understand the presentation below the following characteristics of X10 must be kept in mind.

- An **async**(p) S statement launches an activity at place p. This activity executes S and terminates. The invoking activity continues immediately, without waiting for the newly created activity to terminate.
- An **at**(p) S statement executes statement S synchronously at place p. That is, the execution of the activity is suspended until S terminates. An **at**(p) e expression evaluates e at p and returns the result.
- A **finish** S statement executes statement S and waits until all activities created during its execution have themselves (recursively) terminated. These activities may be launched at the current place or some remote place.
- The implementation below assumes each place is running a single worker, and workers are not dynamically created.[5]
- Therefore user code must explicitly call Runtime.probe() to service incoming (synchronous or asynchronous) requests. It is up to user code to determine the frequency of polling. (See below for a further discussion of this point.)
- Incoming messages are processed only at specific places in user code, namely calls to Runtime.probe() in the user code or to remote operation invocations (i.e., the asynchronous **async**(p) S operation or synchronous **at**(p) S). At this point zero or more incoming messages (**async**'s, **at**'s) may be processed.[6]
- Since all data-structures are touched by a single worker thread, no locks are needed to guarantee atomicity or mutual exclusion.
- Like Cilk, we use deques to store unexecuted tasks; the owning process operates on the shallow end of the deque (using it as a stack), whereas responses to stealing and lifeline requests operate on the deep end of the deque. This is important even in the absence of multiple threads (i.e., contention) because it promotes cache reuse — there is a greater chance that unexecuted tasks that have just been spawned are not stolen and hence are still in cache.
- If no user code is running at a place, the X10 worker continues to run and will process incoming messages until a termination signal is received from place 0.

The code below uses a deque, implemented as a circular buffer. The deque supports push() and pop() operations on one end and a steal(i) operation at the other end (the parameter i specifies the number of elements to remove).

The design satisfies the following invariants:

- At any time, at most one activity runs in a given place.
- At any time, at most one message has been sent on an outgoing lifeline (and hence at most one message has been received on an incoming lifeline).

First, an object is created at each place. This object maintains information about the state of the execution at that place (e.g. counters). This object is referenced through a place local handle, st(). This handle may be freely communicated from place to place. At any place P, the object associated with this handle at that place may be obtained by simply applying st() to the empty argument list, that is, evaluating st()[7]

Specifically the object maintains the following information:

- various parameters associated with the tree being constructed. Of particular interest are: (a) nu, the maximum number of tasks that will be popped for execution from the dequeue before distribution and polling, (b) w, the maximum number of random steals made before turning to lifeline steals, (c) z, the dimensionality of the lifeline graph, (d) k, the number of items to steal at a time ($k = 0$ for "steal half").
- the **Boolean** control variable active that is true *iff* a user-activity is running at the place.
- the **Boolean** control variable noLoot which is used to record whether loot (number of stolen tasks) has arrived asynchronously.
- the array lifelinesActivated contains a **Boolean** per place. lifelinesActivated(p.id) is true only if this place has activated the outgoing lifeline to p, and has not yet received loot in return. In principle only z values need to be recorded; the current implementation keeps an array of P booleans.
- a stack thieves that records which incoming lifelines have been activated. The size of this stack is bounded by z.

Launching work. Computation is initiated from place 0. The given root node is expanded one level; this will typically cause tasks to be added to the deque. Then an **async** is spawned at each place (other than 0). The async is given an initial apportionment of loot; this may be empty. Finally the current activity continues by processing its own deque, after

[3] Let Th_s and Th_p be the throughputs achieved for a program when run serially and in parallel with P processors, respectively. Then, we define efficiency as $Th_p/(Th_s \times P)$.

[4] https://*x10.svn.sf.net/svnroot/x10/benchmarks/trunk/UTS*.

[5] The programmer ensures this by executing the program with the environment variables X10_NUM_THREADS and X10_STATIC_THREADS set to appropriate values.

[6] X10 supports the bodies of incoming **async**'s and **at**'s may themselves perform network operations, including Runtime.probe(). Thus user code must be written in such a way that it is prepared to handle incoming messages recursively.

[7] Effectively one may think of a place local handle as a handle to a distributed array of elements; at any place, the id of that place may be used to access the element of the array stored at that place.

marking itself as active. Once it has finished processing its stack, it terminates.

```
def main (st:PLH, rootNode:TreeNode) {
 finish {
   processBinomialRoot (b0, rootNode, deque);
   val lootSize = stack.size()/Place.MAX_PLACES;
   for (var pi:Int=1 ; pi<P ; ++pi) {
     val loot = deque.steal(lootSize);
     async (Place(pi))
       st().launch(st, true, loot, 0);
   }
   active=true;
   processDeque(st);
   active=false;
 }
}
```

When an activity is launched at a place p from place s, it first resets the (local) **Boolean** flag lifelinesActivated(s) at p (thus enabling code running at p to establish a new lifeline back to s at a future point in time). It then checks to see if an activity is already running (is active true?). If so, it simply processes the loot and terminates. Otherwise it becomes "the" activity at this place. After processing its loot, it determines whether there are any incoming lifelines, and if so distributes work through those lifelines. Finally it processes all the elements remaining on its deque until the deque is empty (note this could take a long time). It then terminates. (Note that launch is invoked at place either by the main activity or through the distribution mechanism – see distribute below.)

```
def launch(st:PLH, init:Boolean,
  loot:ValRail[TreeNode], depth:Int, source:Int) {
    lifelinesActivated(source) = false;
    if (active) {
      noLoot = false;
      for (r in loot) processSubtree(r);
    } else {
      active=true;
      for (r in loot) processSubtree(r);
      if (depth > 0)
        distribute(st, depth+1);
      processDeque(st);
      active=false;
  }}
def processSubtree (node:TreeNode) {
    ++nodesCounter;
    binomial (q, m, node, deque);
 }
```

Local work execution. The deque is processed in a loop until empty. Each time around in the loop, at must nu items are processed from the deque. The network is then probed to handle incoming messages. If some thieves have registered themselves, loot is distributed. When the deque becomes empty, an attempt is made to steal from other workers. If no loot is found[8] the activity terminates, otherwise it resumes processing the deque.

```
  def processDeque(st:PLH) {
    while (true) {
      var n:Int = min(deque.size(), nu);
      while (n > 0) {
        for ((count) in 0..n−1)
          processSubtree(deque.pop());
          Runtime.probe();
          val numThieves = thieves.size();
          if (numThieves > 0)
            distribute(st, 1, numThieves);
          n = min(deque.size(), nu);
      }
      val loot = attemptSteal(st);
      if (null != loot)
        processLoot(loot, false);
      else {
        if (! noLoot) {
          noLoot=true;
          continue;
        }
        else
          break;
  }}}
```

[8] note that loot may arrive synchronously through the call to attemptSteal() or it could arrive asynchronously through an incoming distribution, discussed below

We use a command line parameter (n) to control the frequency of polling. This can affect the performance of the program. Frequent polling incurs overhead and comes in the way of the worker executing real work. Infrequent polling means that steal requests from thieves are not processed quickly, thereby stalling thieves. In practice we have found n=511 or n=1023 gives good results for most UTS workloads.

Distributions. A distribution is made to an incoming lifeline only if there is enough work. Work is popped from the deep-end of the deque. To distribute work, an `async` is launched at the target place. Note that this async is governed by the single **finish** in main().

```
  def distribute(st:PLH, depth:Int) {
    val n = thieves.size();
    if (n > 0)
       distribute(st, 1, n);
  }
  def distribute(st:PLH, depth:Int, var n:Int) {
    val lootSize= deque.size();
    if (lootSize > 2) {
      n = Math.min(n, lootSize−2);
      val s = lootSize/(n+1);
      for ((i) in 0..n−1) {
        val thief = thieves.pop();
        val loot = deque.steal(s);
        async (Place(thief))
          st().launch(st, false, loot, depth);
      }
    }
  }
```

Various modifications are possible and will be explored in future work. Instead of dividing the current set of tasks evenly among all the recipients, the donor could randomly select one of the thieves and send it a portion of the loot (depending on the value of k). This would be a dual to random work-stealing – here work is "pushed" to places that have indicated earlier that they needed work (note they may no longer need work, since some other lifeline may have supplied them).

Another important consideration is that as the code is written it copies the items out of the deque one at a time for each target destination. An alternative would be for the activity to block off portions of the deque and trigger asynchronous DMA's to transfer the loot to the destination. This may result in better performance, particularly at high z's, and in cases where the size of the deque can grow large.

Stealing. To make a steal, an activity randomly guesses another place, and uses the **at** operation to retrieve loot from that place. It tries this at most w times, also breaking immediately if loot arrives asynchronously, for example because it is being distributed on a lifeline.

If no loot is received during this phase, the activity tries its lifelines one after the other, returning as soon as loot is found. Care is taken to ensure that a request is not made of an outgoing lifeline if a request has already been made of that lifeline and no loot has been received from it so far. The difference between the stealHandler() invocations in the direct steal phase and in the lifeline phase is the second argument: this is set to **true** only in the lifeline phase.

```
def attemptSteal(st:PLH):ValRail[TreeNode] {
  if (Place.MAX_PLACES == 1) return null;
  for (var i:Int=0; i<w && noLoot; i++) {
    var q_:Int = 0;
    val p=here.id;
    while((q_= myRandom.nextInt(Place.MAX_PLACES))== p) ;
    val q = q_;
    val loot = at (Place(q)) st().stealHandler(p,false);
    if (loot != null) return loot;
  }
  if (! noLoot) return null;
  for (var i:Int=0;
         (i<myLifelines.length()) && noLoot
         && 0<=myLifelines(i);
        ++i) {
    val L:Int = myLifelines(i);
    if (!lifelinesActivated(L) ) {
      lifelinesActivated(L) = true;
      val loot = at(Place(L)) st().stealHandler(p, true);
      if (null!=loot) {
        lifelinesActivated(L) = false;
        return loot;
      }}}
  return null;
}
```

A try at a steal is handled by examining the size of the deque. If the deque has enough tasks, then they are popped from the deque and returned. Otherwise if isLifeLine is set, the place making the try is recorded in the thieves stack (used during distribution).

```
def stealHandler(p:Int, life:Boolean):ValRail[TreeNode] {
  val length = deque.size();
  val numSteals = k==0u ? (length >=2u ? length/2u : 0u)
  : (k < length ? k : (k/2u < length ? k/2u : 0u));
  if (numSteals==0u) {
    if (life)
      thieves.push(p);
    return null;
  }
  return deque.steal(numSteals);
}
```

This is a very simple scheme and different from the implementation used in [26] and the implementation described in [11]. [26] implements steals as follows. Each worker polls periodically. When it receives a steal request, it determines the amount of loot to release and removes it logically from the deque (without copying it out). It returns a reference to the loot. The thief then initiates a separate DMA to retrieve the loot. This DMA request is handled purely by the network adapter without disrupting the worker. Thus the worker does not need to spend time copying the loot, and avoids polluting its cache. [11] keeps an extra helper thread on the side to perform the control operations necessary for stealing. Data is also transfered by DMA where possible.

X10 has idioms for expressing DMA transfer to remote locations. However for the trees we are considering, the loot to be transferred is rarely more than a few hundred elements, and stealing is relatively infrequent. For such systems it is not clear that the more elaborate implementation wins.[9] In future work we plan to implement DMA steals and evaluate the trade-off on various examples within the context of a unified implementation.

Consistent with the results reported in [26], we have found that "steal half" works best for binomial trees. This is not surprising given that the binomial tree is such that each node has the same potential to generate work. Stealing half leads to more rapid diffusion of work through the system. This is in contrast to the fixed-function geometric tree in which nodes higher in the tree have a much greater potential to generate work than nodes lower in the tree. For such systems steal-half gives poor performance and fixed-size steals are better. In particular for the geometric trees we have investigated, we have found that stealing 7 items at a time works well.

3. Results and Analysis

The results presented in this paper were obtained by compiling the program with the 2.0.5 version of the X10 compiler.

All speedups are reported with respect to the performance of a sequential implementation of UTS. The sequential program is iterative and uses the same deque data-structure used by the parallel program to record the nodes of the tree being constructed. It does not have any parallel overheads associated with it – specifically it does not invoke the `probe` operation. However, because sequential runtimes for large trees can be very large, we reduced the size of the input on which the sequential program was run. Table 1 gives a detailed description of the trees used to test the sequential performance of UTS on our test machines.

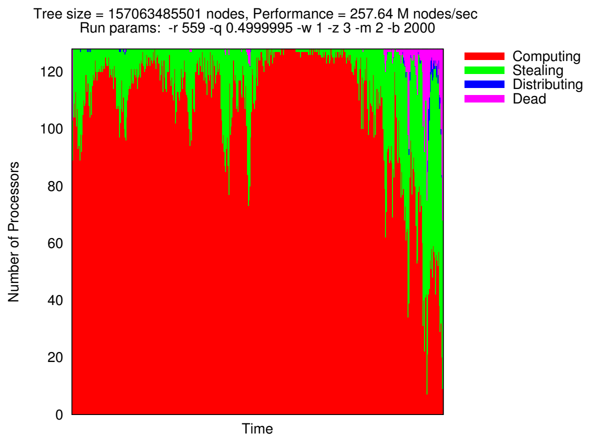

Figure 1. States of the compute nodes during the execution of X10's UTS application execution over time for a BINOMIAL tree of size 157 billion nodes. The results were collected on IBM's Triloka Linux cluster. Each node attempted 1 random steal (the parameter w) before resorting to one of its 3 lifelines (the parameter z).

[9] We instrumented our implementation and determined that the worker was spending less than 0.01% of its compute time in copying this data. This analysis did not account for cache misses due to pollution.

Machine	Tree type	t	r	b	m	q	a	d	Tree size	MNodes/sec
Triloka	BINOMINAL	0	559	2000	2	0.49995	—	—	57354859	2.104
ANL-BG/P	BINOMINAL	0	559	2000	2	0.4995	—	—	2859057	0.54
P7HV32	BINOMIAL	0	42	20000	6	0.166666665	—	—	510009729	2.7
P7HV32	GEOMETRIC	1	0	4	—	—	3	10	6700654	1.85
IBM-BG/P	GEOMETRIC	1	0	4	—	—	3	10	6700654	0.37

Table 1. As sequential execution (*not* parallel execution on 1 node) of UTS for very large trees requires excessive machine time, the sequential performance of UTS on various machines for both BINOMIAL and GEOMETRIC trees was measured using smaller trees. The command-line parameters used to generate the trees for sequential benchmarking are shown in this table. For a full explanation of the meaning of the command-line parameters to UTS, please refer to [24]. ANL-BG/P and IBM-BG/P are abbreviations for the Blue Gene/P clusters at Argonne National Laboratory and IBM T.J. Watson Research Center,respectively.

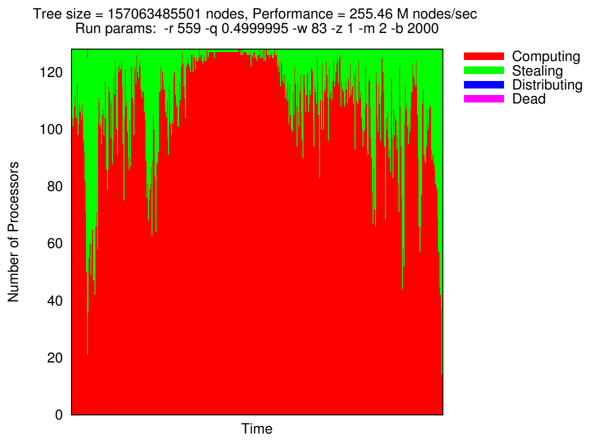

Figure 2. States of the compute nodes during the execution of X10's UTS application execution over time for a BINOMIAL tree of size 157 billion nodes. The results were collected on IBM's Triloka Linux cluster. Each node attempted 83 random steals (the parameter w) before resorting to its single lifeline (the parameter z).

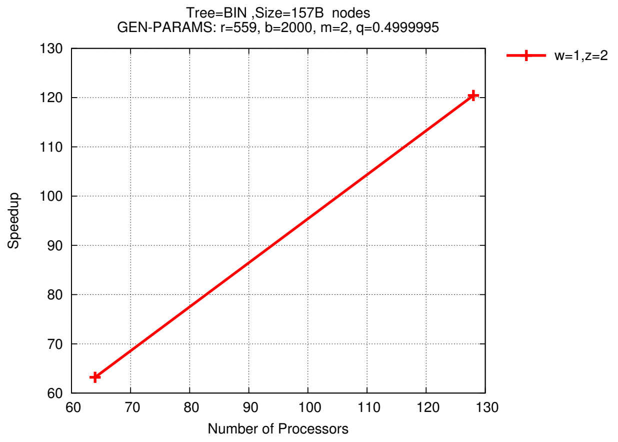

Figure 3. UTS speedup achieved for a 157 billion node BINOMIAL tree on IBM's Triloka cluster. The speedup shown is based on the sequential performance of UTS, which is 2.104 million nodes per second. The parameter w indicates the number of random steals attempted before resorting to lifelines. The parameter z indicates the number of lifelines for each compute node. The UTS revision number used was 15277.

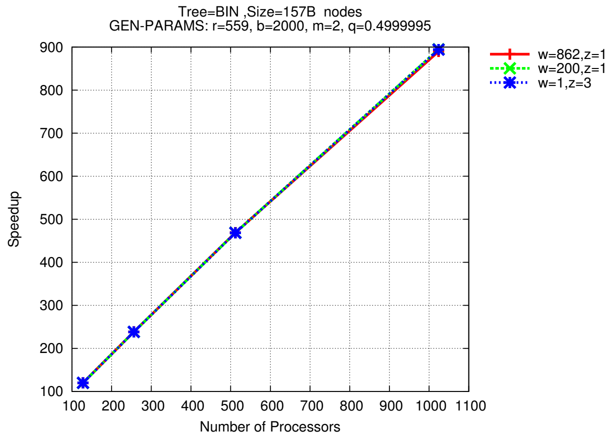

Figure 4. UTS speedup achieved for a 157 billion node BINOMIAL tree on Argonne National Lab's Blue Gene/P cluster. The speedup shown is based on the sequential performance of UTS, which is 0.54 million nodes per second. The parameter w indicates the number of random steals attempted before resorting to lifelines. The parameter z indicates the number of lifelines for each compute node. The UTS revision number used was 15277.

3.1 Experimental Platforms

We used three different experimental platforms for our empirical evaluation.

- Triloka: a small cluster of 16 IBM LS-22 blades connected by a 20Gb/sec IB network. Each node has 2 quad-core AMD 2.3Ghz Opteron processors, 16 GB of memory, and is running Red Hat Enterprise Linux 5.3.
- Blue Gene/P: We used the `surveyor` and `intrepid` systems located at the Argonne National Labratory and the `Watson4P` system located at the IBM T.J. Watson Research Center. Each compute node in a Blue Gene/P system has 2 GB of memory and 4 850 MHz PowerPC 450 processors each with a dual floating point unit.
- P7HV32: This is a 32-node Power7 cluster interconnected by Infiniband. Each compute node has 32 3.3 GHz processor cores, 128GB of physical memory and runs SuSE Linux Enterprise Server v 11.1 for ppc64.

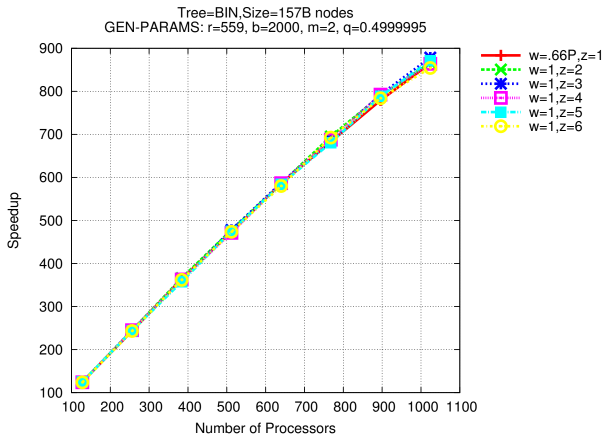

Figure 5. UTS speedup achieved for a 157 billion node BINOMIAL tree on IBM's P7HV32 cluster. The speedup shown is based on the sequential performance of UTS of the same problem, which is 2.7 million nodes per second. The parameter w indicates the number of random steals attempted before resorting to lifelines. The parameter z indicates the number of lifelines for each compute node. The UTS revision number used was 15294.

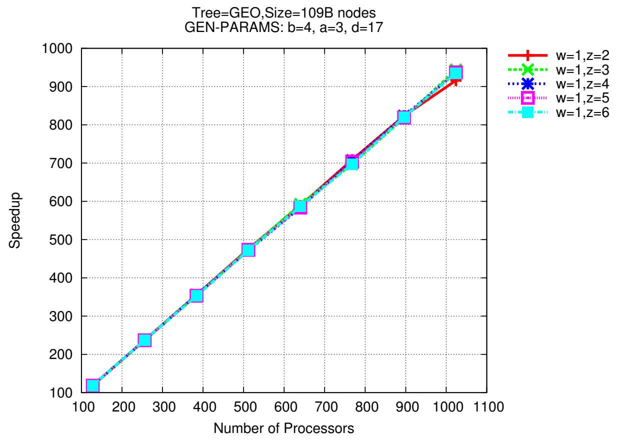

Figure 7. UTS speedup achieved for a 109 billion node GEOMETRIC tree on IBM's Blue Gene/P cluster. The speedup shown is based on the sequential performance of UTS of the same problem, which is 0.37 million nodes per second. The parameter w indicates the number of random steals attempted before resorting to lifelines. The parameter z indicates the number of lifelines for each compute node. The UTS revision number used was 15290.

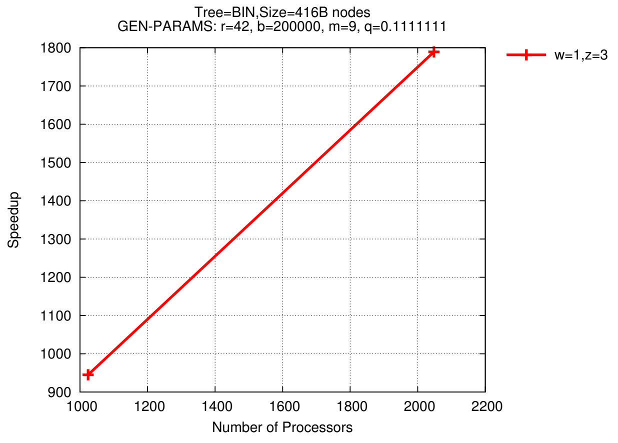

Figure 6. UTS speedup achieved for a 416 billion node BINOMIAL tree on Argonne National Lab's Blue Gene/P cluster. The speedup shown is based on the sequential performance of UTS, which is 0.54 million nodes per second. The parameter w indicates the number of random steals attempted before resorting to lifelines. The parameter z indicates the number of lifelines for each compute node. The UTS revision number used was 15277.

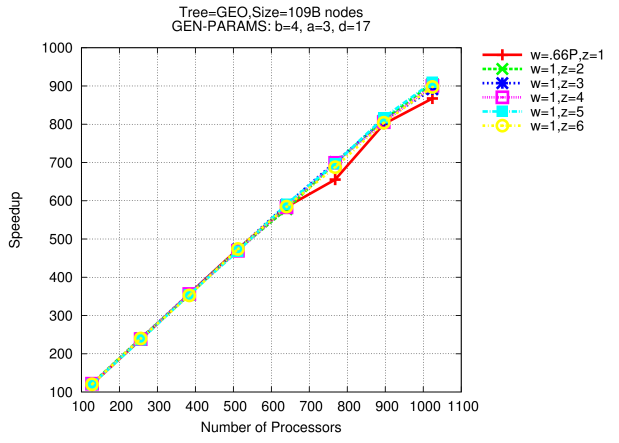

Figure 8. UTS speedup achieved for a 109 billion node GEOMETRIC tree on IBM's P7HV32 cluster. The speedup shown is based on the sequential performance of UTS of the same problem, which is 1.85 million nodes per second. The parameter w indicates the number of random steals attempted before resorting to lifelines. The parameter z indicates the number of lifelines for each compute node. The UTS revision number used was 15294.

On all platforms, we used X10 version 2.0.5. We used the C++ backend, which compiles[10] the X10 source code to C++ files which are then compiled into an executable using a standard C++ compiler. The generated C++ code was compiled with g++ v 4.3.2 on Triloka; g++ v 4.1.2 on Blue Gene/P and g++ v 4.3.4 (Power specific version) on P7HV32.

3.2 Discussion

Figure 1 and Figure 2 show the "lifestory" of a test run. We instrumented the application to record events corresponding to state transitions (i.e. transitions between *computing*, *stealing*, *distributing*, *probing* and *idle*) at each place. The start time for these lifestories was normalized to 0 to account for clock skew between the nodes. The histories were then merged in an order preserving way to determine the composite picture presented here. The picture shows at each instant the number of places that were computing, stealing, distributed or dead.

The two graphs are for different values of (w, z), viz. $(1, 3)$ and $(83, 1)$. Even though the throughput for both runs was the same, the graphs show variations in micro-structure. In particular, the computation with the high w value spends 22% more time stealing than the computation with the low w (not surprisingly) — the green area is substantially larger, and there is a visible dead area towards the end of the computation.[11] The graphs were fairly small and run on small number of cores; we expect the differences to be more pronounced with increase in number of cores.

Figure 3 shows speedup for the small x86 Triloka cluster. The implementation achieves 94% efficiency vs sequential execution for 128 cores. While not indicated here, similar performance curves were obtained for other combinations of w and z for this configuration. The speedups are measured with respect to sequential performance numbers. Since problem sizes run on large machines are typically large, it is impractical to obtain sequential performance numbers by running the same program in sequential mode. Therefore we compute sequential numbers by running the program on a smaller tree of the same basic shape. The actual parameters used to compute the sequential performance are given in Table 1.

Figure 4 presents the data for multiple executions of the 157B tree from [26] for different points in the (w, z) space. It can be seen that performance stays the same as w decreases dramatically, while z is increased modestly. At 1024 cores, the implementation delivers an efficiency of 87%. Figure 5 presents the data for this tree on the P7HV32 cluster, which also delivers 87% efficiency at 1024 cores. Note that the results for high-w ($(w, z) = (0.66 \times P, 1)$) are starting to get marginally worse than results for high-z.[12]

Figure 6 shows that the efficiency is maintained, albeit at a bigger tree size, as the core count is doubled.

Finally, Figure 7 and Figure 8 show speedup for a geometric tree of 109B nodes on the Blue Gene/P and P7HV32 clusters respectively with varying values of w and z. The implementation achieves an efficiency of 92% and 89% respectively. Again, high-w results tend to be slightly worse than high-z results.

4. Related and future work

A fundamental requirement for efficient parallel execution of a program is that work (both computational and otherwise) be evenly divided amongst the available computing resources; that is, the program's execution must be well load balancing. Many applications are *regular*; that is, these applications' computations and the related data can be partitioned *apriori*; regular programs can be efficiently load balanced statically. However, there are many applications which are irregular and dynamic; that is, the computations and the data sets in these applications cannot be partitioned *a priori*. Irregular applications are extremely sensitive to load balancing, and place unique requirements on parallel programming tools and runtimes that have not yet been satisfied. Till now, the best solutions for efficient execution of irregular applications on distributed-memory systems have heavily involved application-level dynamic load balancing.

Some of the early research on dynamic load balancing for shared-memory systems was carried out in the Mul-T Scheme [23] project; load balancing was achieved through lazy task creation wherein threads would steal work from one another when they ran out of work. Cilk [15], an extension to the C language, was the first system to provide efficient load balancing for a wide variety of irregular applications. Load balancing in Cilk applications is achieved by a scheduler that follows the *depth-first work, breadth-first steal* principle [5]. Cilk's scheduling policy, in which each thread of execution maintains its own set of tasks, and steals from other threads on a need-by basis, is often dubbed as *work-stealing*. Currently, there are many solutions for task parallelism that offer work-stealing schedulers. Of these, OpenMP 3.0 [27], Intel's Threading Building Blocks (TBB) [29], Microsoft's Parallel Patterns Library (PPL), and Task Parallel Library (TPL) are the most popular. The other variants of Cilk-like work-stealing in today's parallel frameworks include X10's *breadth-first* [7] work scheduling policy and Guo et al.'s [17] hybrid model for work stealing. Kambadur et al. [19] show that different work-stealing strategies are required for different irregular applications, and present a library framework to that allows easy customization of load balancing policies on shared memory. Berenbrink et al. [3] prove that work-stealing, by virtue of taking a distributed approach to load balancing processors, is stable.

The work-stealing approaches mentioned above efficiently and dynamically load balances applications in a shared-memory environment. However, these approaches are not directly applicable to distributed-memory machines because of a variety of issues such as network latency and bandwidth, and termination detection. Various bodies of work have addressed the problem of dynamic load balancing on distributed-memory machines. Grama et al. [16, 20] discuss various load balancing strategies for distributed parallel searches that are independent of the specific search technique. Blumofe et al. [6] adapt the Cilk work-stealing model to distributed shared memory by limiting the scope of the programs to be purely functional. ATLAS [1] and Satin [32] both use hierarchical work-steal to acheive global load balancing. Charm++ [18, 31] monitors the execution of its distributed programs for load imbalance and migrates computation objects to low-load places to correct the load imbalance. Global load balancing for message passing environ-

[10] The X10 compiler `x10c++` was given the command line arguments `-O -NO_CHECKS` to enable optimizations and disable array bounds and null pointer checking

[11] For $(w, z) = (83, 1)$, 1.835% of the total time was spent stealing, as opposed to 1.43% for $(w, z) = (1, 3)$.

[12] High w runs correspond to unrestricted random work-stealing. High z runs (with $w = 1$) correspond to more extensive use of lifelines.

ments has been researched for specific problems by Batoukov and Sorevik [2].

UTS, an excellent example of an irregular application, was first described by Prins et al. [28]. Although they tried various work-stealing strategies, they were unable to get good speedups on distributed-memory systems on their initial UPC [13] implementation; however, their implementation showed good performance on shared-memory machines. Later, Olivier et al. [24] formally introduced UTS as a benchmark to measure a parallel system's ability to dynamically load balance an application. Similar to [28], their OpenMP and UPC implementations showed good speedup on shared-memory systems, but not on distributed-memory systems. Dinan et al. [12] studied the ability of MPI [21, 22] to support dynamic load balancing required by UTS through application-level load balancing techniques. In this study, both centralized work-sharing and distributed work-stealing approaches were tried, and it was shown that work-stealing approaches with the right stealing granularity performed better on a wider variety of workloads than work-sharing. Olivier and Prins [26] provided the first scalable implementation of UTS on clusters that provided up to 80% efficiency on 1024 nodes. To this end, they employed a custom application-level load balancer along with optimizations such as one-sided communications and novel termination detection techniques. Finally, Dinan et al. [11] provide a framework for global load balancing, which was used to achieve speedups on 8196 processors. Global load balancing and termination detection facilities were provided to express irregular applications. By reserving one core per compute node on the cluster exclusively for lock and unlock operations, this framework allowed threads to steal work asynchronously without disrupting the victim threads. However, the cost paid was a static allocation (one core out of every eight) for communication. This results in lower throughput because the thread is not available for user-level computations.

4.1 Future work

Develop an analytical framework for lifeline graphs. It appears to us that random work-stealing and lifeline distribution graphs are two sides of the same coin. We believe that it should be possible to develop an analytical framework for lifeline graphs which can predict the performance (efficiency) of implementations with given values for parameters.

Some initial experimentation we have done leads to the possibility that high-z configurations deliver better performance at high processor counts with lower CPU utilization. Lower CPU utilization is of value in cloud-based multi-tenancy installations. For the 157B tree at 2048 cores, we observe 820 M Nodes/s for $(w, z) = (1024, 1)$, and 828 M Nodes/s for $(10, 10)$. (The $(10, 1)$ performance is 797 M nodes/s). Also as Figure 8 shows, at higher core counts high-w runs tend to be worse than high-z runs. We plan to run further experiments to investigate this phenomenon.

Implement adaptive stealing. [7] shows that the performance of graph algorithms can be improved in a shared memory context by using work-stealing with adaptive grain size. Workers add chunks of tasks to the deque; a thief steals one chunk at a time. The size of the chunk is determined by the worker based on the current size of the deque. A large deque means there is less demand for work, hence a larger chunk can be built. We believe these ideas can be adapted fruitfully to global load balancing.

Scaling out finish. We are interested in scaling computations to hundreds of thousands of cores. For this the current X10 implementation of `finish` must be restructured to use reduction trees and engineered to work at this scale.

Acknowledgements

We gratefully thank the many members of the X10 team who have contributed to several discussions related to these ideas, in particular David Cunningham, George Almasi, Olivier Tardieu and Igor Peshansky. The design and implementation of the `finish` scheme in the X10 implementation is due to Olivier Tardieu.

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. Atlas: an infrastructure for global computing. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 165–172, New York, NY, USA, 1996. ACM.

[2] R. Batoukov and T. Sorevik. A Generic Parallel Branch and Bound Environment on a Network of Workstations. In *HiPer '99: Proceedings of High Performance Computing on Hewlett-Packard Systems*, pages 474–483, 1999.

[3] P. Berenbrink, T. Friedetzky, and L. A. Goldberg. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, 32(5):1260–1279, 2003.

[4] S. M. Blackburn, R. L. Hudson, R. Morrison, J. E. B. Moss, D. S. Munro, and J. Zigman. Starting with termination: a methodology for building distributed garbage collection algorithms. *Aust. Comput. Sci. Commun.*, 23(1):20–28, 2001.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, 1994.

[6] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 1997. USENIX Association.

[7] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 536–545, Washington, DC, USA, 2008. IEEE Computer Society.

[8] J. Dean, S. Ghemawat, and G. Inc. Mapreduce: simplified data processing on large clusters. In *In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, pages 137–150. USENIX Association, 2004.

[9] E. Dijkstra and C. Scholten. Termination detection for diffusing computations. In *Information Processing Letters*, volume 11, pages 1–4, 1980.

[10] E. W. Dijkstra. Derivation of a termination detection algorithm for distributed computations. pages 507–512, 1987.

[11] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[12] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic Load Balancing of Unbalanced Computations

Using Message Passing. In *IPDPS 07: Parallel and Distributed Processing Symposium*, pages 1–8, Long Beach, CA, March 2007. IEEE International.

[13] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. *UPC Language Specification*, 1.1 edition, May 2003. `http://www.gwu.edu/~upc/downloads/upc_specs_1.1p2pre1.pdf`.

[14] N. Francez and M. Rodeh. Achieving distributed termination without freezing. *IEEE Trans. Softw. Eng.*, 8(3):287–292, 1982.

[15] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published in ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[16] A. Grama and V. Kumar. State of the Art in Parallel Search Techniques for Discrete Optimization Problems. *IEEE Trans. on Knowl. and Data Eng.*, 11(1):28–35, 1999.

[17] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[18] L. V. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of Object Oriented Programming Systems, Languages and Applications, ACM Sigplan Notes*, volume 28, pages 91–108, 1993.

[19] P. Kambadur, A. Gupta, A. Ghoting, H. Avron, and A. Lumsdaine. PFunc: Modern task parallelism for modern high performance computing. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC)*, Portland, Oregon, November 2009.

[20] 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.

[21] Message Passing Interface Forum. MPI, June 1995. `http://www.mpi-forum.org/`.

[22] Message Passing Interface Forum. MPI-2, July 1997. `http://www.mpi-forum.org/`.

[23] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):264–280, 1991.

[24] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: an unbalanced tree search benchmark. In *LCPC'06: Proceedings of the 19th international conference on Languages and compilers for parallel computing*, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[25] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. Uts: an unbalanced tree search benchmark. In *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, LCPC'06, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[26] S. Olivier and J. Prins. Scalable dynamic load balancing using UPC. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 123–131, Washington, DC, USA, 2008. IEEE Computer Society.

[27] OpenMP Architecture Review Board. *OpenMP Application Program Interface, v3.0*. May 2008.

[28] J. Prins, J. Huan, B. Pugh, C.-W. Tseng, and P. Sadayappan. UPC Implementation of an Unbalanced Tree Search Benchmark. Technical Report 03-034, University of North Carolina at Chapel Hill, October 2003.

[29] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, 2007.

[30] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The asynchronous partitioned global address space model. In *AMP'10: Proceedings of The First Workshop on Advances in Message Passing*, June 2010.

[31] A. B. Sinha and L. V. Kalé. A load balancing strategy for prioritized execution of tasks. In *IIPS'93: Proceedings of International Parallel Processing Symposium*, pages 230–237, 1993.

[32] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43, New York, NY, USA, 2001. ACM.