Solving Large, Irregular Graph Problems using Adaptive Work-stealing

Guojing Cong (IBM)
Sreedhar Kodali (IBM)
Sriram Krishnamoorthy (Ohio State)
Doug Lea (SUNY Oswego)
Vijay Saraswat (IBM)
Tong Wen (IBM)
Contact email: vsaraswa@us.ibm.com

Abstract

Solving large, irregular graph problems efficiently is challenging. Current software systems and commodity multiprocessors do not support fine-grained, irregular parallelism well. We present XWS*, the* X10 *Work Stealing framework, an open-source runtime for the parallel programming language* X10 *and a library to be used directly by application writers.* XWS *extends the Cilk work-stealing framework with several features necessary to efficiently implement graph algorithms, viz., support for improperly nested procedures, global termination detection, and phased computation. We also present a strategy to adaptively control the granularity of parallel tasks in the work-stealing scheme, depending on the instantaneous size of the work queue. We compare the performance of the* XWS *implementations of spanning tree algorithms with that of the hand-written C and Cilk implementations using various graph inputs. We show that* XWS *programs (written in Java) scale and exhibit comparable or better performance.*

1 Introduction

Obtaining practical efficient implementations for large, irregular graph problems is challenging. Current software systems and commodity multiprocessors do not support fine-grained, irregular parallelism well. Implementing a custom framework for fine-grained parallelism for each new graph algorithm is impractical.

We present XWS, the X10 Work Stealing framework. XWS is intended as an open-source runtime for the programming language X10 [10], a partitioned global address space language supporting fine-grained concurrency. XWS is also intended as a library to be used directly by application writers. XWS extends Cilk work stealing [2, 4] with several features necessary to efficiently implement graph algorithms, viz., support for improperly nested procedures, worker-specific data-structures, global termination detection, and phased computation.

We present simple elegant programs using XWS for different spanning tree algorithms using (pseudo-)depth-first search and breadth-first search. We evaluate these programs on a 32-way Niagara (moxie), and an 8-way Opteron server (altair) and on three different bounded-degree graphs: (i) graphs with randomly selected edges and (a) no degree restrictions (b) fixed degree, and (ii) planar torus graphs.

We show the performance of BFS and pseudo-DFS search depends crucially on the granularity of parallel tasks. We show that the granularity natural to the algorithms – the examination of a single edge — leads to poor performance at scale. Instead, sets of of vertices must be grouped into *batches*. We show that a fixed-size batching scheme does not perform well. For instance, batches of size 1 yield a peak performance of 20 MEPS (Million Edges Per Second) on Niagara. Instead we develop an adaptive batching scheme in which the batch size is sensitive to the instantaneous size of the work queue. With this scheme, pseudo-DFS shows linear scaling on Niagara and Opteron, achieving peak performance of over 220 MEPS on moxie and substantially outperforming C and Cilk implementations.

1.1 Challenges in solving large, irregular graph problems

The last few years have seen an explosion of mainstream architectural innovation — multi-cores, symmetric multiprocessors, clusters, and accelerators (such as the Cell processor and GPGPUs) — that now requires application programmers to confront varied concurrency

and distribution issues. This raises the fundamental question: what programming model can application programmers use to productively utilize such diverse machines and systems?

Consider for instance the problem faced by designers of graph algorithms. Graph problems arise in traditional and emerging scientific disciplines such as VLSI design, optimization, databases, computational biology, social network analysis, and transportation networks.

Large-scale graph problems are challenging to solve in parallel – even on shared memory symmetric multiprocessor (SMP) or on a multicore system – because of their irregular and combinatorial nature. Irregular graphs arise in many important real world settings. For random and "scale-free" graphs [3] no known efficient static partitioning techniques exist, and hence the load must be balanced dynamically.

Consider the spanning tree problem. Finding a spanning tree of a graph is an important building block for many graph algorithms such as those for biconnected components, ear decomposition [9] and graph planarity testing [8]. Spanning tree represents a wide range of graph problems that have fast theoretic parallel algorithms but no known efficient parallel implementations that achieve speedup without serious restrictive assumptions about the inputs.

Bader and Cong [1] presented the first fast parallel spanning tree algorithm that achieved good speedups on SMPs. Their algorithm is based on a graph traversal approach, and is similar to DFS or BFS. There are two steps in the algorithm. First a small stub tree of size $O(p^2)$ is generated by one of the p worker through a random walk of the graph. The vertices of this tree are then evenly distributed to each worker. Each worker then traverses the graph in a manner similar to sequential DFS or BFS, using efficient atomic operations (e.g. Compare-and-Swap) to update the state of each node (e.g. update the `parent` pointer). The set of nodes being worked on is kept in a local queue. When a worker is finished with its portion (its queue is empty), it checks randomly for any other worker with a non-empty queue, and "steals" a portion of the victim's work for itself.

For efficient execution, it is very important that the queue be managed carefully. For instance, the operation of adding work (a node) to the local queue should be efficient (i.e. should not require locking) since it will be performed frequently. Stealing is however relatively infrequent and it is preferable to shift the cost of stealing from the victim to the thief since the thief has no work to do (the "work first" principle). The graph algorithm designer now faces a choice. The designer may note [1] that correctness is not compromised by permitting a thief to *copy* the set of nodes that the victim is working on. Here the victim is permitted to write to the queue without acquiring a lock. Now the price to be paid is that the thief and the victim may end up working on the same node (possibly at the same time). While work may thus be duplicated, correctness is not affected since the second worker to visit a node will detect that the node has been visited (e.g. because its atomic operation Compare-and-Swap will fail) and do nothing. Alternatively, the designer may use a modified version of the Dekker protocol [2] to resolve the race condition. This guarantees that no work will be duplicated, but the mechanism used is very easy to get wrong, leading to subtle concurrency errors.

The above illustrates that the design of such high-performance concurrent data-structures is difficult and error-prone. This suggests packaging the required components in a library or a framework and exposing a higher-level interface to programmers.

1.2 X10

The X10 programming language [10] has been designed to address the challenges of "productivity with performance" on diverse architectures. X10 augments a core sequential modern object-oriented language (very similar to Java or Scala) with constructs for distribution (*places*) and concurrency (*asyncs*, *finish*, *atomic* and *clocks*).[1] The statement `async S` spawns a new task or activity to execute the statement `S`. The statement `finish S` executes `S` and waits for all activities spawned during its execution to terminate. The statement `atomic S` causes the statement `S` to be executed as if in a single indivisible step (with no other activity executing during this step). An X10 clock is a data-structure that represents a dynamic barrier. The activity creating a clock is said to be registered on that clock. An activity that is registered on a clock `c` may create new activities registered on `c` by executing `async clocked(c) S`. An activity registered on a clock may execute a `next` operation to suspend until such time as all activities registered on the clock have executed a `next` operation (barrier behavior).

X10 may be used to implement spanning tree computations in a very straighforward way:

Example 1.1 (Psuedo-DFS in X10) The parallel exploration of a graph may be implemented thus:

```
1. class V  {
2.   V [] neighbors;
3.   V parent;
4.   V(int i){super(i);}
5.   boolean tryColor(V n) {
6.     atomic if (parent ==null) parent=n;
7.     return parent==n;
```

[1] Discussion of the distribution constructs of X10 and advanced concurrency constructs (`when`) is out of scope of this paper.

```
8.   }
9.   void compute() {
10.     for (V e : neighbors)
11.       if (e.tryColor(this))
12.             async e.compute();
13.   }
14.   void dfs() {
15.     parent = this; // root is visited.
16.     finish compute();
17.   }}
```

Computation is initiated by invoking `r.dfs()` on the root vertex `r`. This visits `r` within the scope of a `finish`. When a vertex is visited (its `compute()` method invoked), each of its outgoing edges is examined in sequence. If the vertex reached from the edge does not have its parent set, its parent is set atomically, and the vertex is (recursively) visited asynchronously. Thus an activity is spawned for each vertex in the connected component containing `r`. (This code does not implement batching, a batched version is discussed later.) □

We note that this program cannot be written in Cilk without excessive synchronization. Cilk enforces a "fully strict" condition: in X10 terminology this condition requires that if the body of a procedure spawns an `async`, then it must contain a `finish` enclosing that `async`. Thus to write this program in Cilk an extra `finish` must be wrapped around the `for` loop on Line 10. This introduces needless extra synchronization which reduces performance.

Example 1.2 (BFS in X10) The breadth-first parallel exploration of a graph may be implemented as follows:

```
1. class V  extends VertexFrame {
2.   V [] neighbors;
3.   V parent;
4.   V(int i){super(i);}
5.   boolean tryColor(V n) { ... }
6.   void compute(clock c) {
7.     for (V e : neighbors)
8.       if (e.tryColor(this))
9.         async clocked(c) {
10.            next;
11.            e.compute(c);
12.         }}
13.  void bfs() {
14.    parent=this;
15.    finish async {
16.     clock c = new clock();
17.     compute(c);
18.    }}}
```

The code differs from DFS in that all asyncs are clocked. A new node is visited only after the clock advances (Line 10-11). Hence all vertices are visited in breadth-first order. □

1.3 XWS

A central challenge in implementing X10 is to efficiently load-balance multiple activities, while respecting dependencies introduced by clocks and `finish`. This paper presents the design, implementation and evaluation of a portion of the X10 runtime system for multicore and SMPs, XWS. XWS implements fine-grained concurrency through an extension of Cilk Work Stealing (CWS) [2, 4]. XWS extends CWS to better support the programming of applications with irregular concurrency. It removes the link between recursion and concurrency introduced by fully strict Cilk model. Crucial to this removal is a method in XWS for detecting termination of a computation without counting all the frames created during the computation. Further, XWS integrates barriers – essential for phased computations such as breadth-first search – with work stealing. Finally, XWS support the implementation of *adaptive batching* schemes by the programmer. Batching is a technique for increasing the granularity of parallel tasks by batching together several small tasks. Thieves steal a batch at a time. Depending on the algorithm, the batching size may have a dramatic impact on the performance of work stealing. XWS permits the (X10 or XWS) programmer to sense key metrics of the current execution and use these to adjust batching size dynamically.

1.4 Rest of this paper

The rest of this paper is as follows. In Section 2, we present the details of the design of XWS. In Section 3 we examine comparable programs written using an application-specific framework (Simple, [1]), as well as Cilk, and compare performance on three different graph inputs. Our graph generators include several employed in previous experimental studies of parallel graph algorithms for related problems. For instance, we include the torus topologies, random graphs and geometric graphs, used in [5], [7] and others.

- **2D Torus** The vertices of the graph are placed on a 2D mesh, with each vertex connected to its four neighbors.
- **Random Graph** We create a random graph of n vertices and m edges by randomly adding m unique edges to the vertex set. Several software packages generate random graphs this way.
- **Geometric Graph** In these k-regular graphs, each vertex is connected to its k neighbors.

We show that the performance of these programs in XWS can be substantially improved with batching. We present schemes for adaptively determining the size of the batch based upon an estimate of the current stealing pressure.

Finally we conclude with a section on related work and acknowledgements. Due to the page

limit, we can not provide more examples on how to use the XWS framework. Readers please refer to the Java implementation of XWS for details, available at `http://x10.sf.net`, in the module `x10.runtime.xws`.

2 X10 work stealing

This section presents the design of the XWS API, focusing on the application programmer who is directly programming to the API. The discussion of the techniques used by the X10 compiler to map X10 constructs to XWS API calls (such as the introduction of "slow" and "fast" variants of methods, *a la* Cilk) are beyond the scope of this paper.

2.1 The XWS API

XWS exposes the following mechanisms to the application programmer: (i) create a *pool* of workers (`new Pool(N)`) (ii) submit a job (containing an activity) to the pool (`pool.submit(job)`), (iii) submit a new activity `t` dynamically (`w.pushFrame(t)`, `w.pushFrameNext(t)`, where `w` is the current worker), (iv) remove the current activity (`w.popFrame()`), and (v) wait for children activities to terminate (`w.sync()`). XWS uses work stealing to schedule these activities dynamically on the workers in the given pool.

The programmer initiates computation by creating a *pool* of N workers (threads). Computation is initiated by submitting a *job* to the pool. The job contains a reference to a frame F (see below), representing the top-level activity. The thread submitting the job suspends until F is executed to completion (thus job submission implies an implicit "global" `finish`). On return, the thread may submit additional jobs to the pool. In the current implementation, at most one job is permitted to be active in a pool at any time.

Activities are represented by subclasses of `Frame`. Subclasses must implement a a `void compute(Worker w) throws AbortOnSteal` method with the code representing the body of the activity. (The `AbortOnSteal` exception is discussed below.) Typically, the subclass will also contain application-specific fields that store the values of local variables in the procedure body in which the activity is created.

Each worker `w` maintains an internal *deque* (double-ended queue) of frames. The programmer creates a new task by allocating a new frame and initializing its state. A task `t` can be added to the deque of the current worker `w` by invoking `w.pushFrame(t)`. On completion, the task may be popped from the deque by invoking `w.popFrame()`. These calls are inserted automatically by the X10 compiler, but must be inserted manually by the XWS programmer.

XWS implements the basic CWS strategy. Per this strategy, each worker executes a basic *scheduling* loop. In this loop, the worker W attempts to acquire a task (if it does not have one already) by randomly choosing another worker V and attempting to remove (steal) a task from the top of its deque. This attempt will succeed only if V has at least two items on its deque. If successful, the task is transferred to W's dequeue and executed. Otherwise the worker continues trying to steal from the next worker after V.

Like Cilk, XWS uses *closures* to permit values to be returned from asynchronous tasks. A closure contains a frame and is used to return values in the case of stealing. There is a one-to-one correspondence between a frame and a task, but there may be multiple closures associated with a task. Each time a task is stolen, a closure is created. (We omit the details for lack of space.)

XWS uses a modified Decker protocol [4] to implement stealing. Each worker is associated with a lock. Before attempting to take an element from the victim's deque, the thief must acquire the victim's lock (this also locks out other thieves trying to mug the same victim). However, the victim is not required to obtain this lock except when it detects a theft has happened (by using a Dekker protocol). This ensures that the fast path computation (the victim pushing and popping tasks from its deque) does not involve obtaining a lock unless a theft is in progress.

Task stashing. To implement the submission of a new task, the programmer may use one of two techniques. In the "activity-as-subroutine" technique, the worker `w` descends into the body of the new task T, through a subroutine call. Before that, it must push a task C onto its deque representing the "continuation": the code that would be executed on return from T. While `w` is busy executing T, some other worker may steal C from `w`'s deque. Hence on return from T, the programmer must ensure that the worker checks whether the current frame has been stolen by executing `w.abortOnSteal(x)`, where `x` is the value returned by T. If the current frame has indeed been stolen, this call will cause the value `x` to be stashed in a way in which it is routed to the closure associated with C, and will throw an `AbortOnSteal` exception that will pop all the (now useless) activation frames on the call stack. The exception is caught by the scheduling loop for the worker, which now returns to the top of the loop (looking for more work).

This technique has the drawback that it requires the programmer to implement continuations (in Java this means keeping a "program counter" field in the frame

and jump tables; sometimes the normal flow of sequential control has to be altered since Java does not permit goto's). It has the advantage that it preserves the "busy leaves" property which establishes a good space bound on the execution of the program (see [2, 4]).

In the dual "task stashing" technique, `w` pushes T onto its deque and continues executing C. No continuations need to be implemented – hence this is a very simple technique for the application programmer. (Unfortunately, Cilk's space bounds cannot be guaranteed with this technique.) This technique is particularly useful when implementing non fully-strict computations: a procedure call may now simply invoke `pushFrame` repeatedly to stash tasks on the deque and return. The scheduling loop must now be modified so that when control returns to it tasks are popped from the bottom of the deque and execute until the deque is empty. Additionally, before stashing tasks on the deque, the programmer must ensure that the current frame is popped so that it is not available to be stolen (recall that the top frame is available to be stolen as soon as there are two or more frames on the deque).

2.2 Global quiescence

In fully-strict computations completion of the first task and the return of the corresponding closure indicates termination. Improperly-nested tasks that do not require a return call chain can do away with closures. Hence, we need an alternate mechanism to identify termination.

In essence, the mechanism we have implemented detects the stable property "all deques are empty" by using a barrier, plus a counter `checkCount` that measures the number of workers with non-empty deques. Initially the count is 0. Whenever a worker checks out a job from the submission queue, it increments the count. Whenever a worker finds its deque is empty and starts stealing, it decrements the count. Whenever it successfully steals, it increments the count before releasing the lock on the victim (thus ensuring that the count remains positive).

Note an important property of this mechanism. Suppose a worker W_0 checks out a task from the submission queue. Its execution generates very few tasks which end up being executed by W_0. In this case the count will go up to 1 and then down to 0 when W_0's queue is empty, and the job will be considered completed. This is the case even though $P - 1$ workers have not participated in the barrier. Thus this mechanism does not require all workers to participate, only those that actually steal work.

We have now exposed enough of the XWS machinery to write the pseudo-DFS program in Java (using XWS):

Example 2.1 (Psuedo-DFS in XWS) The program uses the "task stashing" technique to handle new tasks. There is no need to represent continuations.

```
1. class V  extends VertexFrame {
2.   V [] neighbors;
3.   V parent;
4.   V(int i){super(i);}
5.   boolean tryColor(V n) { ... }
6.   void compute(Worker w) throws StealAbort {
7.     w.popFrame();
8.     for (V e : neighbors)
9.       if (e.tryColor(this))
10.         w.pushFrame(e);
11.  }
12.  void dfs() {
13.    parent=this;
14.    compute((Worker) Thread.currentThread());
15.  }}
```

Since Java does not have atomics, the implementation of `tryColor` is changed to use an appropriate atomic method from Java concurrency utils (code omitted). Since global quiescence detection is used, the `finish` in the body of `dfs()` does not need to be implemented. □

2.3 Phased computations

We also added support for phased computations in which tasks in this phase create tasks to be executed in the next phase (cf BFS search). Phased computations are supported as a generalization of global quiescence. Each worker maintains two dequeues (the *now* deque and the `next` deque). Depending on the phase specified when spawning tasks, a task can be added to the now deque (`w.pushFrame(t)`) or the next deque (`w.pushFrameNext(t)`).[2]

When global quiescence is detected for the current phase, the barrier action steps the computation to the next phase. Each worker keeps track of the phase number it thinks it is in. After each round of stealing, it checks to see if the barrier's phase is the same as its phase; if not, it advances the phase and swaps its next and now deques. When checking into the barrier, each worker specifies whether it has work to do in the next phase. When the barrier is advanced `checkCount` is initialized with the number of workers with work to do in this phase, thus maintaining the invariant associated with the barrier. If this count is 0, the job is terminated.

Note that this design permits workers to *jump phases*. A worker W_i may finish computation in phase k and start searching for work. Meanwhile other workers may

[2] We note in passing that `pushFrameNext` is not adequate to implement all the functionality of X10's clocks. In essence, only clocked activities whose first action is to execute `next` (cf Example 1.2[Line 10]) can be implemented through such a call. The compiler must generate continuation-passing code to implement all the functionality of clocks.

check into the barrier causing it to move to phase $k+1$. This phase may contain very little work, and the barrier may trip repeatedly reaching phase $k+m$, before W_i discovers the phase has advanced and updates its phase to $k+m$. (The algorithm design ensures that W_i may skip phases only if its next deque is empty.)

Example 2.2 (BFS in XWS) The breadth-first parallel exploration of a graph may be implemented as follows. The only change with the DFS code is in Line 10, where a call to `pushFrameNext` is used.

```
1. class V  extends VertexFrame {
2.   V [] neighbors;
3.   V parent;
4.   V(int i){super(i);}
5.   boolean tryColor(V n) { ...}
6.   void compute(Worker w) throws StealAbort {
7.     w.popFrame();
8.     for (V e : neighbors)
9.       if (e.tryColor(this))
10.         w.pushFrameNext(e);
11.  }
12.  void dfs() {
13.    parent=this;
14.    compute((Worker) Thread.currentThread());
15.  }}
```

□

3 Graph algorithms in XWS

We consider implementations of breadth-first and depth-first search for spanning trees of a graph.

3.1 SIMPLE implementation (C)

The C implementation of DFS works as follows. First a small stub-tree of size $O(p^2)$ is generated by one thread randomly walking the graph starting from the root. The vertices encountered in the walk are evenly distributed into the stacks of the p threads. Each thread then starts traversing the graph in DFS order. To prevent race conditions during DFS traversals, lock-free protocols are used to guard against multiple threads coloring the same vertex. For load-balancing, a thread attempts to steal a piece of the stack from another thread in case it runs out of work. As a heuristic, each steal takes one half of the stack from the victim. Note that in order to reduce the cost of stealing, no synchronization is invoked in such steals. The steals may get stale values, yet the correctness is not jeopardized as the thread will later find all the stolen vertices have been visited. When all threads run out of work and there is no work to steal, the algorithm halt. The stacks and their top and bottom pointers are declared as volatile, and each push/pop operation involves operations on volatiles.

The C implementation of parallel BFS follows the SPMD programming paradigm. The expansion of the BFS frontier is implemented as follows. Each thread keeps a local queue, and gets an equal portion of vertices in the current frontier. In the beginning, there is only one vertex (root) in the frontier, and only one thread has an non-empty local queue. After draining the current frontier vertices in the queue, new frontier vertices are placed inside the queue. When all threads are done with the current frontier (guaranteed with a barrier), the newly-added vertices in the queues are merged together to form the new global frontier. Repetitive appearances of nodes in the frontier is allowed. The algorithm iterates until no new frontier vertices are discovered.

3.2 Cilk implementation

The Cilk implementation of parallel DFS can be derived easily from the sequential recursive DFS. Instead of sequential traversal, we spawn a parallel DFS traversal for each unvisited child of the current vertex. To guard against race conditions among traversals, the access to the color of a vertex is protected by lock-free synchronizations. Load-balancing is handled by the Cilk runtime library. However, as discussed earlier Cilk forces each procedure to wait for termination of all its children spawned during its execution.

In the sequential implementation of BFS, iteration is over the vertices in the current frontier. Their unvisited neighbors are then used to form the next frontier. To achieve scalability in Cilk, the iteration has to be performed in a divide-and-conquer recursive way so as to utilize the Cilk work-stealing scheduler. The current frontier is partitioned into blocks and the base case in the recursion is the iteration over a single block. As in the C implementation, repetitive appearances of vertices in a frontier is allowed. The algorithm iterates until the next frontier is empty.

3.3 XWS implementation:Batching

Algorithms for irregular graph problems are in general not directly amenable to divide-and-conquer recursive decomposition. However, we can still approximate the properties that make work-stealing perform well for these problems.

To do this, we first require compact task descriptions. The size of a task description representing exploration starting at each of k nodes should be constant, and independent of k. Otherwise, the communication overhead of pushing and stealing nodes would overwhelm processing, especially in algorithms such as spanning trees, where the per-node costs merely amount to marking nodes and labelling their parents. We address this by building up lists of work via simple linking: Each node enqueued in the work-stealing queue is the head

of a singly linked list possibly containing other nodes as well. The ordering of this list matters only in terms of memory locality and interference with other threads, which favors simple stack-based linking.

We next ask, what value of k should be used to batch a set of unprocessed nodes. For any given node in an arbitrary graph, we cannot know the value that will maximize aggregate throughput. One choice is to empirically choose some fixed value. However, the use of any fixed value would be too large during start up (stalling all but the initial thread), and/or too small during steady state. We can do better by first characterizing the best values to use at boundary points:

- A queued root node represents all of the work in the graph, so requires $k = 1$.
- If processing has nearly completed, and all remaining nodes are dead-ends (i.e., leading to no further expansion) choosing the best value of k is the counterpart to choosing the sequential threshold of a divide-and-conquer algorithm. This value, S, is an empirical threshold relating algorithmic work versus framework overhead.

Unless the per-node costs of an application are high enough to dictate that $S = 1$ (which is not the case for spanning tree algorithms), a rule that causes k to vary from 1 at roots to S at collections of dead-ends will provide better load balance and throughput than one that would use a fixed value. For some special graph types, it is possible to determine a fixed function of this form. For example, if the graph were actually a balanced tree, k should increase exponentially from the root to the leaves. However, in an arbitrary graph, any approach based on distance from roots would be prone to arbitrarily poor estimates. Instead, each task may use its current work-stealing queue depth to help estimate global progress properties: If the queue is empty, then even a single node placed on it is potentially valuable to some other thread trying to steal it and further expand. Conversely, if the queue is very large, then other threads must be busy processing other nodes, and any newly discovered node is less likely to lead to further expansion. Using a simple bounded exponential growth function (here, powers of two) across these points maintains scaling properties: Each of the 2^j nodes in a batch of a size-j queue (for $j \leq log_2(S)$) should have 2^{-j} of the expected expansions as does the single node in a size 1 queue. The choice of base two exponents is not entirely forced here, and different constants might be better for some graph types. However, the choice does coincide with the scaling and throughput properties of work-stealing in the case of divide-and-conquer over balanced binary trees, and adaptively approximates this case by dynamically varying batch sizes based on differential steal rates.

The resulting basic algorithm is a simple variant of the DFS algorithm presented in Section 1: Each task accepts a list headed by one of its nodes. For each node, it labels and expands the edges into a new list, pushing that list onto work-stealing queue when its size exceeds $min(2^Q, S)$, where Q is the queue-size. Notice that in the case of $S = 1$ (which might be used for algorithms with high per-node processing costs), this is identical to plain DFS.

Our adaptive DFS improves on the implementation of this idea by incorporating another common work-stealing programming technique. In classic divide-and-conquer programs, co-execution of two tasks a and b is best implemented by forking a, then directly running b, and then joining a. This saves the overhead of pushing and then immediately popping and running b. We adapt this idea here via some bookkeeping to swap lists rather than pushing and then immediately popping a new list when the original list is exhausted. The performance improvements stemming from this technique are always worthwhile, because they decrease overhead without changing any other algorithmic properties. However, as shown below, the extent of the improvement may vary dramatically across different graph topologies.

As is the case with any work-stealing algorithm, the value of S must be empirically derived. Thresholds are functions of per-node application costs (here, marking and setting spanning tree parents), as well as underlying per-task framework costs (mainly, work-stealing queue operations), as well as platform-level costs (processor communication, memory locality effects and garbage collection), along with interactions among these, and so resist simple analytic derivation. However, each of these component factors are properties of the program, and not, in general, its inputs (i.e., the actual graphs). As is the case for all work-stealing algorithms, choosing values of S larger than necessary will increase the variance of expected throughput: In some executions this may actually increase throughput due to less queue overhead, but in others, a too-large value will cause load imbalances, decreasing throughput. But sensitivity curves across these values are shallow within broadly acceptable ranges. We find that restricting values to powers of two suffices.

Choice of Threshold This choice of threshold was empirically guided by comparing performance across powers of two. The impact of this choice varies across graph types. Normalizing to 1.0 for S of 128, Table 1 shows throughput differences for graphs of 4 million nodes. The best value of S indicates that XWS framework overhead is low enough so that it is profitable to

S	Niagara T	K	R	Opteron T	K	R
1	0.58	0.79	0.81	0.18	0.54	0.55
2	0.68	0.85	0.85	0 33	0.78	0.81
8	0.88	0.93	0.94	0.75	0.97	0.93
32	0.97	1.00	0.99	0.82	0.98	0.94
128	1.00	1.00	1.00	1.00	1.00	1.00
512	0.98	0.92	1.00	0.89	1.00	0.99
2048	0.96	0.92	0.91	0.86	0.97	0.97

Table 1. Relative performance across thresholds

parallelize even batches of only a 100 or so dead-end nodes. The drop-off beyond 128 is very shallow, so larger values could have been used with almost no loss. However, choosing thresholds nearer the lower range of estimated maxima reduces run-to-run variability.

While adaptive batching improves performance over DFS (equivalent to $S = 1$) across graph types, the extent of the improvement varies considerably across graph types. This is due to two main factors, *locality* and *connectivity*.

Locality The graphs used in these experiments are too large to fit into processor caches. Thus, cache misses have a huge impact on performance. The Torus graph is laid out such that each node's row-wise neighbors will normally be adjacent to it in memory, and column-wise neighbors a constant stride away. Thus, traversals of a torus that improve search locality will improve throughput. This effect can be quantified by comparing the performance of simply accessing all of the nodes of the graph via all of its edges in some predefined locality-sensitive versus locality-insensitive order. We have found the relative improvement of a full scan of each edge of each node when performed in stride-1 index order of nodes versus a (wrapped around) stride of 7919 (a prime large enough to minimize cache hits) are much larger for the (4 dual) Opteron than for the (single multicore) Niagara (7.4, 1.3 and 1.4 for the Opteron (for T, K and R) and 2.2, 1.2 and 1.2 for the Niagara). This is due to the higher relative value of hardware prefetching across processors on the Opteron when locality prevails. These results independently indicate that the ability of adaptive batching to better preserve locality of access can be either a major or minor source of improvement, depending on graph layout. And for torus graphs, spanning tree construction throughput exceeds that of simple locality-insensitive traversal.

Connectivity For densely or regularly connected graphs, the ability of a task to swap in a partially created batch when its initial batch is exhausted increases the actual nodes processed per task, up from its nominal value of less than S, to the average number of nodes that may be traversed, with backup partial buffer size of at most S, before hitting a dead end. This value varies significantly across the three graph types we have investigated. For $S = 128$, the average values on the Niagara ranges from 150 for random graph, to 270 for k-graphs, to 2400 for Torus. (Opteron results are similar.) As the number of nodes per task increases, so does throughput: Bypassing the work-stealing queue reduces per-node overhead. Lower queue access rates in turn lead to lower contention for threads attempting to steal work. While these effects are secondary to others described above, they appear to account for the remaining throughput differences across graph types.

Construction of adaptive BFS in the style of our adaptive DFS encounters some new obstacles. BFS proceeds in phases; batching decisions must apply to next phase, not the current phase. Thus, decisions cannot rely on current queue state. Instead we employ a predictive strategy to control batch sizes. During each phase, each thread uses its estimate of average workload in the previous phase to control batch size. Although other choices are possible, we used for the experiments in Figure 1, constant multiples of the previous estimate, bounded by minimum and maximum sizes. This requires the multiplier to use as an empirically guided tuning parameter.

The results demonstrate improvement over non-adaptive versions, although less extreme than DFS. We believe that the differences in magnitude of effects are mainly due to three factors. First, because BFS requires phased computation, improvements are limited by underlying barrier synchronization rates (e.g. the Opteron's hardware prefetching does not come into play). Second, our batch size estimation strategy for BFS cannot be as sensitive to transient dynamic imbalances as DFS. And third, the BFS version does not enjoy as many of the added locality benefits of DFS in-place list-swapping. As further evidence of this third effect, the best tuning threshold was higher, and showed sharper sensitivity, on the Opteron MP than the Niagara multicore, which matches the locality and caching patterns discussed above.

We leave for future work the investigation of other adaptive rules which may yield better performance.

3.4 Results

We present performance data on two machines. Altair (Opteron) is an 8-way Sun Fire V40Z server, running four dual-core AMD Opteron processors at 2.4GHz (64KB instruction cache/core, 64KB data cache/core, 16GB physical memory). Moxie (Niagara) is a 32-way Sun Fire T200 Server running UltraSPARC T1 proces-

sor at 1.2 GHz (16KB instruction cache/core, 8KB data cache/processor, 2MB integrated L2 cache, 32GB physical memory). (We are in the process of benchmarking these programs on a 64-way Power5 SMP as well.)

We present results in Figures 1 for runs of BFS (KGraph, Random) and DFS (KGraph, Torus) on Opteron and runs of BFS (KGraph, Torus) and DFS (KGraph, Torus) on Niagara (y-axis: MEPS, x-axis: P), for 250K, 1M, 4M and 9M vertices.[3] We see that on Opteron XWS code is comparable with Cilk and C code for BFS (Torus not shown), but substantially outperforms them for DFS. On Niagara, XWS code substantially outperforms Cilk and C for all three graphs.[4]

4 Related work

Adaptive batching bears some similarities to the steal-half algorithm of [6], and its variants. Both approaches attempt to cope with non-hierarchical workloads for graph problems. In the steal-half algorithm, each node is queued as its own task; and thieves take half (or some other percentage) of the nodes available per steal attempt. In contrast, in our approach, the tasks are pre-batched, so only one batch is stolen at a time. This can substantially reduce queue overhead, contention and data movement costs, but comes with potential disadvantages because nodes cannot be stolen while they are being batched, and batches cannot be resplit. For example, our approach does not allow for a subset of the nodes from a stolen batch to themselves be re-stolen by other threads (as does steal-half). However, queue-sensing adaptation makes consequent impediments to global progress highly unlikely. Because we adaptively choose batch sizes so that there are always (during steady state processing) some nodes available to be stolen from each active thread, imbalanced progress by any one of them has little impact on the ability of others to find and steal new work. Additionally, by relating batching rules to sequential processing thresholds needed for any work-stealing program, our approach supports simpler empirically guided performance tuning.

5 Conclusion

In this paper we have shown how several graph algorithms can be expressed concisely and elegantly in X10. These algorithms rely heavily on support for fine-grained concurrency. The X10 runtime (XWS) implements fine-grained concurrency through an enhanced work-stealing scheduler. Specifically the scheduler supports improperly nested tasks, detection of global termination, and phased work-stealing. We measure the performance of spanning tree algorithms implemented with pseudo-depth-first search and breadth-first search on two multicore systems. We also present a strategy to adaptively control the granularity of parallel tasks in the work-stealing scheme. We show that the XWS programs scale and exhibit performance comparable with hand-written C programs.

[3] Note that the number of vertices in a torus is the square of the torus size.

[4] Several Cilk runs did not complete successfully and are hence omitted.

Acknowledgements We thank Raj Barik for his contributions to the implementation of the C++ version of XWS. We thank the rest of the X10 team for many discussions of these issues. This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico, Apr 2004.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[3] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proc. 4th SIAM Intl. Conf. on Data Mining*, April 2004.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[5] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proc. 6th Ann. Symp. Parallel Algorithms and Architectures (SPAA-94)*, pages 16–25, Cape May, NJ, June 1994.

[6] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 280–289, New York, NY, USA, 2002. ACM.

[7] T.-S. Hsu, V. Ramachandran, and N. Dean. Parallel implementation of algorithms for finding connected components in graphs. In S. N. Bhatt, editor, *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 23–41, 1997.

[8] P. Klein and J. Reif. An efficient parallel algorithm for planarity. *J. Comput. Syst. Sci.*, 37(2):190–246, 1988.

[9] G. L. Miller and V. Ramachandran. Efficient parallel ear decomposition with applications. Manuscript, UC Berkeley, MSRI, Jan. 1986.

[10] V. A. Saraswat. X10 Language Report. Technical report, 2004.

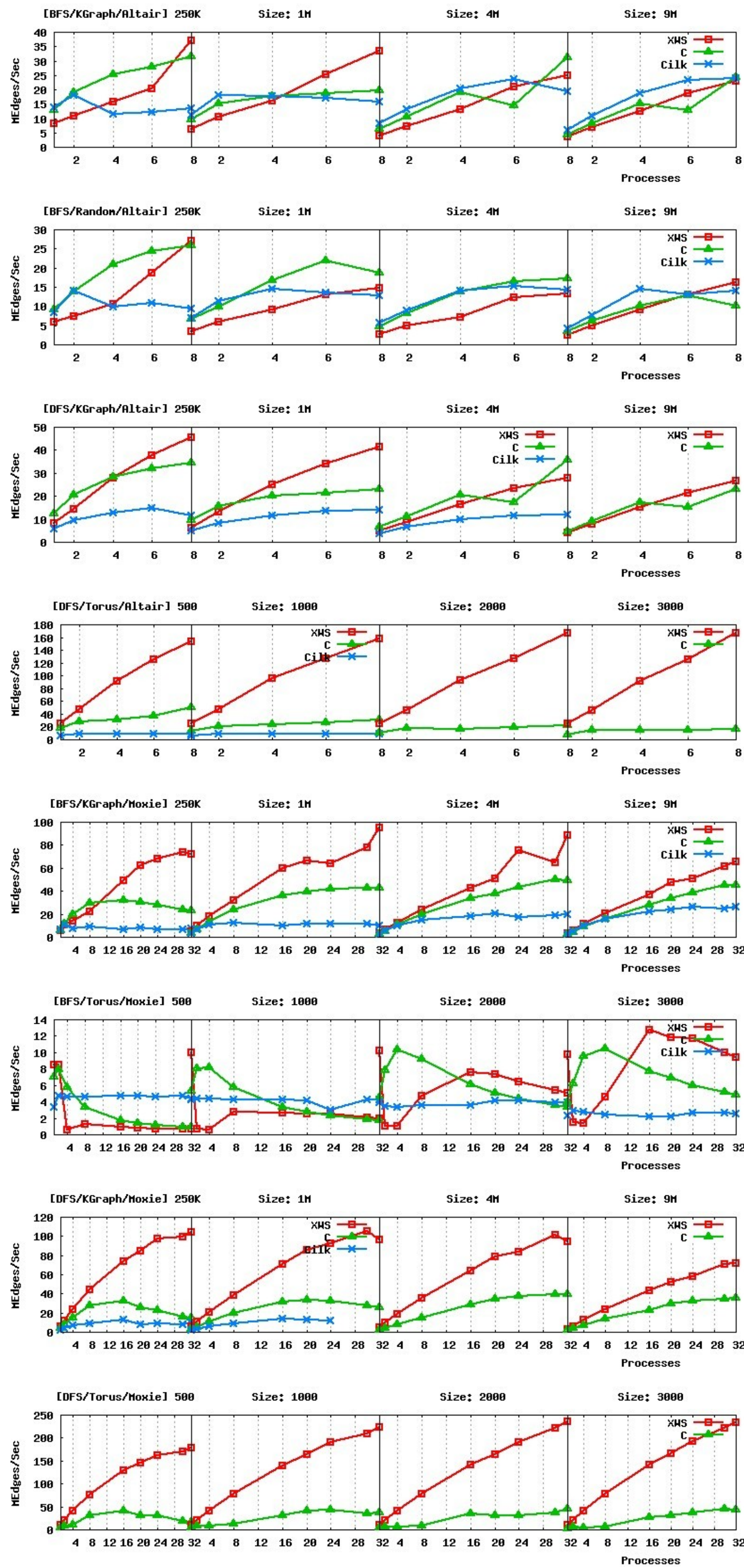

Figure 1. BFS-K,BFS-R,DFS-K,DFS-T for Opteron and BFS-K,BFS-T,DFS-K,DFS-T for Niagara