Solving Large, Irregular Graph Problems in X10

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

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*, a partitioned global address space language supporting dynamic fine-grained concurrency.* XWS *is also intended as 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, worker-specific data-structures, global termination detection, and phased computation.*

We present simple elegant programs using XWS *for different spanning tree algortihms using a (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 grannularity 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. Instead we develop an adaptive batching scheme that is sensitive to the instantaneous size of the work queue. With this scheme, pseudo-DFS shows linear scaling on altair and moxie, achieving peak performance of over 220 Million edges/s on moxie. These programs compare favorably with separate C and Cilk implementations.*

We conclude that a fine-grained concurrent language such as X10 *with a work-stealing based scheduler may provide an attractive framework for the implementation of graph algorithms.*

1. Introduction

The last few years have seen an explosion of mainstream architectural innovation — multi-cores, symmetric multiprocessors, clusters, and accelerators (such as the Cell processor, 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 theoretic problems arise in traditional and emerging scientific disciplines such as VLSI design, optimization, databases, and 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 example, the Internet, social interaction networks, transportation networks, and protein-protein interaction networks. These graphs can be modeled as 'scale-free" graphs [3]. For random and scale-free graphs no known efficient static partitioning techniques exist, and hence the load must be balanced dynamically. Moreover, the irregular

memory access pattern dictated by the input instances is not cache-friendly; graph algorithms also tend to be load/store intensive [4], and they lay great pressure on the memory subsystem.

Compared with their numerical counterparts, parallel graph algorithms take drastically different approaches than the sequential algorithms. Usually they employ fine-grained parallelism. For example, depth-first search (DFS) or breadth-first search (BFS) are two popular sequential algorithms for the spanning tree problem. Many parallel spanning tree algorithms, represented by the Shiloach-Vishkin algorithm [15], take the "graft-and-shortcut" approach, and provide $O(n)$ fine-grained parallelism. In the absence of efficient scheduling support of parallel activities, fine-grained parallelism incurs large overhead on current systems, and often the algorithms do not show practical performance advantage.

Consider the spanning tree problem. Finding a spanning tree of a graph is an important building block for many graph algorithms, for example, biconnected components and ear decomposition [13], and can be used in graph planarity testing [10]. 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 to the algorithm. First a small stub tree of size $O(p)$ is generated by one 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 that work for itself, and add its it from the victim's queue, and adding it to its own queue).

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 preferrable 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 [**?**] 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 will fail) and do nothing. Alternatively, the designer may use a modified version of the Dekker protocol [**?**], by ensuring that the thief and victim each writes to a volatile variable and read the variable written by the other. 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. Those concerns that are of interest to the graph algorithm designer (e.g. expressing breadth-first vs depth-first search) are mixed in with the concerns for efficient parallel representation. This suggests packaging the required components in a library or a framework and exposing a higher-level interface to programmers.

1.1. X10 and XWS

The X10 programming language [**?**, **?**, **?**] has been designed to address the challenges of "productivity with performance" on these diverse architectures. Designed on top of sequential Java, X10 is organized around a few core concurrency constructs built on fine-grained parallelism. Implementations of X10 are available that (a) compile X10 to Java to be run on a single Java Virtual Machine (JVM) instance running on a multicore or an SMP, and (b) compiler X10 to C++ to be run on a cluster of SMPs and machines such as the Blue Gene. Projects are currently under way to implement X10 on Cell-accelerated architectures.

In this paper we present 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) [**?**]. Work stealing is a powerful technique organized around a collection of workers (=threads) that each maintains a double-ended queue (deque) of *frames* (or tasks). A worker pops and pushes frames from the bottom of the deque. When its deque is empty, it randomly selects another worker and attempts to steal a frame from the top of its deque. CWS is carefully organized to streamline parallel overhead so that execution of the code with a single worker incurs a small constant factor overhead over execution of the sequential code. The overhead associated with stealing is deferred to the worker performing the steal (the *thief*) as opposed to the worker being mugged (the *victim*). The

CWS algorithm is known to have nice properties in theory, and can be efficiently implemented in practice.

XWS extends CWS to better support the programming of applications with irregular concurrency. It removes the link between recursion and concurrency introduced by Cilk. 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, XWSintegrates *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 significant impact on the performance of work-stealing. XWS permits the programmer to sense key metrics of the current execution and use these to adjust batching size dynamically.

XWS may be illustrated by the following sample programs (fragments of running programs):

Example 1.1 (Psuedo-DFS) The parallel exploration of a graph may be implemented quite simply by the following program:

```
class V  extends VertexFrame {
   V [] neighbors;
   V parent;
   V(int i){super(i);}
   boolean tryColor() {
     return parentUpdater.compareAndSet(this,0,1);
   }
   void compute(Worker w) throws StealAbort {
     w.popAndReturnFrame();
     for (V e : neighbors)
       if (e.tryColor()) {
         e.parent = this;
         w.pushFrame(e);
       }}}}
```

The class `V` represents a vertex with an array used to represent edges. `V` extends `Frame` and hence can be scheduled by the work-stealing scheduler. On being scheduled its `compute` method is run, with the workder executing the vertex being passed as the argument. The code for `compute` may schedule parallel work by invoking `w.pushFrame`.

Note that a parallel frame corresponds to a single vertex; this code does not implement batching. A batched version of this code is presented in Example **??**. □

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

```
class V  extends VertexFrame {
   V [] neighbors;
   V parent;
   V(int i){super(i);}
   boolean tryColor() {
     return parentUpdater.compareAndSet(this,0,1);
   }
   void compute(Worker w) throws StealAbort {
     w.popAndReturnFrame();
     for (V e : neighbors)
       if (e.tryColor()) {
         e.parent = this;
         w.pushFrameNext(e);
       }}}}
```

Here the code utilizes the implementation of a global clock (barrier) by XWS. Each worker maintains two deques, the *now* and the `next` deque. Always the now deque is active, but execution of a frame may cause frames to be added to the next deque. When all the work in the current phase has terminated (that is, all now deques across all workers are empty), and at least one worker has added a frame to the next deque, computation moves to the next phase by causing each worker to swap their next and now deques. □

1.2. Rest of this paper

The rest of this paper is as follows. In Section **??**, we present the details of the design of XWS. In Section **??** we present comparable programs written using an application-specific framework (Simple, [**?**]), as well as Cilk, and compare performance on three different kinds of loads. Our graph generators include several employed in previous experimental studies of parallel graph algorithms for related problems. For instance, we include the torus topologies used in the connectivity studies of Greiner [7], Krishnamurthy *et al.* [11], Hsu *et al.* [8], Goddard *et al.* [6], and Bader and Cong [2], the random graphs used by Greiner [7], Hsu *et al.* [8], and Goddard *et al.* [6], the geometric graphs used by Greiner [7], Hsu *et al.* [8], Krishnamurthy *et al.* [11], and Goddard *et al.* [6].

- **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, including LEDA [12].
- **Geometric Graphs and AD3** In these k-regular graphs, each vertex is connected to its k neighbors. Moret and Shapiro [14] use these in their empirical study of sequential MST algorithms. **AD3** is a geometric graph with $k = 3$.

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 processor 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 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 Future work and acknoweledgements.

2. X10 Work Stealing

2.1. Cilk work-stealing

This section describes Cilk work stealing, as implemented in XWS. It closely follows the description of the implementation of Cilk in [5].

In summary, Cilk Work Stealing (CWS) is organized around a collection of cooperating threads called *workers*. Each worker maintains a double-ended queue (deque) of tasks. During execution as a worker creates more tasks it pushes them at the bottom of the queue. When it needs more tasks it retrieves the current task from the bottom of the deque. When a worker runs out of work (its deque is empty), it randomly chooses another worker (the *victim*) and attempts to steal a task by fetching it from the top of the deque. Program begins execution when the environment submits a task to a central task queue.

One of the workers retrieves the task from the global queue and begins executing it. When a worker does not have tasks to execute it *steals* tasks available at other workers. Assuming the computation contains sufficient parallelism stealing happens infrequently. The design ensures that there are few overheads during normal execution, referred to as *the fast path*. The additional overheads incurred to load-balance the computation are proportional to the number of steals in the execution. The design of the worker for the the task types supported is discussed in subsequent sections.

The normal execution corresponds to the depth-first sequential execution of the tasks spawned. Thus the execution corresponds to a sequential execution when there is only one worker.

2.2. Support for Properly Nested Tasks

Cilk requires *fully-strict* programs in which a task waits for all its descendents to complete before returning. Such tasks are also called properly nested tasks.

The X10 runtime system is designed to leverage the Cilk design while supporting a larger class of programs. X10 provides support for *strict* computations, in which a ancestor task need not wait for its descendent tasks to be completed. Such tasks are said to be *improperly nested*.

Each worker contains a *closure*. A closure is an object used to return values from the spawned tasks to their parents in the presence of work stealing.

Each closure maintains a stack of frames. Frames corresponding to spawned tasks are pushed into the stack on entry, and popped on return. In the fast path, return values are propagated as they would be in a sequential program. The thief steals a closure together with the bottom-most available frame from its *victim*.

When a thief steals a task, the descendants of the task in the victim's dequeue continue to execute. In order to return values from the descendents to the parent, a new closure is created that on completion returns the result to the parent closure that was stolen.

Thus the closures form a tree of return value propagation corresponding to the steal operation performed. Termination is detected when the closure corresponding to the task inserted by the driver thread returns.

The procedure executed by the workers to handle properly nested tasks is shown in Fig. **??**(a). On completing execution of a closure, a worker first attempts to obtain another closure from its local queue (method:`extractBottom()`). If no local closure is available to execute, the worker attempts to obtain a task either by stealing or from the global queue (method:`getTask()`). It then executes the slow version of the task obtained (method:`execute()`).

2.3. Support for Improperly Nested Tasks

Properly nested tasks t satisfy the property that at the moment when the slow version terminates (method:`compute()`) the frame at the bottom of the worker's dequeue is t. Hence the task can be completed (i.e., removed from the dequeue) by including a `w.popFrame()` call at the end of the compute method. In essence, if a worker is executing only properly nested tasks (this is true when it is executing Cilk code), there is a one-to-one correspondence between the frame stack and the tasks being processed.

X10 permits improperly nested tasks. Such tasks q are used, for instance, to implement the pseudo-depth-first search discussed in this paper. Such a task may add a task r to the deque of its worker (say w) without necessarily transferring control to r. This has two consequences. First, recall that as soon as a worker's dequeue contains more than one task the worker may be the target of a theft. Therefore as soon as q pushes r onto w's dequeue, q is available to be stolen. Therefore q's compute method must record the fact that is computation has begun so that the stealing worker z may do the right thing.

For instance, if q's compute method does not contain any internal suspension point then z must immediately terminate execution of q and pop q off its deque. This can be accomplished by defining a volatile int PC field in q, and adding the following code at the beginning of q's compute method

Second for an improperly nested task when control returns from g's compute method, it may not be the case that the last frame on the dequeue is g. Therefore a call to `popFrame()` at the end of g's compute method would be incorrect. Instead, the compute method returns (without attempting to pop the last frame on the deque). Now whenever the task reaches the bottom of the dequeue, the worker will, as usual, invokes its compute method. However, the code sequence described above will execute, thereby popping the frame from the deque. Thus the code sequence above serves two purposes – it does the cleanup necessary when the task is stolen as well as when it is completed.

The changes to the basic worker code necessary to support improperly nested tasks are shown in Fig. **??**(b). With improperly nested tasks, a worker no long enjoys the property that when control returns to it from the invocation of an execute method on the top-level task, the deque is empty. Indeed, control may return to the scheduler leaving several tasks on the deque, including the task whose execute method has just returned. The scheduler must now enter a phase in which it executes the task at the bottom of the deque:

2.4. Global Quiescence

In fully-strict computations, i.e., those involving properly nested tasks, completion of the first task and the return of the corresponding Closure indicates computation termination. Improperly-nested tasks that do not require a return call chain can do away with the closures. We have implemented a mechanism to efficiently identify termination without closures.

The workers share a barrier. The barrier is used to determine when all workers are out of work. Every worker notifies the barrier of its state through two methods. `checkIn()` is used to enter the barrier barrier and notify that the worker is out of work. When such a worker steals work from a victim, it invokes `checkOut()` to leave the barrier. The barrier maintains a `checkoutCount` on the number of workers checked out. It is triggered when all the workers are in it. The action associated with the barrier is triggered and it signals that the computation has terminated.

The algorithm maintains the invariant:

(#workers - checkoutCount) = #(workers that know they don't have work

A worker knows it has no work if it stealing. Note that the checkoutCount is not always equal to the number of workers with work to do. In particular, consider a victim that finds the current frame as stolen. The victim cannot identify whether it has work without locking its deque. While it aborts, the thief has the stolen frame and could have invoked `checkOut()`. The barrier identifies both workers as having checked out even though there is one task between them. Note that allowing the victim to `checkIn()` when it identifies a steal would lead to the barrier being incorrectly triggered while the thief still has the stolen task but it yet to invoke `checkOut()`.

2.5. Phased Computations

We also added support for phased computations in which tasks in this phase create tasks to be executed in the next phase. The implementation of the breadth-first search algorithm proceeds one level at a time. The nodes processed at this level are used to determine the nodes to be processed in the next level.

Phased computations are supported as a generalization of global quiescence. Each worker maintains two stacks of frames, referred to as caches. Depending on the phase specified when spawning tasks, a task can be added to the current cache or the next cache, the cache for the next phase.

When global quiescence is detected for this phase, the barrier action invokes `advancePhase()` that steps the computation into the next phase. When a worker runs out of tasks, it checks that the current phase of the worker is the same as the global phase of the computation. If the phase of the computation has advanced further, the workers updates its phase information and swaps the current and next task collections.

Each worker specifies the number of tasks it has outstanding for the next phase when it invokes `checkIn()`. This information is used to identify if the next phase has any tasks left to be processed.

Note that the global phase could have advanced much further than the phase operated on by this worker. This would happen when this worker has no work for current phase and has checked-in notifying that it has not tasks for the next phase. The other workers could then progress multiple phases before this worker observes the computation progress.

When global quiescence for this phase is triggered, the number of workers with tasks for the next phase is known. The computation is said to have terminated

when the current phase has quiesced and no worker has any task for the next phase.

For a given phase, maintaining the invariant mentioned above for global quiescence is more involved with multiple phases. For example, consider a worker advancing its phase to match the global phase of the computation and its next cache is non-empty. Since the worker now has local tasks in this phase, it implicitly checks out of the barrier.

3. Graph algorithms in XWS

We consider implementations of breadth-first and depth-first search.

3.1. XWSimplementation

The code for psuedo-DFS is straightforward. A job created from the root is submitted to the pool. This call returns only when all the (concurrent) work associated with this job has terminated.

```
pool.invoke(root.makeJob());
```

3.2. SIMPLE implementation (C)

3.3. Cilk implementation

4. Performance Evaluation

While classic graph traversals such as DFS and BFS can be supported by XWS, they do not take advantage of some of the algorithm design techniques that exploit work-stealing to provide performance superior to that of other approaches to parallel programming.

Work-stealing schemes are natural fits for many classic parallel divide-and-conquer algorithms. (In fact, for some problems and criteria, work-stealing provides optimizal solutions.) For example, to process all of the elements of an array of size N, a root task represents array, 0, N which is then subdivided into two tasks array, 0, n/2 and array, n/2, n, and so on until the size of each task is less than an empirically-guided sequential threshold indicating that further subdivision is not profitable because the task overhead would be greater than the amount of work to process the task sequentially. Notice that in a shared memory environment, the memory needed for task descriptions themselves is small and constant.

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 possibling 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 would use of 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 arbitarily poor estimates. Intead, 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 proprties: Each of the of 2^j nodes in a batch of a size-j queue (for $j \leq log_2(S)$) should have 2^{-j} of the expected ex-

pansions 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 sec ??.: 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 to 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, 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.

Results The main results, with threshold value $S == 128$, are presented in fig ??. 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 that is profitable to 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.

Relative performance across thresholds

	Niagara			Opteron		
S	T	K	E	T	K	E
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 perofrmance across thresholds

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 rowwise 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. As a demonstration, table ?? shows 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 – chosen as a prime large enough to minimize cache hits. The effects on the (4Xdual) Opteron, especially for the Torus graph, are much larger than on the (single multicore) 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.

	opteron	niagara
T	7.4	2.2
K	1.3	1.2
E	1.4	1.2

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 signifcantly 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 numbers 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.

Related work Adaptive batching bears some similarities to the steal-half algorithm of Shavit et al, 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.

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] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Proc. 18th Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, New Mexico, Apr 2004.

[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] G. Cong. An evaluation of parallel algorithms on current memory consistency models. In *Proc. IASTED Paralle and Distributed Computing and Sytems, (IASTED-PDCS)*, pages 513–519, Dallas, TX, Nov. 2006.

[5] 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.

[6] S. Goddard, S. Kumar, and J. Prins. Connected components algorithms for mesh-connected parallel computers. 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 43–58, 1997.

[7] 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.

[8] 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.

[9] http://x10.sf.net. The X10 Programming Language v1.0, 2006.

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

[11] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. 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 1–21. American Mathematical Society, 1997.

[12] K. Melhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

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

[14] B. Moret and H. Shapiro. An empirical assessment of algorithms for constructing a minimal spanning tree. In *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science: Computational Support for Discrete Mathematics **15***, pages 99–117. American Mathematical Society, 1994.

[15] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algs.*, 3(1):57–67, 1982.

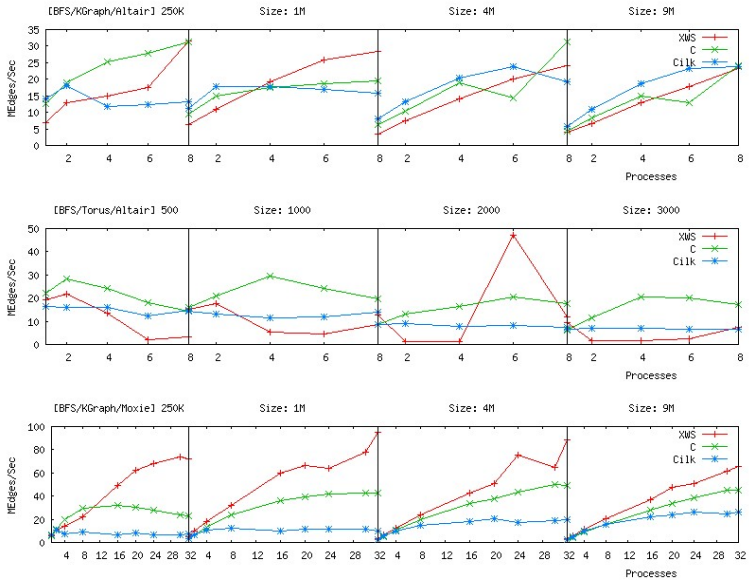

Figure 1. Psuedo-DFS and BFS for Altair(Opteron) and Moxie(Niagara)