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

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. For instance, batches of size* 1 *yield a peak performance of 20 MEPS (Million edges/second) on Niagara. Instead we develop an adaptive batching scheme that in which the batch size 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 MEPS on moxie and substantially outperforming C and Cilk implementations.*

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 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, for example, biconnected components and ear decomposition [7], and can be used in graph planarity testing [6]. 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 $O(p^2)$ 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).

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 [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 will fail) and do nothing. Alternatively, the designer may use a modified version of the Dekker protocol [2], 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 [8] has been designed to address the challenges of "productivity with performance" on these diverse architectures. In this paper we present the design, implementation and evaluation of a portion of the X10 runtime system for multi-core and SMPs, XWS. XWS implements fine-grained concurrency through an extension of Cilk Work Stealing (CWS) [2]. 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, 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 – for instance we have observed that on the 32-way Niagara the best performance for batches of size 1 (i.e. without batching) is approximately 20 Million edges per second (MEPS), whereas with batching it goes up to nearly 240 MEPS. 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 parentU.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 worker 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 is discussed later.) □

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 parentU.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 and causes 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 2, we present the details of the design of XWS. In Section 3 we present comparable programs written using an application-specific framework (Simple, [1]), 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, random graphs and geometric graphs, used in the by [4], Hsu *et al.* [5], 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 Graphs** In these k-regular graphs, each vertex is connected to its k neighbors.

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 Related work and acknoweledgements.

2. X10 Work Stealing

2.1. Support for Improperly Nested Tasks

Cilk requires *fully-strict* (properly nested) programs in which a task waits for all its descendents to complete before returning.

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

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 first pop q off the deque before spawning new asyncs. (Hence the `w.popAndReturnFrame()` in the body of `compute` in Example 1.1.)

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:

2.2. 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 the closures. We have implemented a mechanism to efficiently identify termination without closures; in essence the mechanism detects the stable property "all deques are empty."

The workers share a barrier with count `checkCount` – this will measure 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.

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 or the next deque.

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 this number, 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 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.)

3. Graph algorithms in XWS

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

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 activitiy 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 activities spawned during its execution.

Computation in Cilk has to be partitioned in a divide-and-conquer way so as to conform to the Cilk programming model in order to achieve scalability. To avoid using locks, there are two sets of buffers Red and $Black$ in the Cilk implementation of BFS. The buffer set Red contains the current BFS frontier, distributed across its member buffers and initially containing the root vertex. The buffer set $Black$ will contain the neighbors of the vertices in the current frontier to form the next one. The Cilk implementation partitions the current frontier into at most `np` blocks (specified by a command-line argument). There are `np` buffers in each buffer set). The neighbors of the ith block of vertices in the current frontier are stored in ith buffer of the buffer set $Black$. Iteration over the blocks of the current frontier has to be organized in a divide-and-conquer way where the base case is the iteration over a single block. Once the next frontier is formed, the colors of the buffer sets are switched after a Cilk `sync` statement that functions as a barrier. As in the C implementation, repetitive appearances of vertices is allowed. The algorithm iterates until no new frontier vertices are discovered.

3.3. XWS implementation

The core code for the XWS implementation is presented in Section 1. We now discuss 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 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 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 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.

	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 performance across thresholds

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 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.

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. As a demonstration, the table below 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.

	opteron	niagara
T	7.4	2.2
K	1.3	1.2
E	1.4	1.2

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.

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.

3.3.1. Adaptive BFS 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 1,2, 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.

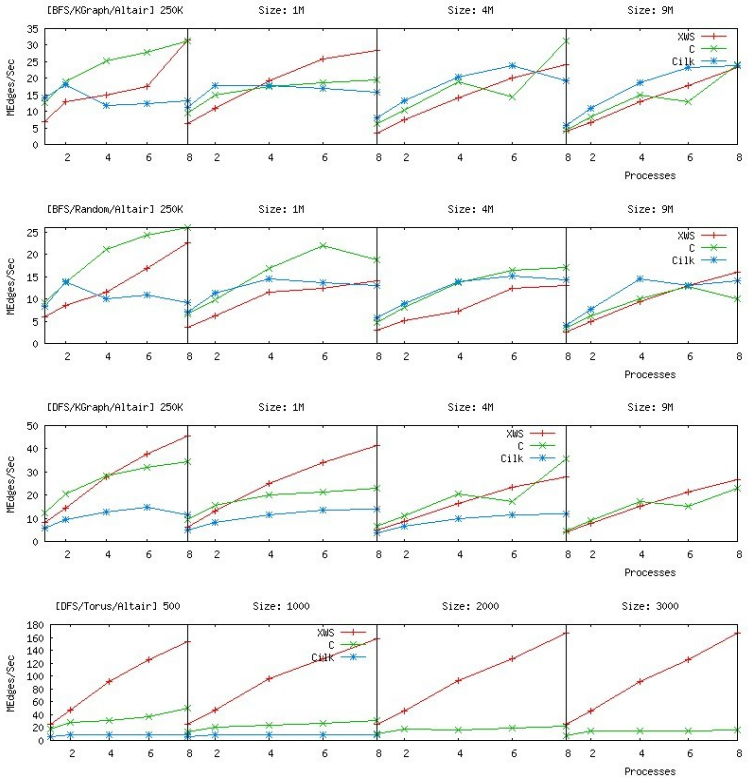

Figure 1. BFS and DFS for Opteron

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

3.4. Results

We present results in Figure 1 for runs of BFS (KGraph, Random) and DFS (Kgraph, Torus) on Opteron and in Figure 2 for runs of BFS (KGraph,Torus) and DFS (KGraph, Torus) on Niagara (y-axis: MEPS, x-axis: P), for 250K, 1M, 4M and 9M vertices. (Note that the number of vertices in a torus are the square of the torus size.) 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.[1]

1 Several Cilk runs did not complete successfully and are hence omitted.

4. Conclusion and Related work

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 re-split. 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.

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 ehanced 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 show that the XWS programs scale and exhibit performance comparable with hand-written C programs.

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.

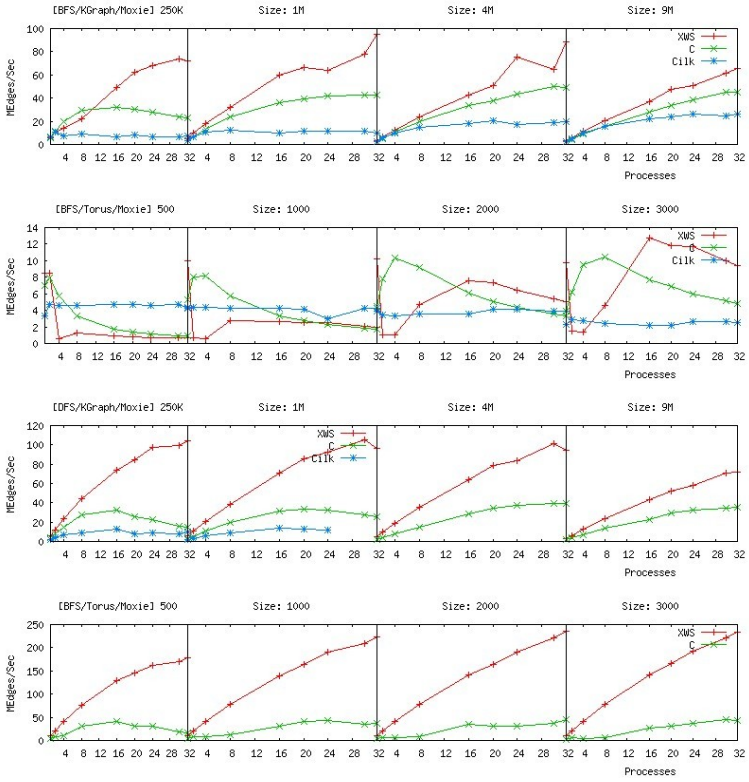

Figure 2. BFS and DFS for Niagara

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] 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.

[5] 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.

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

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

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