Solving Large, Irregular Graph Problems in X10

Guojing Cong, Vijay Saraswat, and Tong Wen
IBM T. J. Watson Research Center
{ gcong, vsaraswa, tongwen }@us.ibm.com
Sreedhar Kodali
IBM Systems and Technology Group
srkodali@in.ibm.com
Sriram Krishnamoorthy
Ohio State University
krishnsr@cse.ohio-state.edu

Abstract

Graph problems have many applications in high performance computing disciplines. Obtaining practical efficient implementations for large, irregular graph instances remains a challenge. Current hardware and software systems do not support fine-grained, irregular parallelism well. Implementing a custom framework for fine-grained parallelism for each new graph algorithm is impractical.

In this paper we take the approach of writing graph algorithms in a high-level concurrent language with dynamic fine-grained parallelism (X10). The X10 compiler translates this code into calls to an X10 runtime (written in Java) organized around a Cilk-like work-stealing scheduler. [1] The scheduler extends the Cilk scheduler with several features necessary to efficiently implement graph algorithms, viz., support for improperly nested procedures, support for global termination detection, and support for phased computation.

We show how to express spanning tree algortihms using (a) pseudo-depth first search, (b) breadth-first search and (c) Shiloach-Vishkin (SV) algorithm in a simple elegant fashion. We compare the performance of the code with code written manually in C, and with code written in Cilk. (The Cilk and C codes perform no garbage-collection.) Tests are performed on a 32-way Sun Fire T200 server (moxie), and an 8-way SunFire V40Z server (altair). We compare performance on (i) graphs with randomly selected edges (Random), (ii) graphs with all vertices having a fixed outdegree (KGraph),and randomly selected edges, (iii) Torus planar graphs (each vertex is connected to its four neighbors). For the SV algorithm, we were unable to write a Cilk program; we show that the X10 program scales at the same rate as the hand-written C program, but its performance is marginally poorer. For DFS and BFS algorithms, we show that on large Random and KGraphs, X10 programs perform comparably to hand-written C, and better than Cilk, whereas on large Torus graphs, X10 performs better than C and substantially better than Cilk. We note that the BFS and DS X10 programs are approximately a dozen lines long, whereas the hand-written C code is several hundred lines long.

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

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 (e.g. on a shared memory symmetric multiprocessor 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 [5], and they lay great pressure on the memory subsystem. Compared with their numerical

[1] The compiler is still under development, all tests in this paper were performed on code manually compiled from X10 source.

counterparts, parallel graph algorithms take drastically different approaches than the sequential algorithms, and usually 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 [17], 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.

We take the spanning tree problem as our example. Finding a spanning tree of a graph is an important building block for many graph algorithms, for example, biconnected components and ear decomposition [15], and can be used in graph planarity testing [12]. 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 restricting 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 (by removing 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.

Thus 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 Cilk Work-stealing

Such a framework has been proposed for a class of parallel programs for shared memory machines by the designers of Cilk. Cilk is an extension to C which permits the programmer to specify that some statements may be executed in parallel `spawn`, and to specify that execution of the current procedure body should suspend until all statements spawned during its execution are terminated (`sync`).

The Cilk compiler translates source code into C by using a "continuation passing transformation" and adding calls to the runtime (Cilk work-scheduler) which manages a double-ended queue (deque) for each worker (= thread). On each procedure entry an activation frame is pushed onto thhe bottom of the deque. On executing a spawn a field in the activation frame is updated to point to the instruction after the spawn, and the code in the body of the spawn is executed as a sequential procedure call. Now if a thief visits this worker looking for work it may steal the current activation frame and continue executing the body of the procedure at the instruction after the spawn. When the victim returns from executing the spawn it will discover (using the Dekker protocol) that the parent frame has been stolen and will now itself go looking for work (by randomly selecting a victim worker and attempting to steal from it). The sync statement is executed by keeping track of the number of live children, and setting up the remainder of the body of the procedure to be executed once this count reaches zero.

The Cilk compiler gains further performance improvements by generating two copies of each procedure (one for

```
class V {
  V parent;
  V[] neighbors;
  void dfsTree() {
    parent=this;
    finish dfsTraverse();
  }
  void dfsTraverse() {
    for(V v : neighbors) {
      atomic v.parent = (v.parent==null?this:v.parent);
      if (v.parent == this)
         async v.dfsTraverse();
    }
  }
}
```

Table 1: Pseudo Depth-first spanning tree algorithm core in X10

the "fast path" and one for the slow path). The fast path corresponds to execution with no steals and gives efficiency very close to sequential execution.

Since synchronization in Cilk is naturally tied Cilk naturally lends itself to the expression of recursive data-structures since synchronization is tied with procedure invocation. The literature has many examples of elegant exploitation of such recursive parallelism (e.g. cache-oblivious algorithms [**?**]).

1.2 X10 work-stealing

However, there are problems with Cilk work-stealing when applied to graph algorithms. Consider the code for a parallel form of pseudo-depth-first traversal (for spanning tree) in Table **??**. (The code is written in X10 [**?**].) Here on visiting a node (in the body of `traverse`) we wish to spawn parallel activities (`async`) to visit each neighboring vertex that has not been visited before. However, there is no reason for the current activity to wait for all these activities to terminate before returning from the body of the `traverse` routine. Specifically, the invocation of `traverse` does not return a value – it merely updates the heap to record the parent for the visited node. Hence there is no need for the invoker to wait for a return value. Thus the body does not need a `sync` statement. But this makes `traverse` illegal as a Cilk procedure. Said differently, a natural expression of parallel graph algorithms requires the use of improperly nested computation.

Further, note the use of `finish` in the body of the `tree` method. This states that the invoking activity must wait until all activities spawned during the execution of `traverse()` have terminated. `tree()` is invoked precisely once – at the root node for the spanning tree – and hence there is exactly one sync for the duration of the spanning tree computation. Thus another refinement of Cilk is natural and useful here: the implementation must support global termination detection (i.e. it must detect when the dequeues associated with all queues are empty).

Phased computation Finally, we note that breadth-first search requires another idea, that of *phased computation*. Here we mark vertices with a level number corresponding to the distance from the root node. It is necessary to ensure that a vertex is visited at distance i is visited in the ith phase. This means that activities cannot be launched immediately for vertices marked in the ith phase. Instead all these activities must wait for all the vertices marked in the $i-1$st phase to be processed, i.e. one must integrate barrier-based synchronization with work-stealing.

We show the program for expressing this computation in Table 2. While this program will be discussed in detail later, we mention now that `c` should be thought of as a *barrier*. The activity spawned to examine a newly discovered vertex must wait until the barrier is reached (`next`) before commencing its own traversal. Thus the program combines barrier-based synchronization with fine-grained parallelism. The program clearly expresses the strong relationship with the depth-first search variant – there are no details about the work-stealing implementation polluting the program.

1.3 Rest of this paper

In this paper we show that pseudo- Depth-first, breadth-first and Shiloach-Vishkin algorithms for spanning tree can be programmed quite elegantly in X10. Further we present the design of XWS, the X10 work-stealing scheduler, which extends the Cilk work-stealing scheduler with support for improperly nested computation, termination detection and phased computation. These mechanisms are sufficient to execute the three graph algorithms efficiently.

```
class V {
  V parent;
  V[] neighbors;
  void bfsTree() {
    parent=this;
    finish async {
      clock c = new clock();
      bfsTraverse(c);
    }
  }
  void bfsTraverse(clock c) {
    for(V v : neighbors) {
      atomic v.parent = (v.parent==null?this:v.parent);
      if (v.parent == this)
         async clocked(c) { next; v.bfsTraverse();}
    }
  }
}
```

Table 2: Breadth-first spanning tree algorithm core in X10

We use a collection of sparse graph generators to evaluate these algorithms. Our 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 [9], Krishnamurthy *et al.* [13], Hsu *et al.* [10], Goddard *et al.* [8], and Bader and Cong [2], the random graphs used by Greiner [9], Hsu *et al.* [10], and Goddard *et al.* [8], the geometric graphs used by Greiner [9], Hsu *et al.* [10], Krishnamurthy *et al.* [13], and Goddard *et al.* [8].

- **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 [14].
- **Geometric Graphs and AD3** In these k-regular graphs, each vertex is connected to its k neighbors. Moret and Shapiro [16] use these in their empirical study of sequential MST algorithms. **AD3** is a geometric graph with $k = 3$.

We compare the performance of XWS to two other programs. First is a hand-written C program, a variant of [1]. Second, we have written versions of pseudo-DFS and BFS in Cilk. (We were unable to write the SV algorithm in Cilk.)

We present performance data on two machines. Altair 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 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.

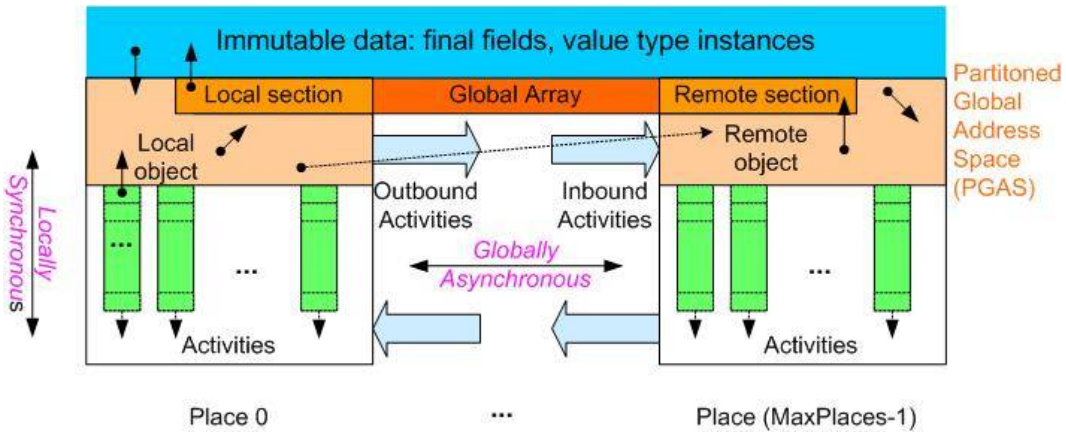
We discuss in detail the design of XWS, including our extensions to CWS.

Finally we conclude with a discussion of related and future work.

2 X10: Designed for High Productivity

X10 is a new Partitioned Global Address Space (PGAS) language being developed at IBM as part of the DARPA HPCS project [4]. It is designed to address both programmer productivity and parallel performance for modern architectures from the multicores, to the heterogeneous accelerators (as in the Cell processor), and to the scale-out clusters of SMPs such as Blue Gene. The language is based on sequential Java with extensions for programming fine-grained and massive parallelism. Unlike other PGAS languages such as Co-Array Fortran, Titanium, and UPC whose model of parallelism is Single Program Multiple Data (SPMD), X10 supports dynamic and structured concurrency with SPMD as a special case. In this section, we provide a brief introduction to the basic concepts in the X10 programming model and the language constructs used to implement the graph algorithms of interest. For more details and other features of X10, readers please refer to [4].

1. **Activities** – All concurrency in X10 is expressed as asynchronous *activities*. An activity is a lightweight thread of execution, which can be spawned recursively in a fork-join manner. The syntax of spawning an activity



Dynamic parallelism with a Partitioned Global Address Space. All concurrency is expressed as asynchronous activities. Each vertical green rectangle above represents the stack for a single activity. An activity may hold references to remote objects, that is, at a different place. However, if it attempts to operate on a remote object, then it has to spawn a new activity at the remote place to perform the operation. Immutable (read only) data is special which can be accessed freely from any place providing opportunity for single-assignment parallelism.

Figure 1: X10 Programming Model

is `async S`, where a new child activity is created executing statement `S`. The granularity of an activity is arbitrary – `S` can be a single statement reading a remote variable or a sequence of statements performing a stencil operation on a grid. Our experience has been that this single notion of an asynchronous activity can subsume many levels of parallelism that a programmer may encounter such as threads, structured parallelism (including OpenMP), messaging (including MPI), and DMA transfers.

2. **Places** – The main program starts as single activity at *place* 0. Place is an X10 concept which can be considered as a virtual SMP, but multiple places can be mapped to one physical SMP node. The global address space is partitioned across places. Data and activities have affinity with and only with one place. Activities can only operate on data local to them, that is, within the same place. To access data at another place, a new activity has to be spawned there to perform the operation. The syntax for spawning an activity at place p is `async (p) S`. The diagram in Figure 1 describes the X0 programming model.

3. **X10 arrays** – X10 supports a rich set of multidimensional array abstractions and domain calculus as in Titanium [6]. The array index space is global where each index is an integer vector named *point*, and a *domain* is a set of points which can be either rectangular or not. The distribution of an array across places is specified by a *distribution* which can be defined by users. Each distribution maps a set of points in a region to a set of places.

4. **Parallel loops** – There are two kinds of parallel loop in X10: `foreach` and `ateach`, for looping over a region and a distribution respectively. Their difference is that the `for` loop spawns activities locally, whereas the `ateach` loop spawns activities at the places specified by the distribution.

5. **Finish and clock** – The statement `async S` returns immediately when it is executed even if the statement `S` is not finished, which may also spawn other activities. The `finish S` construct is used to specify that the next instruction is executed only when the statement `S` has finished globally, that is, all transitively spawned child activities have finished. While `finish` permits the detection of global termination, there are many cases in which a barrier-like coordination is needed for a set of activities during the middle of their computation. X10 uses *clock* to coordinate such a set of activities. A clock has phases, and the activities registered with this clock can be synchronized by waiting for their finish of the current clock phase. An activity can be registered with multiple clocks and it can drop any of them at any time.

6. **Atomic blocks** – X10 uses *atomic* blocks for mutual exclusion. An atomic statement/method is conceptually executed in a single step, while other activities are suspended. An atomic block must be nonblocking, sequential

(without spawning activities), and local (no remote data access). *Conditional* atomic block is another parallel language construct of X10 which can be used, for example, to implement point-to-point synchronization. The syntax for a conditional atomic block is `when (E) S`, where the executing activity suspends until the boolean expression `E` is true, then `S` is executed atomically.

3 XWS

We have implemented a runtime system for X10. As an initial implementation, we have developed a Java-based runtime system to support shared-memory parallelization. It is distributed as a Java package, `x10.runtime.cws`, under an open-source license [11]. In this section, we discuss the implementation of key features of the runtime system.

The computation is organized as collection of *tasks*. A task is a sequence of instructions that can spawn other tasks and wait for completion of the spawned tasks. The computation begins with a single task and is considered complete when there are no more tasks executing in the system.

3.1 Cilk work-stealing

This section describes Cilk work stealing, as implemented in XWS. It closely follows the description of the implementation of Cilk in [7].

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.

We now describe the details.

Every *async* in the X10 program is compiled into a sub-class of `Frame`. An *async* is said to consist of *threads*, where a thread is a non-blocking sequence of instructions terminating in a spawn or waiting on an *async*.

Each class created for an *async* contains as member variables the local variables used in the *async* body, a counter (*PC*) that specifies the next instruction in the body of the *async* after the current thread.

Cilk implements a scheme for handling continuations. Two versions of the *async* body are created – the fast and the slow versions. The fast version of the program is executed during the normal course of the program. The slow version is invoked by the worker to initiate processing of a task. Fig. 2 shows the fast and slow versions for the Fibonacci method shown in Fig. 2(a).

Each worker maintains a deque consisting of stack of Frames. On entering a fast version, a frame object is created and pushed onto the stack (method: `pushFrame()`). This frame contains the up-to-date values of variables whenever any other worker might steal this task. This frame is popped from the stack (method: `popFrame()`) when this method returns.

Whenever a task is spawned, the worker immediately proceeds to execute the spawned task like a sequential method call. When the spawned task finishes execution, the worker checks whether the current frame was stolen. If so, the result computed by the child task is stored to be passed onto its parent and the execution of the task aborts (method: `abortOnSteal()`). The method call stack is unwound by throwing an exception (`StealAbort`) that is caught in the main routine executed by the worker. The values in the frame of the parent stack are updated before proceeding to execute a spawned child, as the worker now has a non-executing task and hence is target of a steal.

All the descendents are guaranteed to be completed in the fast version when a `finish` is encountered. Hence the finish statements are ignored.

```
int fib(int n) {
  int x, y;
  if(n<2) return n;
  finish {
    async x=fib(n-1);
    async y=fib(n-2);
 }
  return x+y;
}
```

(a)

```
int fib(Worker w, int n)
 throws StealAbort {
  int x, y;
  if (n < 2) return n;

  FibFrame frame = new FibFrame(n);
  frame.PC=LABEL_1;
  w.pushFrame(frame);

  x = fib(w, n-1);
  w.abortOnSteal(x);

  frame.x=x;
  frame.PC=LABEL_2;

  y=fib(w, n-2);
  w.abortOnSteal(y);

  w.popFrame();
  return frame.x+y;
}
```

(b)

```
void compute(Worker w,
 Frame frm) throws StealAbort {
  int x, y;
  FibFrame f=(FibFrame)frm;
  int n = f.n;
  switch (f.PC) {
  case ENTRY:
    if (n < 2) {
      result = n;
      setupReturn();
      return;
   }
    f.PC=LABEL_1;
    x = fib(w, n-1);
    w.abortOnSteal(x);
    f.x=x;
  case LABEL_1:
    f.PC=LABEL_2;
    int y=fib(w,n-2);
    w.abortOnSteal(y);
    f.y=y;
  case LABEL_2:
    f.PC=LABEL_3;
    if (sync(w)) return;
  case LABEL_3:
    result=f.x+f.y;
    setupReturn();
 }
}
```

(c)

Figure 2: (a) X10 program for Fibonacci. (b) Fast version. (c) Slow version

The slow version restores any local variables and uses the *PC* to start execution of the task past the execution point at which it might have been stolen. A task might have been stolen when its descendents are executing. Thus a *finish* statement, translated to the `sync()` method, might cause the invoking task to suspend on completion of non-terminated children. The value of the `result` variable is returned to the parent task on invocation of the `setupReturn()` method.

In a program with sufficient parallelism, the slow version is expected to execute infrequently. The design tries to minimize the overheads in the fast version even at the expense of the slow version (the "work-first" principle).

3.2 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. 3(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()`).

```
public void run() {
  Executable cl=null; //frame/closure
  int yields = 0;
  while (!done) {
    if (cl == null ) {
      //Extract work from local queue.
      //It will be a closure
      //cl may be null. When non-null
      //cl is typically RETURNING.
      lock(this);
      try {
        cl = extractBottom(this);
      } finally {
        unlock();
      }
    }
  }
  if (cl == null)
    //Steal or get from global queue
    cl = getTask(true);
  if (cl !=null) {
    // Found some work! Execute it.
    Executable cl1 = cl.execute(this);
    cl=cl1;
    cache.reset();
  } else Thread.yield();
}
```

(a)

```
public void run() {
  Executable cl=null; //frame or closure
  int yields = 0;
  while (!done) {
    if (cl == null ) {
      //Addition for GlobalQuiescence.
      //Keep executing current frame
      //until dequeue becomes empty.
      if (jobMayHaveImproperTask) {
       Cache cache = this.cache;
        for(;;) {
          if(!cache.empty())
            Frame f=cache.currentFrame();
          if (f == null) break;
          Executable cl1=f.execute(this);
          if (cl1 != null) {
            cl=cl1;
            break;
          }
        }
      }
      //Rest of worker code same as for
      //properly nested tasks ...
    }
  }
}
```

(b)

Figure 3: Code executed by workers for (a) only properly nested tasks (b) properly and improperly nested tasks. Note that (b) is an extension of (a)

3.3 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

```
if (PC==1) {
  w.popFrame();
  return;
}
PC=1;
```

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. 3(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:

```
Closure steal(Worker thief) {
  lock(thief); //lock victim deque
  Closure cl = peekTop(thief, victim);
  if (cl==null)
    return null; //nothing to steal
  //cl = Closure that can be stolen
  cl.lock(thief);
  Status status = cl.status();
  if (status == READY) {
    //Closure not processed by victim
    //steal the Closure
  }
  else if (status == RUNNING) {
    //Possible contention with victim
    //Need to steal head frame in Closure
    if (cache.dekker(thief)) {
      //>1 Frame available in Closure
      //Promote child frame to Closure
      //Steal this Closure & head frame
    }
  }
  return null; //No Closure to steal
}
```

(a)

```
Frame stealFrame(Worker thief) {
  Worker victim = this;
  lock(thief);

  //>1 frame in victm's frame stack?
  boolean b=cache.dekker(thief);
  if (b) {
    //Frame available to steal
    Frame frame = cache.headFrame();
    //Mark this frame as stolen
    cache.incHead(); //H=H+1
    return frame;
  }
  return null; //No frame to steal
}
```

(b)

Figure 4: Work stealing algorithm for (a) Properly nested tasks (b) Improperly nested tasks. Both are invoked on victim's Worker object (victim==this). Locks held are freed before returning.

3.4 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:

$$(\#\text{workers} - \text{checkoutCount}) = \#(\text{workers that know they don't have work})$$

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()`.

3.5 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.

3.6 Explicitly Partitioned Programs

While work-stealing provides a convenient abstraction for load-balancing fine-grained programs, the performance of certain applications can benefit from the explicitly partitioned programs that would exist as a typical parallel global address space program. For example, in the Shiloach-Vishkin algorithm we observe that explicit partitioning of the edges and vertices amongst the workers leads to better performance. This is achieved by having the task, called the job task, submitted by the driver thread spawns tasks, called worker tasks. The number of worker tasks spawned is equal to the number of workers in the system. The job task is improperly nested and returns upon spawning the worker tasks. Each of the spawned tasks now execute one part of the program. Note that these tasks are spawned and available at the worker that executed the job task. This worker now has work and it executes one of the worker tasks. The other workers identify the presence of work in this worker while stealing and steal a worker task. Assuming the worker tasks are sufficiently load-balanced, the rest of the computation proceeds without work-stealing with each worker executing a worker task.

3.7 Performance Analysis

In this section, we discuss the performance implications of the different components of the runtime.

The overheads of the fast version over the sequential execution are:

1. Method's frame needs to be allocated, initialized, pushed onto the deque. Cost: A few assembly instructions.
2. Method's state needs to be saved before each spawn. Cost: writes of local dirty variables and *PC*, and a `StoreLoadBarrier`
3. Method must check if its frame has been stolen after each return from a spawn. Cost: two reads, compare, branch and a `StoreLoadBarrier`
4. On return, frame must be freed. Cost: A few assembly instructions
5. An extra variable is needed to hold the frame pointer. Cost: Increased register pressure.

Note that average cost of allocating and de-allocating memory can be reduced to a couple of statements with an efficient concurrent memory allocation scheme. Thus the overhead in the fast path is very little, as is also demonstrated in the experimental evaluation.

The number of closures created and invocations of the slow version of an *async* is proportional to the number of successful steals. The number of locks requested is proportional to the numbers of attempted steals.

Improperly nested tasks take advantage of the lack of return value to avoid the creation of closures, further reducing overhead.

The computation is identified as terminated, the moment the last worker starts trying to steal. Thus there is no significant delay between the actual termination of the computation and its detection. The mechanism itself incurs the overhead of two atomic updates to a shared counter for every successful steal. The overheads incurred in supported phased execution are similar.

Explicitly partitioned programs begin execution with all the tasks on one worker. There is a delay before which all workers attempt to steal from this worker and obtain a worker task. Assuming a truly random scheme in the choice of the victim to steal from, the worst case in the number of steals before work is found is proportional to the number of workers. Since the worker tasks are load-balanced and there is no work-stealing after this initial delay, the number of steals in the worst case is independent of the program running time and problem size, and hence much smaller than that incurred in a typical work-stealing strategy. We observe this in our experimental evaluation as well.

Thus the common execution path in which all workers are busy involves little overhead. The remaining in other execution paths are proportional to the number of attempted and successful steals performed by the workers. The experimental evaluation section demonstrates that both are far lower than the number of *asyncs* spawned, effectively enabling load-balanced execution of fine-grained parallel programs.

4 Performance Evaluation

4.1 C implementation

All the C implementations follow the SPMD programming style. P threads are created, and each of them is assigned a piece of work . The DFS implemenation balances the workload through a simple, explicit work-stealing scheme as described in [**?**]. Both the BFS and SV implementaions simply distribute the input array evenly to each processor to work on. Ajacency array are used for the input representation in BFS and DFS. SV simulates the corresponding PRAM algorithm, and uses the edge list representation. In DFS, mutual-exclusion through atomic instruction is used for synchronization. BFS and SV employ only barriers, and the barrier implemenation is the usual $O(\log P)$ tree implemenation.

DFS in Cilk use the same input represenation as the C implementation. Concurrency and synchronization are supported by Cilk runtime and Cilk lock (an efficient implementation through atomic instructions).

The performance numbers for handwritten C application and for the XWS code are given below, for psuedo-depth-first search, breadth-first search and the Shiloach Vishkin algorithm for spanning tree (from top to bottom). Numbers are presented for two machines, altair (containing 4 dual-core Opterons) on the left and moxie, a 32-processor Niagara on the right. The best numbers for ten consecutive runs are presented.

The Java programs were run using the experimental Java 1.7 release on both machines. C programs were compiled with Sun cc v 5.8 and Cilk programs with the 5.3.2 compiler.

The graphs show good scaling for the XWS version of applications on both machines. In all cases the numbers for XWS are better than that for the C code at higher processor counts.

Cilk does not perform well for BFS. (We were unable to get a Cilk version of SV running in time for the submission.)

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 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 (DFS), breadth-first search (BFS) and a Shiloach-Vishkin algorithm on two multicore systems. We show that the XWS programs scale and exhibit performance comparable with hand-written C programs.

Acknowledgements. XWS is being designed and implemented in collabration with Doug Lea. 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.

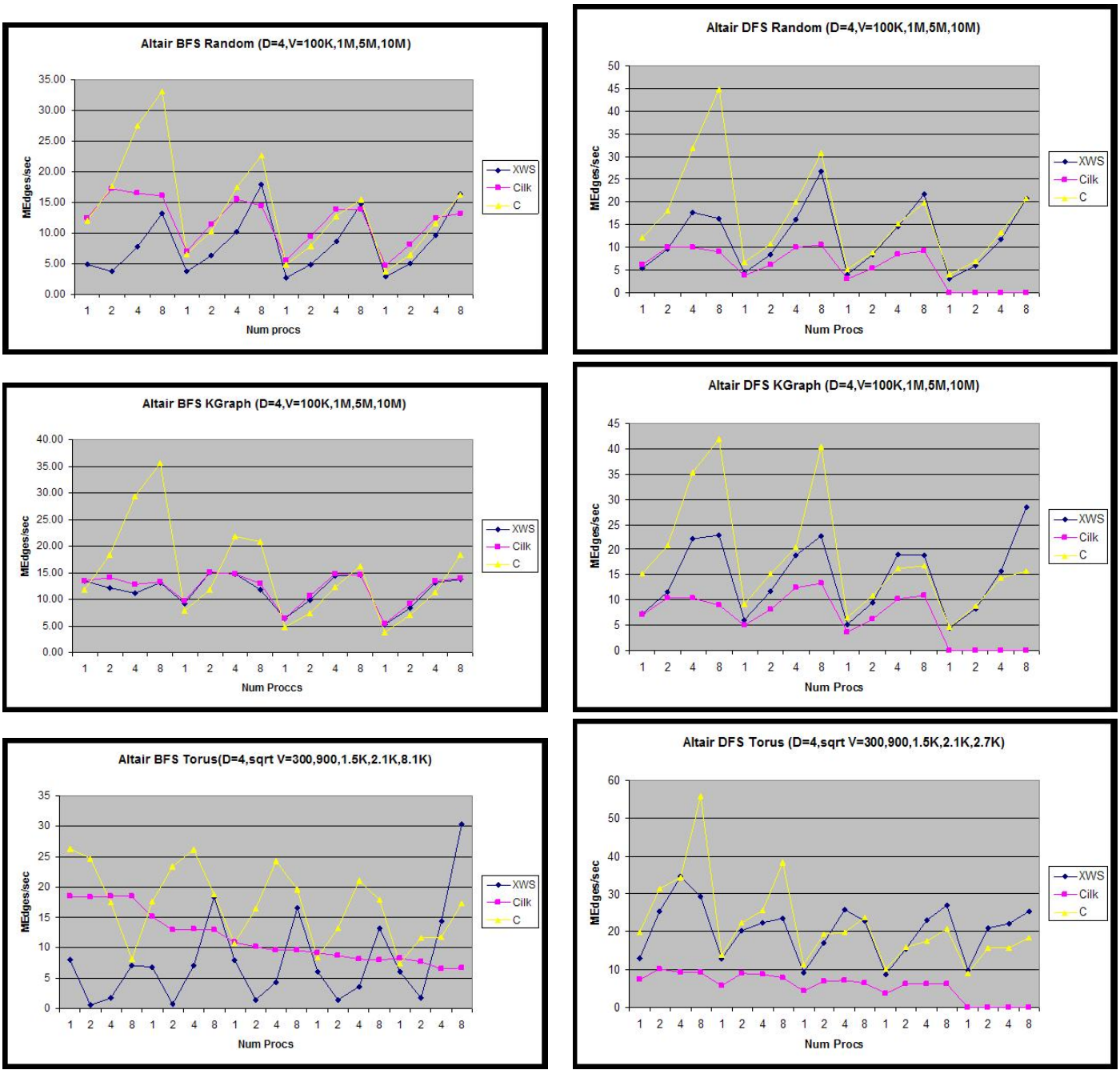
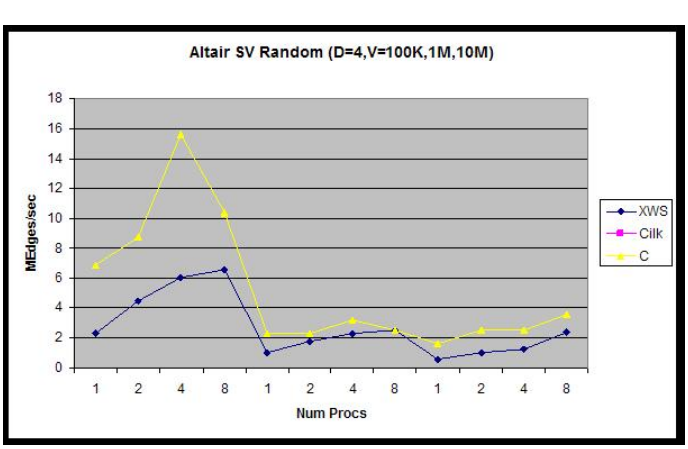
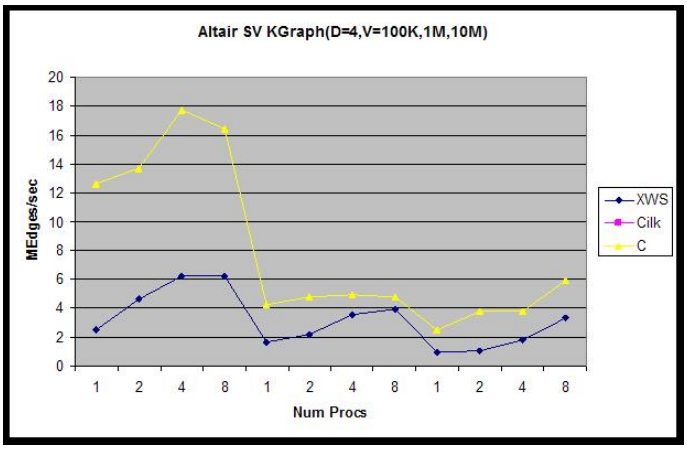


Figure 5: Psuedo-DFS and BFS for altair





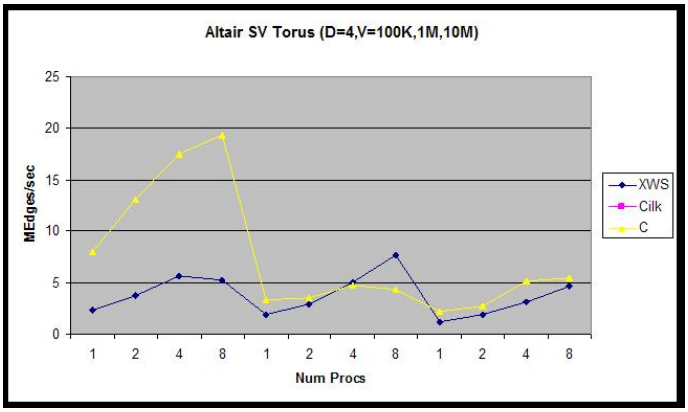


Figure 6: SV for altair

[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] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. V. Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2005.

[5] 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.

[6] P. H. (ed.) et. al. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, University of California, Berkeley, 2005.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

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

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

[16] 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.

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