Idempotent Work Stealing

Maged M. Michael
IBM Thomas J. Watson Research Center
magedm@us.ibm.com

Martin T. Vechev
IBM Thomas J. Watson Research Center
mtvechev@us.ibm.com

Vijay A. Saraswat
IBM Thomas J. Watson Research Center
vsaraswa@us.ibm.com

Abstract

Load balancing is a technique which allows efficient parallelization of irregular workloads, and a key component of many applications and parallelizing runtimes. Work-stealing is a popular technique for implementing load balancing, where each parallel thread maintains its own work set of items and occasionally steals items from the sets of other threads.

The conventional semantics of work stealing guarantee that each inserted task is eventually extracted exactly once. However, correctness of a wide class of applications allows for relaxed semantics, because either: i) the application already explicitly checks that no work is repeated or ii) the application can tolerate repeated work.

In this paper, we introduce *idempotent work stealing*, and present several new algorithms that exploit the relaxed semantics to deliver better performance. The semantics of the new algorithms guarantee that each inserted task is eventually extracted *at least* once–instead of *exactly* once.

On mainstream processors, algorithms for conventional work stealing require special atomic instructions or store-load memory ordering fence instructions in the owner's critical path operations. In general, these instructions are substantially slower than regular memory access instructions. By exploiting the relaxed semantics, our algorithms avoid these instructions in the owner's operations.

We evaluated our algorithms using common graph problems and micro-benchmarks and compared them to well-known conventional work stealing algorithms, the THE Cilk and Chase-Lev algorithms. We found that our best algorithm (with LIFO extraction) outperforms existing algorithms in nearly all cases, and often by significant margins.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, scheduling, synchronization, threads*.
General Terms: Algorithms, Management, Performance.

1. Introduction

Statically parallelizing applications with irregular workloads is a very challenging task. The key problem in trying to come up with a scalable static algorithmic solution is that the amount of available parallelism can change dramatically from one invocation of the algorithm to another. One answer to this challenge is the well-known dynamic technique of load balancing. Load balancing works by dynamically distributing the work to each process. It is a key technique used in many runtimes for parallel languages such as Cilk [4] and X10 [5]. It is also a core component of parallel garbage collectors [7], now a part of most modern virtual machines. Increased proliferation of load balancing techniques and their central place in most parallelizing systems dictate the need for high-performing load balancing algorithms.

Work-stealing is a technique that implements load balancing. Effectively, each thread maintains its own set of tasks. The owner thread stores and takes items from that set. Typically, when there are no more tasks in the set (the owner thread has nothing more to do), to keep busy, the thread can steal work items from other threads. Hence, in this scheme, only the owner thread can add tasks to its set, but all threads (including the owner) can take items from the owner's set.

This working set of items (called a work stealing queue from now on) supports three main operations: `put` and `take` which are used only by the queue's owner to insert and extract tasks, and `steal` which is used by other threads to steal work.

Current algorithms for work stealing queues comply with the following semantics: each inserted task is eventually extracted–by the owner thread or other threads–exactly once. However, these semantics are too restrictive for a wide range of applications dealing with irregular computation patterns. Sample domains include: parallel garbage collection, fixed point computations in program analysis, constraint solvers (e.g. SAT solvers), state space search exploration in model checking as well as integer and mixed programming solvers.

The key observation is that the correctness invariants of these applications allow for a *relaxation* of the traditional work stealing semantics. The fundamental reason is that in these problems: i) the application already ensures that no work is repeated, for example by checking whether a task is completed, or ii) the application tolerates repeatable work.

Informally, the relaxed semantics state that each inserted task should be eventually extracted *at least* once–instead of *exactly* once as it is with the conventional semantics. We exploit this invariant relaxation and introduce *idempotent work stealing*. We present several new algorithms that exploit the relaxed semantics to deliver better performance. Note that even with these relaxed semantics, subtle issues need to be handled in order to ensure correct and efficient operation. For example, the algorithms must guarantee that no tasks are lost and all extracted tasks contain valid and consistent information while at the same time avoiding the use of expensive synchronization instructions in the owner's operations: `put` and `take`.

On mainstream processors, existing algorithms for conventional work stealing queues require store-load memory ordering fence instructions in the critical path of the owner's operations [1, 6, 8, 10, 11]. A store-load fence prevents loads from being executed before the completion of stores to independent locations where the stores appear earlier in program order. In general, special atomic instructions and store-load fence instructions are substantially slower than regular instructions. Our new algorithms are designed to optimize the owner's operations by avoiding the high overheads of these instruction in the owner's operations. That is, in our algorithms, unlike existing algorithms, owner operations avoid using special atomic instructions and expensive store-load fence instructions.

We have evaluated ours and existing state-of-the-art algorithms with both microbenchmarks and representative non-trivial graph applications whose correctness invariants allow the usage of relaxed work stealing semantics. In particular, we performed experimental evaluation on several fundamental graph problems such as transitive closure and spanning tree computation for various graph types and sizes. The results indicate performance gains of up to 5x on microbenchmarks and up to 3x on graph applications. On graph applications, gains of 40% are common.

The contributions of this paper are the following:

- Introducing the concept of idempotent work stealing, a useful relaxation of the conventional semantics, applicable to a wide-class of applications.
- New high-performance work-stealing algorithms that adhere to these new relaxed semantics while avoiding expensive synchronization in the critical path of the owner's operations.
- Experimental evaluation of our new and existing state-of-the-art algorithms. The results indicate that our algorithms often significantly outperform existing state-of-the-art algorithms.

The rest of the paper is organized as follows. In Section 2, we discuss related work and atomic and fence instructions. Section 3 describes the new algorithms in detail. Section 4 presents our experimental performance results. We conclude the paper with Section 5.

2. Background

2.1 Atomic and Fence Instructions

To build efficient and correct concurrent algorithms, implementations often rely on the use of special atomic and memory fence instructions.

Atomic Instructions: Current mainstream processor architectures support either Compare-and-Swap (CAS) or the pair Load-Linked and Store-Conditional (LL/SC).

CAS was introduced on the IBM System 370 [12]. It is supported on Intel and Sun SPARC processor architectures. In its simplest form, it takes three arguments: a memory location, an expected value, and a new value. If the memory location holds the expected value, the new value is written to it, atomically. A Boolean return value indicates whether the write occurred. If it returns true, it is said to succeed. Otherwise, it is said to fail.

LL and SC are supported on the PowerPC architecture. LL takes one argument: a memory location, and returns its contents. SC takes two arguments: a memory location and a new value. Only if the memory location has not been written since the current thread last read it using LL, the new value is written to the memory location, atomically. A Boolean return value indicates whether the write occurred. Similar to CAS, SC is said to succeed or fail, if it returns true or false, respectively. For architectural reasons, implementations of LL/SC, do not allow the nesting or interleaving of LL/SC pairs, and infrequently often allow SC to fail spuriously, even if the target location was never written since the last LL. These spurious failures happen, for example, if the thread was preempted or a different location in the same cache line was written.

For generality, we present the algorithms in this paper using CAS. As discussed in Section 3, if LL/SC is supported rather than CAS, simpler implementations are possible.

Fence Instructions: Mainstream processor architectures allow some independent memory accesses to be executed out of program order, for the sake of hiding memory access latency and hence improving performance in the general case where reordering memory accesses has no effect on correctness. These architectures provide fence instructions that allow programmers to enforce order among memory accesses–that otherwise could be reordered–if such an ordering is required for correctness. This situation typically occurs in the implementations of concurrent algorithms.

While processor architectures vary in their relaxation of memory access ordering, all mainstream processors require fence instructions for preventing loads being executed before the completion of stores to independent locations where the stores appear earlier in program order (i.e., enforce store-load ordering). For architectural and historical reasons, fences that enforce store-load order are typically quite expensive and take tens of processor cycles.

2.2 Related Work

There have been several published algorithms for work-stealing, all adhering to the strong semantics. In a paper by Frigo et. al. [8], the authors present the *THE* work stealing algorithm implemented in the Cilk language runtime [4]. That algorithm is based on Dijkstra's mutual exclusion protocol and uses locks in the `steal` operation and in the corner case when the queue is empty it uses locks in `take`. Another algorithm presented by Arora et. al. [1] presents a non-blocking double-ended work queue but in the worst-case requires unbounded memory even if the number of waiting tasks at any one time is bounded. The Chase-Lev algorithm [6] rectifies this situation while preserving the performance of the Arora et. al. algorithm.

The correctness of all of these algorithms depends on enforcing the order of a write before a read in the critical path of the owner's `take` operation. For example, in the Chase-Lev algorithm [6, Figure 3], the write in line 23 must be ordered before the read in line 24; in the Cilk THE algorithm [8, Figure 4], the write in line 5 must be ordered before the read in line 6. Similarly, for the algorithms by Arora et. al. [1], Hendler and Shavit [11], and Hendler et. al. [10].

3. Algorithms

3.1 Overview

In this section we describe in detail our algorithms for idempotent work stealing. The main motivation behind these algorithms is to exploit the relaxed semantics to deliver better performance. In particular, the relaxed semantics enable us to build algorithms which speed up the common path consisting of the owner's operations: `put` and `take`. We present three algorithms, each with a different choice for how the items are extracted. In all of our algorithms, the owner inserts new tasks at the tail of the queue. In the first algorithm (idempotent LIFO) tasks are always extracted from the tail, while in the second algorithm (idempotent FIFO) tasks are always extracted from the head. In the third algorithm (idempotent double-ended), the owner extracts from the tail while thieves extract from the head of the queue. We use the term queue loosely to mean a structure with items stored in the order in which they were inserted.

The key challenge we faced in designing our algorithms is how to avoid the need for special atomic instructions (i.e. CAS or LL/SC) and store-load ordering in our `put` and `take` operations while guaranteeing the following:

- No lost tasks: That is, each inserted tasks will eventually be extracted (one or more times).
- No garbage task information: That is, `steal` operations always return valid task information (i.e., that is safe to execute). The task information, which may span multiple words and hence may be read and written non-atomically, must be complete and consistent and represent an actual task that was inserted into the work queue.

Ordering Requirements: In all three of our algorithms, the `take` operation does not require any special ordering among memory accesses beyond what is implied by data dependence. This is in contrast to existing work stealing algorithms [1, 6, 8, 10, 11], which all require a store-load fence in the critical path of the `take` operation. Store-load ordering requires a fence instruction on all mainstream architectures (e.g., `sync` on PowerPC and `mfence` or `lock` prefix instructions on Intel X86). The avoidance of these fence instructions in the common case is crucial to improving performance.

In the `put` operation of each of the three new algorithms, the writing of the task information into the queue structure must be completed before updating the tail index, in order to guarantee that interference with concurrent `steal` operations does not lead to lost items (and hence a violation of the correctness criteria), or the extraction by a `steal` operation of invalid task information that may lead to unpredictable errors or failures. On architectures such as Intel X86 and Sun Sparc (with total store order), no special fence instructions are needed. On PowerPC, a light-weight fence instruction (`lwsync`) is needed. Existing work stealing algorithms [1, 6, 8, 10, 11] require the same store-store ordering in their `put` operations for the same reasons why it is needed in ours. Hence, this is not a new overhead added in our algorithms.

Ordering requirements for the `steal` operations are indicated in each of the algorithm listings in Figures 1, 2, and 3.

ABA Problem and Prevention: The ABA problem is common in non-blocking algorithms, mostly in relation to the use of CAS. It was first encountered in a free list implementation listed in the IBM System 370 documentation [12]. Typically, the ABA problem occurs when a thread reads some value A from a shared variable, and then other threads write to the variable some value B, and then A again. Later, when the original thread checks if the variable holds the value A, e.g., using CAS, the comparison succeeds, while the intention of the algorithm designer is for such a comparison to fail in this case, and to succeed only if the variable has not been written after the initial read. However, the semantics of CAS prevent it from distinguishing the two cases. The classic solution for the ABA problem [12] is to pack a tag with the shared variable and increment the tag when the associated variable is updated, so that other threads can detect that the variable has been updated. This solution assumes that the tag is large enough that it is unlikely to wrap around and reach the same value while a thread is executing the read-check scenario mentioned above. The packing of a tag with index variables in one atomic word limits the size of the index.

Two of the algorithms (idempotent LIFO and idempotent double-ended) need to guard against the ABA problem in the `steal` operation as discussed below in more detail.

Our algorithms are presented using CAS and ABA prevention tags. However, the tag is a specific implementation choice. At the abstract level, the algorithms do not require the use of the tag mechanism but can use any ABA prevention mechanism. For example, the PowerPC architectures supports the LL/SC instructions, which are inherently immune to the ABA problem. In such a case, there is no need for the tag, because we can replace the read and the CAS of the index variable in the `steal` operations by LL and SC, respectively. In the absence of hardware support for LL/SC, software mechanisms can be used to simulate them without using tags packed with values [14].

Expanding and shrinking: We present in detail for each algorithm how to expand the queue size, where task arrays are replaced by new larger ones. The shown algorithm code assumes support for automatic garbage collection, where old arrays are freed automatically. Without garbage collection, buffer pools as described by Chase and Lev [6] can be used. The old array can be remembered in the expand() operation and then freed to the buffer pools right after the end of the `put` operation. As for the actual tasks, they are not dynamic objects. They are written and read directly to and from elements of the task array.

3.2 Algorithm with LIFO Extraction

In the idempotent LIFO algorithm (Figure 1), the queue is represented by an array of tasks, and an anchor variable that contains two subfields: the index of the tail of the queue and an ABA prevention tag. The capacity of the queue (i.e., the size of the tasks array) can be changed only by the owner and hence only the owner accesses the capacity variable. Initially, an empty array of some size is allocated and the tail of the queue is set to 0, indicating that the queue is empty.

Put: In the `put` operation, the owner starts by reading the anchor variable in line 1. In line 2, the owner checks if there is enough space to put the new task. If not it expands the array and restarts. For brevity and simplicity, we assume that the owner will be able to expand the array. In an actual implementation, there should be a check if the expansion succeeded or not. If there is enough space in the array, the thread proceeds to line 3 and writes the task information into the task array. In general, this write need not be atomic and thus the task information may span multiple words.

Finally, in line 4, the thread writes to the anchor variable the two packed values read in line 1 with each of the values incremented by one.

Take: In the `take` operation, the owner starts by reading the anchor variable in line 1, then checking in line 2 if the queue is empty (i.e., if $t = 0$). If so, the operation returns an indicator of an empty queue. Otherwise, it proceeds to line 3 and reads the task information from the array element with index $t-1$.

In line 4, the thread writes to the anchor variable the two packed values read in line 1, with the tail index decremented by one.

Steal: A thread starts the `steal` operation by reading the anchor variable in line 1. In line 2, the thread checks if the queue is empty (i.e., if $t = 0$). If so, the operation returns an indicator of an empty queue. Otherwise, it proceeds to line 3 and reads a pointer to the tasks array. The read in line 3 must be ordered after the read in line 1. Otherwise, a thief may read a stale pointer to the tasks array and then get a tail index from the anchor variable that is inconsistent with the stale array, if in the meantime the owner expanded the size of the queue.

In line 4, the thread reads the array element with index $t-1$. Reading the task information need not be atomic as it is synchronized by being ordered in between the read in line 1 and the CAS in line 5, and thus the task information is allowed to span multiple words.

```
   Structures:
      Task: task information
      LifoIwsq:
         anchor: ⟨integer,integer⟩; // ⟨tail,tag⟩
         capacity: integer
         tasks: array of Task

   constructor LifoIwsq(integer size) {
      anchor := ⟨0,0⟩;
      capacity := size;
      tasks := new Task[size];
   }

   void put(Task task) {
      Order write in 3 before write in 4
1:    ⟨t,g⟩ := anchor;
2:    if (t = capacity) {expand(); goto 1;}
3:    tasks[t] := task;
4:    anchor := ⟨t+1,g+1⟩;
   }

   TaskInfo take() {
1:    ⟨t,g⟩ := anchor;
2:    if (t = 0) return EMPTY;
3:    task := tasks[t−1];
4:    anchor := ⟨t−1,g⟩;
5:    return task;
   }

   TaskInfo steal() {
      Order read in 1 before read in 3
      Order read in 4 before CAS in 5
1:    ⟨t,g⟩ := anchor;
2:    if (t = 0) return EMPTY;
3:    a := tasks;
4:    task := a[t−1];
5:    if !CAS(anchor,⟨t,g⟩,⟨t−1,g⟩) goto 1;
6:    return task;
   }

   void expand() {
      Order writes in 2 before write in 3
      Order write in 3 before write in put:4
1:    a := new Tasks[2*capacity];
2:    for i = 0:capacity−1, a[i] := tasks[i];
3:    tasks := a;
4:    capacity := 2*capacity;
   }
```

Figure 1. Idempotent work stealing queue with LIFO extraction.

Finally, after the task information is read in line 4, the CAS in line 5 checks that since the execution of the read in line 1, the owner has not overwritten the array element with index t−1. By using the tag packed with the tail index in the anchor variable, we aim to avoid the following scenario that may occur if the ABA problem is not addressed. A thread X (thief) executing a `steal` operation reads the tail index t from the anchor variable in line 1. Then it reads the information of some task a from tasks[t−1] in line 4. In the meantime another thread extracts task a from tasks[t−1] and sets the tail index to t−1 and then the owner puts a new task b in the now available space in tasks[t−1] and sets the tail index back to t. Finally, in line 5 of the `steal` operation, thread X performs CAS on the tail index that succeeds (since it now has value t) and sets the tail index to t-1. The `steal` of thread X returns task a. The problem is that task b is lost as it is never extracted and its entry in the tasks array will be overwritten by the next `put` operation by the owner.

The use of the tag and incrementing it on every `put` guarantees that the CAS in line 5 will not succeed if the owner has written items (in particular, with index less than t) in the tasks array between the execution of lines 1 and 5 of the `steal` operation. In the otherwise problematic scenario above, but using the tag, thread X would detect in line 5 that the tag has changed and restart the `steal` operation, avoiding the incorrect outcome.

Another variation of the above problematic scenario is also avoided by preventing the ABA problem. In this variation, thread X reads the tail index t from the anchor variable in line 1 of `steal`. Then the owner performs a `take` operation that extract task a from tasks[t−1] and decrements tail to t−1. Then the owner starts executing a `put` operation of a task b. While the owner is in the process of overwriting the information of task a with the information of task b in line 3 of `put`, thread X reads the task information in tasks[t−1] in line 4 of `steal`. Then the owner updates tail to t again in line 4 of `put`, and thread X's CAS in line 5 of `steal` succeeds, and thread X ends up with invalid task information that may be an inconsistent mix of the information of tasks a and b.

Expand: For the owner to expand a full queue, it allocates a new larger array (e.g., with double the current capacity) in line 1. Then it copies the tasks from the current array to the newly allocated one in line 2. After that, it sets the tasks pointer to the new array in line 3. Note that the writes in line 2 (which need not be atomic) must be ordered before the write in line 3. Otherwise, a thief may read uninitialized task information. Finally, the owner updates the capacity variable (recall that the capacity variable is read and modified only by the owner).

The write in line 3, setting the tasks pointer to the new array, must be ordered before the subsequent write in line 4 of `put` that updates the anchor variable. Otherwise, a `steal` operation may observe a new tail index value and uses it to access a stale task array. This may result in an out of bound access to the task array or returning incorrect task information.

3.3 Algorithm with FIFO Extraction

In the idempotent FIFO algorithm (Figure 2), the queue is represented by an array of tasks, and two anchor variables, head and tail, that hold the indices of the head and the tail of the queue, respectively. The task array is encapsulated in a structure that contains both the array and its size. This structure simplifies expanding and shrinking the queue size.

Initially an empty array of tasks of some size is allocated and both the head and tail of the queue are set to the same value (e.g., 0), indicating that the queue is empty.

Put: The algorithm for the `put` operation starts by reading the head and tail variables in lines 1 and 2. In line 3, the owner checks if the queue is full (the difference between tail and head is equal to the size of the tasks array). If so, it expands it and restarts the `put` operation. If not, it proceeds to line 4 to write the new task information to the tasks array element with index t modulo the size of the array. The writing of the task information which can span multiple locations need not be atomic. Finally, in line 5, the owner writes to the tail variable the value t+1.

Take: The owner starts a `take` operation by reading the head and tail variables in lines 1 and 2. In line 3 if the values read from head and tail are equal, then the queue is found to be empty and an empty indicator is returned. If not, the owner proceeds to line 4 and reads the task information from the element of tasks array with

```
   Structures:
     Task: task information
     TaskArrayWithSize:
       size: integer
       array: array of Task
     FifoIwsq:
       head: integer;
       tail: integer;
       tasks: TaskArrayWithSize

   constructor FifoIwsq(integer size) {
     head := 0;
     tail := 0;
     tasks := new TaskArrayWithSize(size);
   }

   void put(Task task) {
     Order write at 4 before write at 5
1:   h := head;
2:   t := tail;
3:   if (t = h+tasks.size) {expand(); goto 1;}
4:   tasks.array[t%tasks.size] := task;
5:   tail := t+1;
   }

   TaskInfo take() {
1:   h := head;
2:   t := tail;
3:   if (h = t) return EMPTY;
4:   task := tasks.array[h%tasks.size];
5:   head := h+1;
6:   return task;
   }

   TaskInfo steal() {
     Order read in 1 before read in 2
     Order read in 1 before read in 4
     Order read in 5 before CAS in 6
1:   h := head;
2:   t := tail;
3:   if (h = t) return EMPTY;
4:   a := tasks;
5:   task := a.array[h%a.size];
6:   if !CAS(head,h,h+1) goto 1;
7:   return task;
   }

   void expand() {
     Order writes in 2 and 4 before write in 5
     Order write in 5 before write in put:5
1:   size := tasks.size;
2:   a := new TaskArrayWithSize(2*size);
3:   for i = head:tail−1,
4:     a.array[i%a.size] := tasks.array[i%tasks.size];
5:   tasks := a;
   }
```

Figure 2. FIFO idempotent work stealing queue.

index h modulo the array size. In line 5, the owner writes the value h+1 to the head variable.

Steal: A thread starts the `steal` operation by reading the head variable in line 1 and then the tail variable in line 2. In line 3, it checks if the values read from head and tail are equal. If so, then the queue is empty and an empty indicator is returned. Note that the order of the reads is required only to avoid returning an empty indicator for a queue that was never empty during the `steal` operation. This could happen if tail were to be read first, then some number of `put` and `take` operations are completed resulting in a value of head that is equal to or larger than the value read earlier from tail.

In line 4, the thread reads a pointer to the tasks array. The read of head in line 1 must be ordered before the read in line 4. Otherwise, a thief may read a stale pointer to the tasks array and then get a head index from the anchor variable that is inconsistent with the stale array, if the owner has expanded the array size in the meantime. In line 5, the thread reads the array element with index h modulo the array size. Reading the task information which may span multiple words need not be atomic.

Finally, the CAS in line 6 checks that the value of head is the same as that read in line 1. While it is possible for a `steal` operation to observe the same value of head at lines 1 and 5, even if the value of head has changed in the meantime, the algorithm's correctness does not require preventing this ABA situation (unlike in the idempotent LIFO algorithm), for the following reasons: Only a `take` operation by the owner can overwrite head with a smaller value. However, this can happen only as long as the owner has not observed higher values of head. Therefore, it is impossible that a `put` by the owner has overwritten the task read in line 4 of the `steal` operation. It follows that it is impossible that the thief has extracted a stale task instead of a new task (which otherwise can lead to a lost task situation), and that the task extracted by the `steal` operation is not corrupt or inconsistent.

If successful, the CAS in line 6 updates the head variable with value higher by one than that read in line 1 to indicate the extraction of a task.

Expand: For the owner to expand a full queue, it allocates a new larger array (e.g., with double the current capacity) in line 2. Then, it copies the tasks from the current array to the newly allocated one in lines 3 and 4. After that, it sets the tasks pointer to the new array in line 5. The writes in lines 2 and 4 must be ordered before the write in line 5. Otherwise, a thief may read uninitialized task information.

The write in line 5 must be ordered before the subsequent write in line 5 of `put` that updates the tail variable. Otherwise, a `steal` operation may return an old task, while the new task is lost without ever being executed.

3.4 Algorithm with Double-Ended Extraction

In the idempotent double-ended algorithm (Figure 3), the queue is represented by an array of tasks, and an anchor variable that is packed with three subfields indicating the index of the head of the queue, the size of the queue, and an ABA-prevention tag. The task array is encapsulated in a structure that contains both the array and its size.

Put: The owner starts the `put` operation by reading the anchor variable in line 1. In line 2, the owner checks if there is enough space to put the new task by checking if the size subfield is less than the size of the tasks array. If not, it expands the array and restarts. Otherwise, it proceeds to line 3 and writes the task information into the task array at the tail of the queue (i.e., h+s modulo the size of the array). The writing of the task information which can span multiple words need not be atomic.

Finally, in line 4, the owner writes to the anchor variable the three packed values as read in line 1 with the size and tag subfields each incremented by one, indicating the addition of a task and to prevent the ABA problem in concurrent `steal` operations as discussed below.

Take: The owner starts the `take` operation by reading the anchor variable in line 1, then checking in line 2 if the queue is empty (i.e., if s = 0). If so, the operation returns an indicator of an empty queue.

```
   Structures:
     Task: task information
     TaskArrayWithSize:
       size: integer
       array: array of Task
     Delwsq:
       anchor: ⟨integer,integer,integer⟩; // ⟨head,size,tag⟩
       tasks: TaskArrayWithSize

   constructor Delwsq(integer size) {
     anchor := ⟨0,0,0⟩;
     tasks := new TaskArrayWithSize(size);
   }

   void put(Task task) {
     Order write in 3 before write in 4
1:   ⟨h,s,g⟩ := anchor;
2:   if (s = tasks.size) {expand(); goto 1;}
3:   tasks.array[(h+s)%tasks.size] := task;
4:   anchor := ⟨h,s+1,g+1⟩;
   }

   TaskInfo take() {
1:   ⟨h,s,g⟩ := anchor;
2:   if (s = 0) return EMPTY;
3:   task := tasks.array[(h+s−1)%tasks.size];
4:   anchor := ⟨h,s−1,g⟩;
5:   return task;
   }

   TaskInfo steal() {
     Order read in 1 before read in 3
     Order read in 4 before CAS in 6
1:   ⟨h,s,g⟩ := anchor;
2:   if (s = 0) return EMPTY;
3:   a := tasks;
4:   task := a.array[h%a.size];
5:   h2 := h+1 % MaxSize;
6:   if !CAS(anchor,⟨h,s,g⟩,⟨h2,s−1,g⟩) goto 1;
7:   return task;
   }

   void expand() {
     Order writes in 2 and 4 before write in 5
     Order write in 5 before write in put:4
1:   ⟨h,s,g⟩ := anchor;
2:   a := new TaskCircularArray(2*s);
3:   for i = 0 : s−1,
4:     a.array[(h+i)%a.size] := tasks.array[(h+i)%tasks.size];
5:   tasks := a;
   }
```

Figure 3. Double-ended idempotent work stealing queue.

Otherwise, it proceeds to line 3 and reads the task information at the tail of the queue, i.e., from the array element with index h+s−1 modulo the array size.

In line 4, the owner writes to the anchor variable the three packed subfield values read in line 1 with the size subfield decremented by one to indicate the extraction of a task.

Steal: A thread starts the `steal` operation by reading the anchor variable in line 1. In line 2, the thread checks if the queue is empty (i.e., if $s = 0$). If so, the operation returns an indicator of an empty queue. Otherwise, it proceeds to step 3 and reads a pointer to the tasks array. The read in line 3 must be ordered after the read in line 1. Otherwise, a thief may dereference a stale pointer to the tasks array after it has been replaced by the owner in order to expand the queue, which may lead to a lost task scenario.

In line 4, the thread reads the array element with index h. Reading the task information which can span multiple words need not be atomic, as the reading is synchronized by being ordered in between the read in line 1 and the CAS in line 5.

Finally, the CAS in line 6 checks that values of three subfield of anchor are the same as when read in line 1. The checking of the tag subfield (or ABA prevention in general) guarantees that since the read in line 1 the owner has not overwritten the array element with index h modulo the array size. That is, the task information read in line 4 was indeed consistent and that no task was lost. The CAS in line 6, if successful, updates the anchor variable with the values read in line 1 except with the head subfield incremented (modulo some size bound) to indicate the stealing of a task.

Expand: Array expansion for this algorithm is similar to the array expansion for the idempotent FIFO algorithm.

4. Performance Results

We have implemented all of our algorithms as well as Chase and Lev's [6] algorithm and the *Cilk THE* [8] work stealing algorithm presented with the Cilk parallelizing runtime [4]. The Chase-Lev algorithm is an improvement of the algorithm presented in Arora et. al. [1] and is targeted at avoiding the limitation of using fixed-sized arrays.

4.1 Benchmarks

We first studied the algorithmic upper limits that one can expect from our algorithms over existing ones. In order to perform such a limit study, we developed our own parametric framework where we can control the cost of each task. This allowed us to easily experiment with relative costs of tasks: from zero cost as studied in [6] to more expensive computations. In particular, the measurements obtained on tasks with zero cost allowed us to more accurately compare the differences in the operations of the various work stealing structures. As discussed later, the limit study indicated that some of our algorithms often performed a factor of 10 better than any of the existing ones.

Next, we evaluated our algorithms on challenging problems in high performance computing. For this work, we selected two graph problems of irregular nature: i) transitive closure computation and ii) spanning tree computation, both for an undirected connected graph. In both cases, the challenge is to speed up the computation via parallel traversal of the graph. Because the workload is irregular, work stealing becomes of key importance in implementing an efficient load balancing scheme. Both of these applications fit naturally with our approach. In the case of transitive closure, the application can tolerate some work to be repeated (and hence does not use synchronization mechanisms to coordinate the parallel threads during graph traversal). Conversely, in the case of spanning trees, the application requires synchronization (either in the form of compare-and-swap or memory barriers) in order to make sure it operates correctly (e.g. the result at the end of the computation is a spanning tree). Hence, the application already uses synchronization to check for repeatability of work. Using a diverse set of graph types and graph sizes, we have evaluated the work stealing algorithms on these two applications. The results indicate performance gains of up to 300% and gains of 20-40% are common.

4.2 Experimental Results

4.2.1 Microbenchmark Results

We use a microbenchmark to estimate the upper limits of the performance gains of using our algorithms relative to existing algorithms.

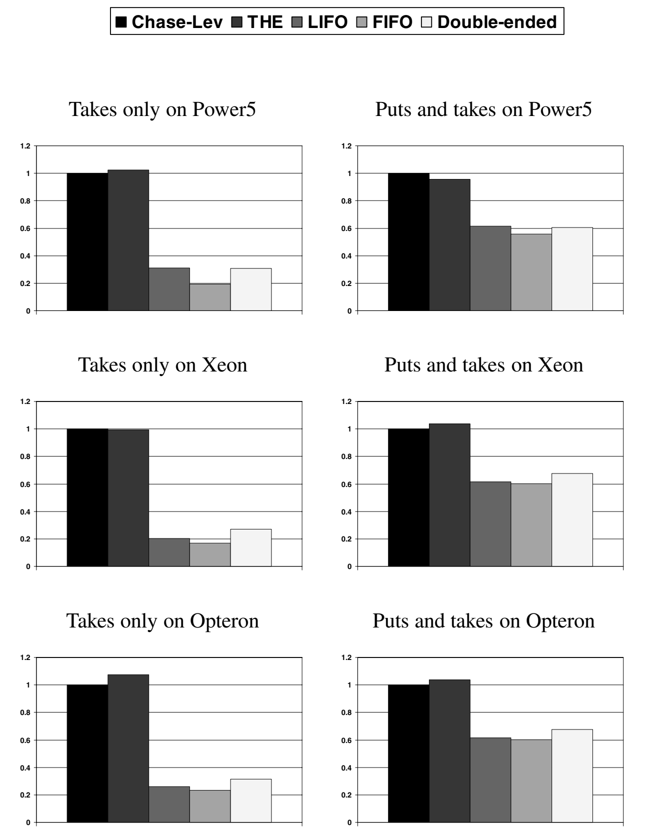

Figure 4. Execution times normalized to Chase-Lev.

The key objective of our algorithms is minimizing the overheads of the common case, which is the owner's `put` and `take` operations. So, in the microbenchmark measurements we focus on the owner's operations, where the owner performs a number of puts (10 millions) followed by an equal number of takes without executing any work.

We ran these experiments on three processor architectures: 3.8 GHz hyperthreaded Intel Xeon (Netburst), 2.4 GHz AMD Opteron, and 1.5 GHz dual-threaded Power5. The store-load ordering in the Chase-Lev [6] and THE [8] algorithms is enforced by using the `sync` instruction on PowerPC. On x86 architectures (Intel and AMD), either the `mfence` instruction or some types of `lock` prefix instructions can be used [13]. We tried both and found that the use of `mfence` resulted in substantially higher overheads than the use of `lock` prefix (approximately double the overhead). As a principle, we always report the more conservative results, i.e., less favorable to our algorithms. That is, for a conservative comparison of our algorithms with the THE and Chase-Lev algorithm, we report results for the implementations with the lower overheads (i.e., `lock` prefix) for the existing algorithms. We do so also for the results of the graph applications described later. Results using `mfence` amplified the gains of our algorithms by a factor of 1.5x-2x.

We report results for executing both puts and takes and for executing takes only. The latter case is relevant to applications where work queues may be populated privately in earlier phases of computation. Note that the experiments do not invoke takes on empty and one item queues, as these cases are slow corner cases for the THE and Chase-Lev algorithms, respectively. That is, our result exercise common case execution paths for all the algorithms.

Figure 4 shows the execution times normalized to that of the Chase-Lev algorithm. The benchmark results show significant speedups of our idempotent LIFO, FIFO, and double-ended algorithms over the THE and Chase-Lev algorithms in the ranges of 1.55x–4.92x, 1.66x–5.87x, 1.47x–3.69x, respectively.

4.2.2 Irregular Graph Applications

To specify and evaluate the transitive closure and spanning tree algorithms, we used the SIMPLE framework, described in Bader et. al. [3]. A detailed discussion of the spanning tree algorithm can be found in Bader and Cong [2]. The algorithm already uses a form of work-stealing to ensure load balancing. We adapted the algorithm to run on architectures which do not provide sequential consistency. The key point now is that repeatable work is detected at the application level which allows us to use more relaxed semantics (e.g. our algorithms) for the work-stealing operations.

We have evaluated all of the algorithms on a 8-core 2.4GHz AMD Opteron on 3 different types of graphs used in the works of [2, 9] and others:

- Geometric Graph (Kgraph): These are *k*-regular graphs where each vertex is connected to its *k* neighbors.
- 2D Torus Graph: In this graph, the vertices are positioned on a 2D torus, with each vertex connected to its 4 neighbors.
- Random Graph: A random graph of *n* vertices together with *m* randomly added unique edges.

All of the graph structures are implemented using the standard adjacency matrix representation. The results of our experiments are shown in Figures 5-10. The reported times (speedups) are the best times computed from five separate runs with the same graph and the same set of starting roots. Similar results were obtained when we measured the median time (throwing out the best and worst times and computing the median of the remaining 3). We ran each experiment several times because some variation is expected due to the way parallel threads interleave in processing the graph. The figures represent the speedups with respect to the time it takes to complete Chase-Lev with a single thread. We present results only for the transitive closure application with two graph sizes for each graph type. The results for other graph sizes as well as the spanning tree application are similar and are omitted from the presentation.

In the case of Kgraph, where each neighbor has 3 vertices, the system scales up to 5 or 6 threads (as shown in Figures 5 and 6). Regardless of the number of threads, the idempotent LIFO algorithm consistently outperforms Chase-Lev and Cilk THE by 15-20%, and sometimes by 50% (with 2 million vertices and 7 threads). In the very rare cases where idempotent LIFO is not the best performing algorithm, we have observed that it is usually within 2-3% of the fastest one. Additionally, in almost all experiments idempotent FIFO outperforms Chase-Lev and Cilk THE, but is slower than idempotent LIFO. For Kgraph, these figures are representative of the type of performance gains we observed when running the system with a wide variety of graph sizes.

In the case of 2D Torus graphs (shown in Figures 7 and 8), we observe that with more than two threads, the application does not scale. However, idempotent LIFO significantly outperforms all others, regardless of the number of threads, often by a factor of 3. Before the point of scaling, idempotent FIFO is usually the worst of all the algorithms, but after the point of scaling, in this case for

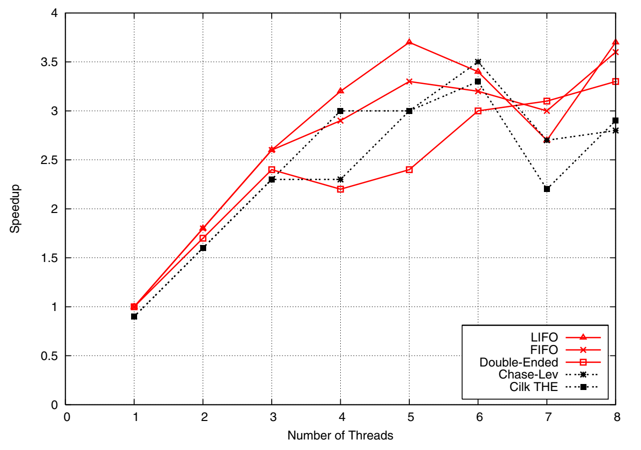

Figure 5. Kgraph: 1,000,000 vertices, 3 neighbors

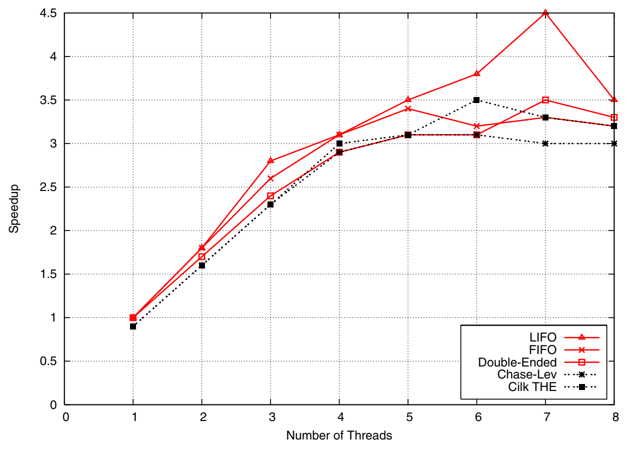

Figure 6. Kgraph: 2,000,000 vertices, 3 neighbors

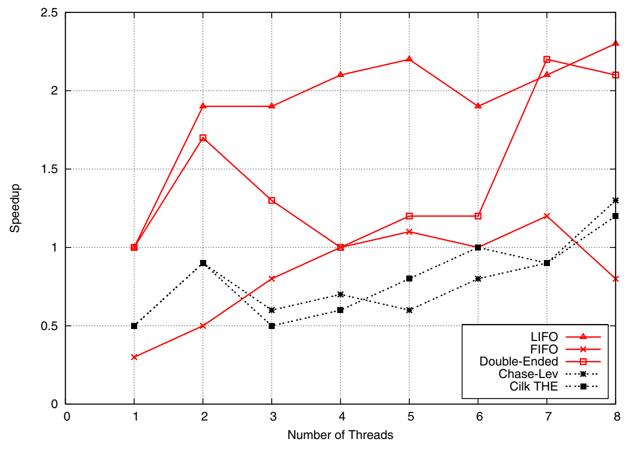

Figure 7. 2D Torus: 1,000,000 vertices

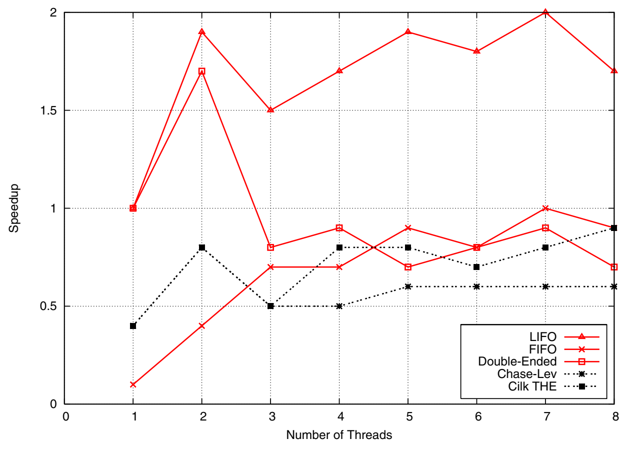

Figure 8. 2D Torus: 6,000,000 vertices

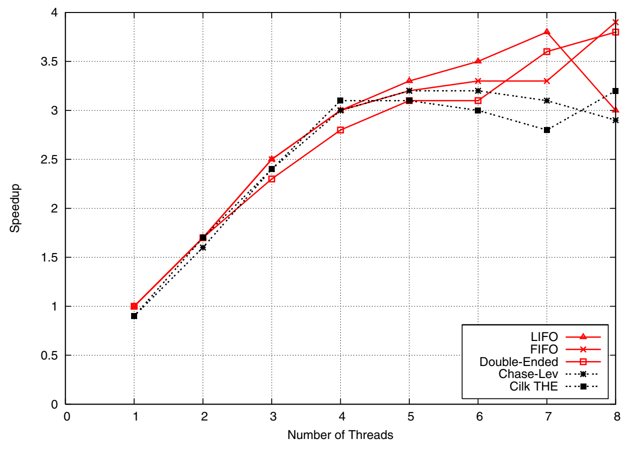

Figure 9. Random: 1,000,000 vertices, 3,000,000 edges

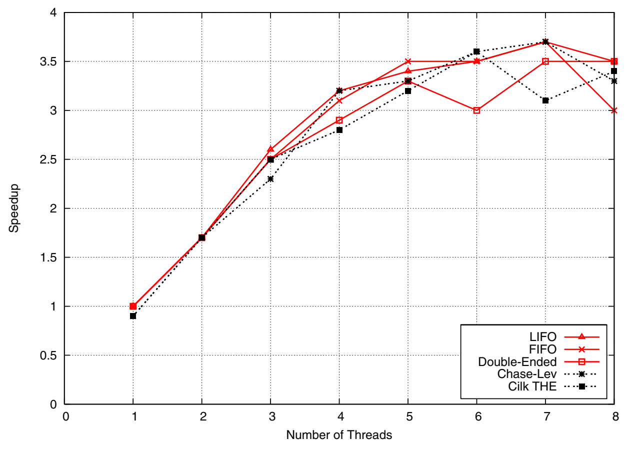

Figure 10. Random: 2,000,000 vertices, 6,000,000 edges

3 or more threads, idempotent FIFO may be better than Cilk THE and Chase-Lev by up to 70%.

In the case of Random graphs (shown in Figures 9 and 10), we have found that the various algorithms produce similar performance results. Usually, idempotent LIFO is better than the rest of the algorithms by 2-10%, but the speed up rarely surpasses 15%.

In our experiments, the double-ended task queue is frequently outperformed by other algorithms. This may seem like a surprising result as one would expect that it should perform better due to the reduced contention in the `take` operations. We offer two explanations for this result. First, other algorithms may use simpler structures which leads to faster operations. Second, the graph applications do not exhibit much locality. Traversal from a vertex to its neighbors, mostly leads to unrelated cache lines. This results in cache misses regardless of whether the task is executed by the owner or a thief which negates the potential locality benefits of double-ended extraction.

Finally, the relaxed semantics of our algorithms introduce the potential for executing speculative work that turns out to be redundant. We measured redundant work and found that on average, 2% of tasks are redundant. Further, in all experiments, at most 6% of redundant tasks were observed.

4.2.3 Discussion

The results presented here are a representative sample of many different experiments that we performed. For instance, among other things, we tried different graph sizes and configurations as well as different representations of the graph (using pointers instead of indices).

For graph algorithms, the time for each individual task can be relatively high, even though the task performs short code sequences for each vertex (e.g. marking and recording its neighbors if not already visited). The time per task can be dominated by cache misses proportional to the number of neighbors of a vertex. Because of the high cost of cache misses, the overall execution can be dominated by the tasks themselves and not by the work queue algorithms. However, the presence of memory barriers can exacerbate the cost of cache misses in the tasks as it hampers architectural mechanisms for latency hiding of cache misses. So, even though the cost of work queue manipulation is small relative to tasks, the impact of avoiding fences has a global effect and goes beyond the mere cost of work queue manipulation.

Optimizing the graph representation so that memory locations of neighbors are close together (e.g. one can sort the children indices of each vertex) may reduce cache misses to some degree. This is likely to be even more advantageous in enhancing the gains from our algorithm, since lower task costs amplify the impact of the work stealing operations. For example, when comparing the algorithms on smaller graphs (up to 250,000 vertices), we observed substantial gains even for Random graphs, since smaller graphs are more likely to fit completely in the cache and hence the performance is dominated by the cost of the work stealing operations and not by the cost of tasks. We only reported results for large graph sizes as they are more relevant to real uses of work stealing.

Regardless of the application in question, our experimental results strongly indicate that idempotent LIFO is the preferred work stealing structure for problems whose correctness properties are not violated by the relaxed semantics that our algorithms provide.

5. Conclusion

In this paper we introduced the concept of idempotent work stealing, a useful relaxation of the conventional work stealing semantics. The relaxation of the semantics, where tasks are extracted at least once instead of exactly once, is applicable to a wide class of irregular applications relying on work stealing. We presented new concurrent algorithms that exploit the relaxed semantics to deliver lower overheads than existing algorithms that support the conventional work stealing semantics. Our algorithms do not require the use of special atomic instructions or costly store-load fence instructions in the common case, that is, the owner put and `take` operations on the work stealing structure. The benefits are demonstrated by our experimental evaluation in comparison to existing state-of-the-art work stealing algorithms using graph applications and microbenchmarks. The performance gains of our algorithms are evident even on graphs with millions of vertices. In particular, our idempotent LIFO algorithm outperforms the existing algorithms in nearly all cases, sometimes by a factor of 3 and gains of 40% are common.

Acknowledgments The authors thank the reviewers for valuable comments and suggestions. We also thank Guojing Cong for helping us set up the infrastructure required for our graph experiments.

References

[1] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA*, pages 119–129, June 1998.

[2] David A. Bader and Guojing Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 65(9):994–1006, September 2005.

[3] David A. Bader and Joseph JáJá. SIMPLE: a methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, July 1999.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pages 207–216, October 1995.

[5] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the Twentieth Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 519–538, October 2005.

[6] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 21–28, July 2005.

[7] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the First Java Virtual Machine Research and Technology Symposium, JVM*, pages 21–21, April 2001.

[8] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI*, pages 212–223, June 1998.

[9] John Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA*, pages 16–25, June 1994.

[10] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18(3):189–207, 2006.

[11] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC*, pages 280–289, July 2002.

[12] *IBM System/370 Extended Architecture, Principles of Operation*, 1983. Publication No. SA22-7085.

[13] Doug Lea. *The JSR-133 Cookbook for Compiler Writers*. Web page.

[14] Maged M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In *Proceedings of the Eighteenth International Conference on Distributed Computing, DISC*, pages 144–158, October 2004.