Efficient Synchronization-Free Work Stealing

Umut A. Acar Arthur Charguéraud Mike Rainey

Max-Planck Institute for Software Systems
{umut,charguer,mrainey}@mpi-sws.org

Abstract

Work stealing is a classic algorithm for scheduling parallel computations on parallel/multicore computers. Due to its many desirable properties, work stealing is used widely in practice, especially in shared memory systems. All known shared-memory implementations of work stealing rely on synchronization primitives to enable safe distribution of parallel tasks among parallel processors. We describe a synchronization-free work-stealing algorithm for shared-memory systems. Our algorithm can be used to execute any computation DAG and has the same observable behavior as the traditional work-stealing algorithm but requires no expensive synchronization primitives (e.g., memory fences, compare-and-swap, fetch-and-add instructions). We provide a prototype implementation of the algorithm and perform an experimental evaluation with several benchmarks. Experiments show that the algorithm remains competitive with or improve over the traditional work-stealing algorithm.

1. Introduction

To ensure correct execution on multiprocessor or multicore computers, concurrent algorithms and data structures critically rely synchronization primitives. One class of syncronization primitives consists of *atomic instructions* that allow the algorithm designer and implementor to control the interleaving between instructions streams of different processors. Common atomic instructions include compare-and-swap, and load-linked-store-conditional. While atomic instructions crucially help in designing and implementing correct concurrent algorithms and data structures, they alone are often insufficient. This is because modern multiprocessors also allow individual processors to reorder instructions, e.g., the order between a memory load and a store may be altered. Such reordering of instructions weakens the memory model substantially. Modern multiprocessors therefore provide *memory fences* or *memory barriers* to allow control over the reordering of instructions. We refer to atomic instructions and memory barriers together as *synchronization instructions*.

While synchronization instructions are crucially important to the design and implementation of concurrent software, they are also expensive. They are known to take tens even hundreds of cycles, e.g., a load-after-store (read-after-write) barrier requires flushing the store buffers to the memory. Since some synchronization instructions have to restrict usage of the memory system by limiting multiple processors from accessing memory, e.g., by locking down the memory bus, they can also harm scalability of algorithms that use them to large number of processors. It is therefore desirable to develop algorithms and implementation that are *synchronization free*, i.e., that require no synchronization instructions. Completely eliminating all synchronization instructions in an implementation of an algorithm on a modern or future multicore, however, is likely nearly impossible, because some control over re-ordering will likely be necessary. We therefore relax this definition slightly to include only expensive synchronization instructions, which we define to be all atomic instructions and the store-load (read-after-write) memory barriers.

While highly desirable, designing and implementing synchronization free algorithms is challenging. For example, Herlihy showed that wait-free implementation of many common algorithms must use atomic synchronization operations such as compare and swap [18, 19]. Attiya et al [3] show that even if algorithms can avoid atomic instructions, they may have to use memory fences. These impossibility results highlight a fundamental challenge in parallel computing with modern multiprocessors: many key algorithms employed by a broad range of software artifacts appear to be destined to rely on expensive synchronization primitives. Indeed, no synchronization-free algorithms are known even for important, widely used algorithms.

Work stealing is an algorithm that is crucial to many modern multiprocessor applications. Going back to the eighties [8, 15], work stealing has been widely employed in scheduling of parallel computations. Its popularity and effectiveness is likely due to its many desirable properties, such as its simple structure, and its distributed pools, represented often as *deques* (doubly ended queues) that minimize congestion, and its provable efficiency [6]. Systems that employ work stealing include shared-memory systems such as Cilk [13], and X10 [9] as well as distributed systems [7, 11]

Earlier implementations of work stealing used blocking synchronization operations (lock) [13]. Arora et al [2] presented the first non-blocking work-stealing algorithm, which has subsequently been improved by Chase and Lev support unbounded deques [10]. While non-blocking, all of these algorithms require synchronization primitives including both memory fences and atomic instruction. In the light of recent results, this is not surprising: Attiya et al show that the traditional work-stealing algorithm with shared queues (deques) must use both atomic instructions and memory barriers [3]. Michael et al [22] present an implementation of work stealing algorithm that eliminates some synchronization instructions but not all. Unfortunately, even eliminating some of the synchronization instructions comes at the cost of losing generality: Michael et al's approach requires tasks to be idempotent, because

it disposes of a key invariant of work stealing—that each task is executed exactly once.

In this paper, we present a syncronization-free work-stealing algorithm for shared memory multiprocessors. The algorithm relies neither on expensive memory barriers nor atomic instructions. Our algorithm (Section 2) follows the outlines of the standard work stealing algorithm but carefully controls communication to eliminate all expensive synchronization. We represent each process as a *worker* and assign workers task queues that are entirely private, i.e., each task queue is accessed by its owner only. In addition to executing tasks, workers communicate to balance load and to notify tasks whose dependent tasks are completed. To communicate in a synchronization-free manner, workers use channels, each of which is assigned for communication between two specific workers. Since each channels is dedicated to a pair of workers, it can be implemented as a producer-consumer buffer without using any synchronization instructions. Our use of channels for communication between pairs of workers is similar to the work-dealing algorithm of Hendler and Shavit [17].

To perform load balancing and to ensure progress by detecting tasks that become ready for execution during the computation, our algorithm requires each worker to communicate with other workers periodically. Specifically, each worker periodically checks a randomly selected message buffer and handles the messages found; in addition each worker periodically sends a task to a randomly selected target worker if the target is idle and the worker has additional tasks to share. Since we employ no synchronization between workers, a specific kind of race can happen: a worker can conclude incorrectly that another worker as idle and send a task to that worker. Since all messages are processed, however, such races do not affect correctness.

Since workers operate on their private tasks queues, busy and idle workers do not collide; we thus avoid worker-thief synchronizations required in the traditional work stealing. Since idle workers simply wait to receive tasks to be shared with them, they do not collide; we thus avoid thief-thief synchronizations required in the traditional work stealing. Since workers notify dependent tasks via messages instead of updating a shared cell, they do not collide; we thus avoid worker-worker synchronizations required in the traditional work stealing. Since all our additional data structures including task queues and producer-consumer buffers are private to one or two workers, they can be implemented as relatively simply without additional synchronization primitives.

Our syncronization-free algorithm is fully general and can be used to execute any computation DAG. As an example, we show how to support the interface for parallel programming in the fork-join style (Section 3), which we also use in our experimental evaluation.

To show evidence that the algorithm can be realized in practice, we have completed a prototype implementation of the simple (non-improved) algorithm and compared it the state-of-the-art work-stealing algorithm [10] (Section 4). The experiments (Section 4.3) with several traditional work-stealing benchmarks show that our algorithm is competitive with traditional work stealing. Our experiments show that in cases where small tasks abound, our algorithm can perform nearly twice as fast. This suggests that our approach may be preferable in irregular parallel computations.

The contributions of this paper include the synchronization-free work-stealing algorithm, the modifications needed to make the algorithm efficient, its implementation, and its evaluation.

2. Algorithm

We give an overview of our algorithm and then present a detailed pseudo code that precisely specifies it. Our algorithm is fully general: as with traditional work stealing, it can be used to execute any computation DAG. When presenting the pseudo-code, we assume series-parallel computation DAGs, where each task has only one immediate descendant that is a "join" node, i.e., a node with incoming degree 2 or more. This assumption causes no loss of generality because a DAG can be transformed by splitting nodes appropriately or alternatively. It is also straightforward to extend the pseudo-code to support general DAGs directly.

2.1 Overview

Our algorithm closely follows the outline of the standard work stealing algorithm, but carefully controls communication to eliminate all expensive synchronization. As in standard works stealing, we represent each process as a *worker*. Each worker owns a private *task queue* that contains tasks. Workers communicate by sending each other messages via channels. A *message* holds either a task or an *acknowledgement* indicating that a task on which another task depends on has been executed to completion. Workers share their work load by exchanging task messages and notify each other of the completion of dependent tasks by exchanging acknowledgement messages.

A *task* is a closure consisting of a piece of code and an environment. When a worker creates a task, it owns the task and places the task in its task queue when that task becomes ready. When a worker needs a task, it removes one from its task queue and executes it. After the completion of a task, the worker sends an acknowledgment message to the owner of any other task that depends on the executed task. When a worker finds that its task queue is empty, it idles by repeatedly cycling through its message buffers to check for arriving messages until it receives a task.

Workers send and receive messages via a set of *channels*, each of which is private to two specific workers: for any two distinct workers i and j, the channel (i, j) is dedicated for messages from i to j and another channel (j, i) dedicated for messages from j to i. Consequently, there are $P^2 - P$ total message buffers.

Our algorithm relies critically on workers performing periodic communication in order to perform load balancing and to handle acknowledgement messages. Busy workers balance their load periodically by randomly selecting a target worker and, if this target is idle, sending a task to it. Workers receive messages by periodically polling a uniformly randomly selected channel. When a worker receives a message that contains a task, it starts working on the task. If the worker finds multiple tasks in its channel, then it places them into its queue, making them available for another idle processor. When a worker receives an acknowledgement message that is associated with a given dependent task, the worker checks whether the task is ready and places it into task queue only if the task is ready. A dependent task is ready when all of its dependent tasks have been executed.

2.2 Data structures

Figure 1 shows the main datatypes for our algorithm. We define a *closure* as a record consisting of a function (code pointer) and an array of arguments. With closures, we define tasks (`task`), continuations (`cont`), and joins (`join`). A *task* is a record consisting of a closure to execute followed by a continuation to perform. A continuation can be a simple return, similar to return from a call, a termination indicating completion, a single task to execute, or a join of some number of tasks. A join represents a task that has multiple incoming dependencies. Its type is that of a record consisting of a task to perform, an owner designated to perform the join, and a counter counting the number of completed dependencies of the join. Later, we will later relax the constraint that the only the owner performs the join. We define a message as a union (*i.e.*, sum type) of either a task or an acknowledgement of a task completion.

```
type closure = {fun_ptr fun, void* arg}

type task = {closure c, cont k}

type cont =
  RETURN | END | ONE(task* _) | JOIN(join* _)

type join = {task* task, int owner, int count}

type msg = MSG_TASK(task* _) | MSG_ACK(join* _)
```

Figure 1. Datatypes.

```
// shared variables
bool  is_idle[P]
pcb*  channels[P][P]

// worker-specific variables
int   my_id
deque my_tasks
cont  my_cont
```

Figure 2. Global and local (worker specific) variables.

```
type  deque
bool  deque_empty       (deque* d)
void  deque_push_bottom (deque* d, task* t)
void  deque_push_top    (deque* d, task* t)
task* deque_pop_bottom  (deque* d)
task* deque_pop_top     (deque* d)
```

Figure 3. The interface for deques.

```
type cell = {cell* t, msg m}

type pcb = {cell* front, cell* end}

msg no_msg = MSG_TASK(NULL)

pcb* pbc_create()
  cell* c = new cell(NULL, no_msg)
  return new pcb(c, c)

void push(pcb* pcb, msg m)
  pcb.end.m = m
  pcb.end.t = new cell(NULL, no_msg)

msg try_pop(pcb* pcb)
  cell* c = pcb.front
  if c.t == NULL then return no_msg
  msg m = c.m
  pcb.front = c.t
  free c
  return m
```

Figure 4. Producer-consumer buffers (PCB's) with linked lists.

As in traditional work stealing, we use doubly ended queues, *deque*s to implement task queues. Figure 3 illustrates the signature for deques. Since deques are private, however, they can implemented in a synchronization-free manner by using conventional techniques.

We implement channels using produce-consumer buffers (abbreviated as PCBs), that allow a single worker to produce and a single worker to consume. Figure 4 illustrates the pseudo code for PCB's implemented as linked lists. The consumer maintains a pointer to the front of the list while the producer maintains a pointer to the end of the list. The last cell of the linked list always remains unused. The buffer always contains at least one cell. Because we need one between every pair of distinct workers, the total number of channels is $P^2 - P$.

Our algorithm maintains a global state (Figure 2), which is shared and visible to all processors, and some local state private to each worker. The global state consists of idle flags (`is_idle`), indicating whether a worker is idle or busy, and a two-dimensional array of message buffers for communication (`channels`). The worker-local state consists of the id of the worker, a deque of tasks, and a continuation. The `id` is a unique integer between 0 and $P - 1$. The task deque `my_tasks` is, as we have already explained, a double-ended queue. The continuation `my_cont` is used to indicate what action a worker should perform after the completion of its current task.

2.3 The algorithm

Figure 5 shows the pseudo-code for our work-stealing algorithm. The algorithm consists of two main functions `work` and `wait` supported with several helper functions. The `work` function first checks whether the worker has a task on its local deque (`my_tasks`). If it finds its deque empty, the function `wait` is called, which sets the idle flag and repeatedly polls for messages using the helper `poll` function which we describe later on. Otherwise, the deque is non-empty and the worker pops a task from the bottom of its deque and executes it.

To execute a task, the worker first sets its local continuation (`my_cont`) to the continuation of the task and then invokes the task by applying the closure to the arguments. After the task completes, the worker checks its local continuation, which may have changed during the execution of the task, (e.g., if the tasks performed a "fork" operation). If the continuation is a return, no further action is needed. If the continuation indicates termination, an exception is thrown to terminate all workers properly. If the continuation is another task, this task is simply added to the bottom of the local deque. If the continuation is a join task, then the helper function `handle_join` is called.

The helper functions `handle_join` and `decr_join_count` implement a protocol for determining whether a join task (a task with multiple predecessors) is ready to be executed. The idea is to assign each join task an *owner*, which is naturally chosen to be the worker that creates the task, and have the owner collect acknowledgment messages regarding the completion of predecessor tasks that the join task depends on. When all predecessors are acknowledged, the owner places the task into its deque for execution. Any worker that completes an immediate predecessor of a join task sends an acknowledgment message to the owner when the predecessor completes its execution. A worker determines the owner by checking its continuation of the predecessor which has the information about the join task as well as the owner.

The helper function `handle_join` starts by checking (via `decr_join_count`) if the worker that executed the task is the owner of the join. If so, there is no need to send a message, as the worker can itself decrement the join counter. If the join counter reaches zero, the join task is pushed at the bottom of the local deque. Otherwise, if the worker is not the owner of the join task, then the worker sends a message to the owner to acknowledge completion of one of the immediate predecessors of the join task. In this case, the function `decr_join_count` is called by the owner on reception of the acknowledgement message. This protocol is correct because all the write operations peformed during the execution of the predecessor tasks occur before the owner receives the last acknowledgement message. The correctness of this protocol relies on the assumption that writes by one processor become visible in the same order to any other processor.

When executed by some worker, the helper function `poll` checks if the worker has received any messages from another ran-

```
void work()
  repeat
    if deque_empty(my_tasks) then wait() else
      task* t = deque_pop_bottom(my_tasks)
      my_cont = t.k
      t.c.fun(t.c.arg)
      free t
      handle_mycont()

void handle_mycont()
  match my_cont with
  | RETURN  → return
  | END     → throw Parallel_job_completed
  | ONE(t)  → deque_push_bottom(my_tasks, t)
  | JOIN(j) → handle_join(j)

void wait()
  is_idle[my_id] = true
  while deque_empty(my_tasks) do poll()

void handle_join(join* j)
  decr_join_counter(j)

void decr_join_counter(join* j)
  if j.owner == my_id then
    j.count--
    if j.count == 0 then
      if j.task ≠ NULL then
        deque_push_bottom(my_tasks, j.task)
      free j
  else
    push(channels[my_id][j.owner], MSG_ACK(j))

void poll()
  int id = random ∈ {0, .., P-1}\{my_id}
  repeat
    match try_pop(channels[id][my_id]) with
    | MSG_TASK(t) → if t == NULL then return
                    else deque_push_top(my_tasks, t)
    | MSG_ACK(j)  → decr_join_counter(j)
```

Figure 5. Pseudo-code for the work-stealing algorithm.

```
void communicate()
  poll()
  if not deque_empty(my_tasks) then deal()

void deal()
  int id = random ∈ {0, .., P-1}\{my_id}
  if not is_idle[id] then return else
    is_idle[id] = false
    task* t = deque_pop_top(my_tasks)
    push(channels[my_id][id], MSG_TASK(t))
```

Figure 6. Basic communication function to be called regularly.

domly selected worker, and handles each such message. If the message contains a task, then the task is pushed to the top of the local deque. If the message is an acknowledgement, then the helper function `handle_join` is called to determine whether the join task is ready.

The algorithm described thus far performs no load balancing. To perform load balancing, we rely on the workers to call the `communicate` periodically at fixed intervals. We discuss later in (Section 4) the techniques that can be used in practice to implement this regular communication phase. Figure 6 shows the pseudo-code for the communication functions. The function `communicate` first performs a poll by checking one of its message buffers (via a single call to `poll`). Then, in case the worker has a non-empty local deque, it attempts to share some of its work by calling the function `wait`. This function picks another worker randomly and checks whether the selected worker is idle. If so, it pops a task from the top of its local deque and sends it to the worker using the dedicated communication channel for this pair of workers.

```
type join = {..., bool ack_one}

void handle_join(join* j)
  if j.ack_one then
    deque_push_bottom(j.task)
    j.task = NULL
  else
    j.ack_one = true
  decr_join_count(a)
```

Figure 7. Join optimization

2.4 Optimized Joins

The join protocol that we have described previously has the advantage that it involves no synchronization instruction regardless of the incoming degree of the join nodes. However, this comes at the cost of a possible delay before the join task becomes ready. We now explain how to optimize the protocol in the case where the in-degrees of the join nodes are bounded by a small constant e.g., two. This situation typically appears in fork-join programs. We first explain for how to handle the case where the in-degree is at most two.

Consider the case of a task B that depends on two tasks A1 and A2. If a worker finishes one of the dependencies, say task A2, and if it is able to observe that the other dependency (task A1) is already finished, then this worker can start executing task B immediately. In order to implement this idea, we extend the join construct with an additional boolean field `ack_one`, with initial content false. Whenever a worker finishes one dependencies and sees the field `ack_one` unset, then this worker, in addition to sending a acknowledgement to the owner of the join, sets the field `ack_one`. Whenever a worker finishes one dependencies and sees the field `ack_one` set, then this worker can execute the join task immediately. This protocol ensures that when there is no race on the field `ack_one`, the join task is made available immediately.

One important observation is that it is not possible to deallocate the join record when acquiring the join task, because there might still exists a message referring to this join record. In order for deallocation of join records to be performed properly, we impose that an acknowledgement message be sent even when a worker sees the field `ack_one` set and acquires the join task. The worker makes sure to first set the task pointer contained in the join record to `NULL` so as to notify the owner of the join that it needs not execute the join task as well. This protocol is correct because we know that, in the eye of the owner, the write of the value `NULL` happens before the write of the acknowledgement message.

This optimized join protocol can be generalized to in-degrees greater than 2 in at least two different ways. One possibility is to use one field `ack_one` per dependency, and have each worker read all of them. In this case the overhead is quadratic in the in-degree. Another possibility is to replace `ack_one` with an integer field which plays the same role as the join counter with the difference that all workers can decrement it directly. In the case where no race occurs on this field, the join task will be available immediately after the last dependency completes. If, however, one or more race occurs, then only the owner will be able to schedule the join task. In summary, our optimized join protocol optimizes for the case where no race happens, while in the same time relying on the acknowledgement messages for recovering from races.

```
void fork_one(closure c, closure ck)
  task* tk = new task(ck, my_cont)
  deque_push_bottom(new task(c, ONE(tk)))
  my_cont = RETURN

void fork_two(closure ca, closure cb, closure ck)
  task* tk = new task(ck, my_cont)
  join* j = new_join(tk, my_id, 2)
  deque_push_bottom(new task(ca, JOIN(j)))
  deque_push_bottom(new task(cb, JOIN(j)))
  my_cont = RETURN

void start_scheduler(closure c)
  task* t = new task(c, END)
  push(channels[my_id][0], MSG_TASK(t))
```

Figure 8. Implementation of a fork-join library.

3. A Fork-Join Interface

Using the basic data types supporting continuations, tasks, and join's, and the operations on them, we can dynamically create an arbitrary computation DAG and allow our synchronization-free work-stealing algorithm evaluate the tasks in the DAG. Here, we describe as an example how to implement the fork-join interface.

The functions `fork_one` and `fork_two` allow the user to fork one or two tasks and a dependent continuation task specified by the closure `ck`. Both functions essentially create one task for each closure, make the current task to be the continuation for the given continuation task (`tk`), and create the desired relationship between the tasks and the continuation task. The function `start_scheduler` start the scheduler by creating a task for the specified closure with a `END` continuation and sending this task to an arbitrary worker. As we explained earlier on, when the scheduler encounters the `END` continuation, it knowns that the execution of the parallel job is complete.

As an example Figure 9 shows the code for the Fibonacci function implemented in fork-join style. The `fib` program computes the n^{th} fibonacci number for a given n using the standard mathematical definition. The program begins evaluating branches in parallel until the input size falls below cutoff parameter of our choice, at which point the program switches to a sequential version.

Observe from the code in Figure 9 that the programmer is responsible to handle the allocating and freeing of closure records and environments. We do not necessarily advocate this style of programming in practice, because it is well known from the work on the Cilk compiler [13] that such instances of memory management can be handled automatically and in an efficient manner with proper compiler support. We expect that our scheduling interface can be readily integrated into intermediate language of a compiler such as Cilk, so as to take advantage of higher-level programming constructs, such as Cilk's `spawn` and `synch` instructions.

4. Implementation and Evaluation

We compare our work-stealing algorithm with the state-of-the-art Chase-Lev work-stealing algorithm [10]. From hereon, we refer to our work-stealing algorithm as `our-ws` and the Chase-Lev algorithm as `std-ws`. Moreover, in this section we also study the effect of the delay between communication phases on the execution time.

4.1 Implementation

We implemented `our-ws` and `std-ws` inside our own, stand-alone C library, using POSIX threads (`pthreads`) to implement the workers.

The implementation of `our-ws` corresponds, up to minor code optimization, to the scheduling algorithm described in the pseudocode described in Section 2 and Figure 8, extended with the join optimization (Section 2.4). In this section, we outline a few properties of our implementation that are relevant to the performance results to follow.

```
type env = {int n, int* r}

type join_env = {int r1, int r2, int* r}

int cutoff // user defined

int fib_seq(int n)
  if n < 2
    return 1
  else
    return fib_seq(n-1) + fib_seq(n-2)

void fib_par(void* untyped_env)
  env* env = (env*) untyped_env
  int n = env.n
  int r = env.r
  free env
  if n < cutoff || n < 2
    *r = fib_seq(n)
  else
    join_env* jenv = new join_env(0, 0, r)
    env* env1 = new env(n-1, &(jenv.r1))
    env* env2 = new env(n-2, &(jenv.r2))
    fork_two(new closure(fib_par, env1),
             new closure(fib_par, env2),
             new closure(fib_join, jenv))

void fib_join(void *untyped_jenv)
  join_env* jenv = (join_env*) untyped_jenv
  int r1 = jenv.r1
  int r2 = jenv.r2
  int r = jenv.r
  free jenv
  *r = r1 + r2

int fib(int n)
  int r
  env* e = new env(n, &r)
  start_scheduler(new closure(fib_par, e))
  return r
```

Figure 9. Parallel fibonacci function.

Recall from Figure 6 the `communicate` function, which polls for messages, then deals tasks to idle processors. In our pseudocode, we assume that this function is called at a regular interval by each busy processor. It is crucial that this function be called regularly, because calling it is the only way in which work can be distributed away from a busy processor. But it is important for performance that the function is not called too often, as the polling and dealing cost slow down busy workers. In our implementation, we maintain for each processor a time stamp called `last_communicate`, which records the tiem at which the processor last called `communicate`. Then, we modify the `work` function (Figure 5), so that `communicate` is always called after a fixed delay. To obtain the time, we use the x86 time-stamp counter instruction, which provides a cheap and accurate way of measuring time. This technique is applicable under the assumption that no task from the computation DAG executes for longer than the time between two communication phases. There are a number of other mechanisms that may be more robust, such as the high-resolution interrupt timers used by OS-virtualiazation software or software polling. We did not implement these mechanisms because none of our benchmarks involved tasks that were large enough to increase delay noticably.

The implementation of `std-ws` follows the code from the Chase and Lev's paper [10], in which we placed the appropriate memory fences as described in Kuperstein et al [21]. For the representation of tasks and join records in `std-ws`, we used data structures very similar to that of `our-ws`.

In both the implementation of `our-ws` and `std-ws`, we had to carefully prevent GCC from reordering critical pairs of instructions, which could violate our memory-ordering invariants. In order to prevent the optimizations of GCC from performing unsafe reorderings, in addition to declare several variables `volatile`, we had to place *compiler barriers* of the form `asm volatile("":::"memory")` in a few locations. For example, such compiler barriers are required in-between the two write from the PCB `push` operation (Figure 4).

4.2 Benchmarks

We used five programs in our empirical evaluation. The first two generate synthetic work loads that are useful for measuring various scheduling costs. The `fib` program computes the n^{th} Fibonacci number using a naive doubly-nested recursive function, that is, with exponential complexity. Consequently, much of its execution time is spent making recursive calls and creating tasks, as the little computation it does is insignificant in front of the calling and scheduling costs. We show the implementation in Figure 9. The program begins evaluating branches in parallel until the input size falls below cutoff parameter of our choice, at which point the program switches to a sequential version. For this benchmark, we picked $n = 37$. The programs `cilksort`, `matmul`, and `heat` are adapted from Cilk programs, each of which is detailed in the Cilk manual [25]. Our `cilksort` benchmark sorts an array of 63 million 64-bit integers. Our `matmul` benchmark multiplies two 3500×3500 matrices. Our `heat` benchmark computes a 2D grid with 20,000 nodes in each of the two dimensions and takes 50 iteration steps. In addition to these benchmarks, we considered the `bal` program, which grows a complete binary tree of height 18 and which performs for each leaf task one single memory write at a random location inside a shared 1Gb array of 64-bit integers. This benchmark was intended to reveal the cost of the memory fence used in standard work stealing implementations (`std-ws`).

4.3 Experiments

Our test machine has four eight-core Intel Xeon X7550 processors running at 2.0GHz. Each core has 32Kb each of L1 instruction and data cache and 256 Kb of L2 cache. Each processor has an 18Mb L3 cache that is shared by all eight cores. The system has 1Tb of RAM and runs Debian Linux (kernel version 2.6.32.22.1.amd64-smp). We consider just 30 out of the 32 cores on the machine to reduce interference with the operating system. All of our code is compiled by GCC (v4.3.2) with the `-O3` option.

We present three experiments. The first two compare the performance of `ours-ws` and `std-ws`. In both experiments, we set `delay` to 50 microseconds. In the third experiment, we evaluate the robustness of `our-ws` under various delay settings.

Small tasks. In our first experiment, we compare performance of `our-ws` and `std-ws` on work loads that generate only small tasks. Having only small tasks enable us to analyze overheads precisely, because when tasks become too large, scheduling costs are typically masked by the task-execution times. The first benchmark we consider, `fib`, performs only local computation and involves little reading and writing to main memory. Recall that `fib` has a parameter called `cutoff`, which controls the point at which execution goes serial. In this benchmark, we set `cutoff` to its minimum value, which is two. We show the results in Figure 10. Because `fib` spends most of its execution time doing local computation, its performance is barely affected by the memory bus. Therefore, we expect that performance is unaffected by the memory fences and atomic operations performed by `std-ws`. Indeed, as we predicted, `fib` execution times of `ws-ours` and `ws-std` are very close. `ws-ours` is only behind `ws-std` by a small fraction. This result can be explained by the cost of polling and dealing in `ws-ours`.

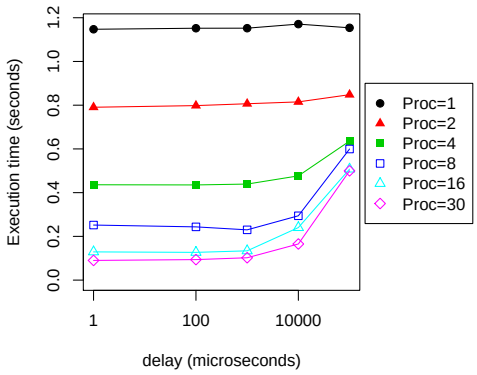

Figure 12. Runtime versus delay by number of processors for `fib`. The horizontal axis is on a logarithmic scale.

It should be clear that programs in which access to main memory is frequent are be slowed down by increasing traffic over the memory bus. We therefore expect to observe a slow down from the extra fences and atomic operations in `std-ws` on programs such as `bal` . Indeed, the results of the `bal` benchmark show that the fences and atomic operations in `std-ws` have a significant effect. Observe that, for more than eight processors, performance of `std-ws` is always at least 25% slower than that of `our-ws`. Although not plotted here, we observed that this slow down is consistent across various values for the tree height and can be as high as a factor of two.

Big tasks. Figure 11 shows, for each benchmark, the speedup curves of `our-ws` and `std-ws` side by side. [1] In the case where the task grain size is big, that is, at least tens of microseconds of CPU time, the performance of `our-ws` and `std-ws` are similar. This result is not surprising, because the overhead cost of scheduling is neglectable in front of the average task-completion time and because `our-ws` and `std-ws` are based on the same fundamental work-stealing algorithm.

We observed, like others [5], that the number of steals in practice is of the same order of magnitude as the number of steals predicted by the bound PH, where H is the number of forks in the critical path. Furthermore, we observed that the the `our-ws` and `std-ws` make about the same number of steals on average.

Delay. Perhaps surprisingly, we found that the performance of `our-ws` is quite robust even when delay between two communication phases is large. Figure 12 shows the execution time as a function of delay. On thirty processors, the execution time is unaffected until the delay exceeds one millisecond, which is 1% of the total execution time. After that, execution time slows down because processors are lacking work, and the curves in Figure 12 start going up exponentially. However, this effect is due to logarithmic scale used in the x-axis; on a linear scale, the runtime appears to grow linearly with the delay.

Summary. To summarize, our main results is that, for a program with fine-grain tasks that involves significant memory traffic, the performance of `our-ws` is better than that of `std-ws`. We also provide evidence that, in practice, our algorithm handles communication delay in a robust manner.

[1] For the baseline measurement of the speedup, we use the sequential version of the program.

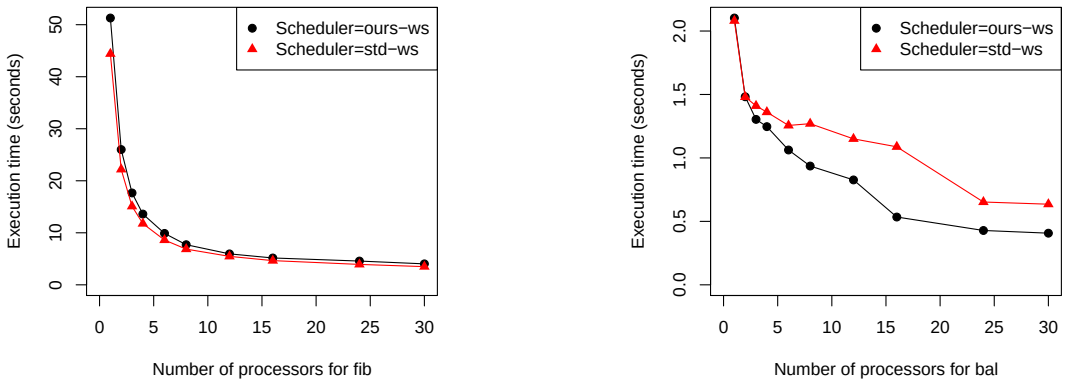

Figure 10. Runtime versus number of processors by scheduler.

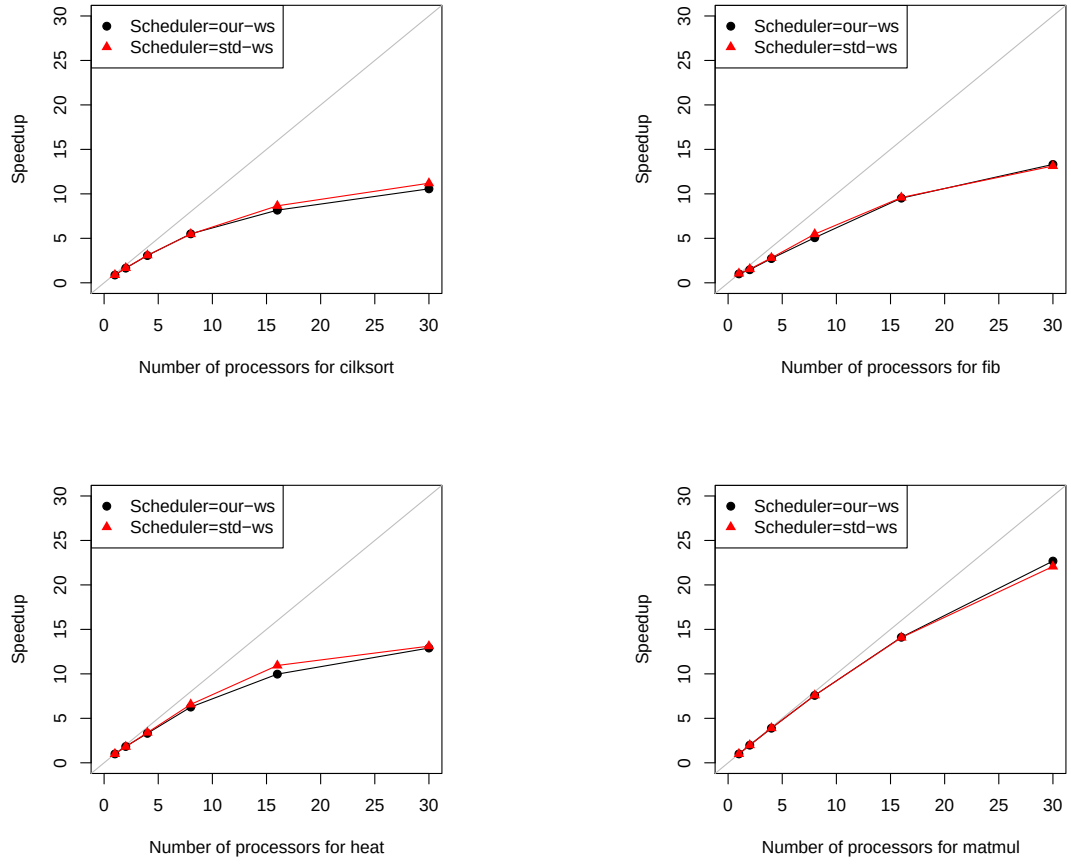

Figure 11. Comparison between our synchronization-free work stealing and standard work stealing.

5. Related Work

Load balancing. Load balancing is a key problem in multiprocessor computing, because load-balancing algorithms directly determine the run time on multiprocessors. Broadly speaking load balancing algorithms can be classified into offline approaches which only assign tasks to processors as tasks are created, perhaps based on some kind of estimation of their sizes, and online, *i.e.*, dynamic or adaptive, algorithms that move tasks between processors depending on the load. Many computations, especially the common forms that include irregular computations, require online load-balancing algorithms. In this paper, we are interested in online algorithms and have considered the work stealing algorithm [2, 8, 13, 15, 16]. Other online algorithms include the space-efficient depth-first search algorithm [4, 23].

To balance load, our synchronization-free work-stealing algorithm uses what is sometimes referred to as a *sender-initiated* load balancing, because it relies on the sender (the busy worker) rather than the receiver (the idle worker) to send tasks. In contrast, in *receiver-initiated* load balancing, it is the idle processors who perform load balancing by actively stealing tasks from busy processors. While our primary motivation for choosing this approach is elimination of synchronization primitives, the sender-initiated approach has been studied for different reasons. For example, Eager, Lazowska, and Zahorjan compare the sender- and reciever-initiated approaches and find the sender-initiated approach to be more effective in their simulations. Furuichi et al [14] study a sender-initiated load balancing technique in the context of a hierarchical scheduler for certain search problems.

Polling in work stealing. There are other implementations of work stealing that rely on polling as a mechanism for sending tasks between processors [12, 20, 24]. In these implementations, each worker thread regularly polls a single bit, which indicates whether a steal is being requested by an idle processor. When a busy worker sees its bit set, the worker synchronizes with the thief and sends it a task. These implementations are not synchronization free in the sense atomic operations are still necessary to handle the synchronization between the thief and the victim and the synchronization on the activation of join tasks. As such, they still suffer from heavy overheads when tasks are fine grain. It is worth nothing, however, that the polling versions can avoid expensive memory-fence operations on the deque, as each deque is private to its worker.

Work dealing. Sender-initiated load balancing have also been used to improve data locality in scheduling. Acar et al [1] extend work stealing to enable workers to send a task to another worker that the task may have affinity for, while also allowing tasks to remain in the worker's deques. This Hendler and Shavit [17] propose techniques to eliminate such synchronization by using producer-consumer buffers. Their approach, called work dealing, however, does not perform online load balancing: it only send tasks to workers when tasks are created. Once assigned to a worker, a task never moves. Furthermore, they don't consider synchronizing tasks such as those with multiple precedents (e.g., what we call "join" nodes). To support communication via messages in a synchronization-free manner, our algorithm uses producer-consumer buffers in a way similar to work dealing.

6. Conclusion

We have presented a synchronization-free work-stealing algorithm, presented an implementation of it, and studied its practical effectiveness. To achieve synchronization freedom, our algorithm assigns a private task queues to each worker, and uses message passing protocols to determine ready tasks and to perform load balancing. Our implementation shows that the algorithm can be made practical. When compared to traditional work stealing, our experiments show that our algorithm remains competitive and can outperform the traditional algorithm when small, memory-intensive tasks abound in the computation.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. *Theory of Computing Systems (TOCS)*, 35(3):321—347, 2002.

[2] 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 '98, pages 119–129. ACM Press, 1998.

[3] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 487–498. ACM Press, 2011.

[4] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46:281–321, March 1999.

[5] 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 *PPOPP '95: Proceedings of the fifth ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 1995.

[6] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[7] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 10–10, Berkeley, CA, USA, 1997. USENIX Association.

[8] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Functional Programming Languages and Computer Architecture (FPCA '81)*, pages 187–194. ACM Press, October 1981.

[9] Philippe Charles, Christian Grothoff, Vijay 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 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005.

[10] 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 '05, pages 21–28. ACM, 2005.

[11] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11. ACM, 2009.

[12] Marc Feeley. *An efficient and general implementation of futures on large scale shared-memory multiprocessors*. PhD thesis, Waltham, MA, USA, 1993. UMI Order No. GAX93-22348.

[13] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[14] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, PPOPP '90, pages 50–59. ACM, 1990.

[15] Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, 1985.

[16] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distrib. Comput.*, 18:189–207, February 2006.

[17] Danny Hendler and Nir Shavit. Work dealing. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 164–172. ACM, 2002.

[18] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[19] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[20] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based load balancing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 55–64, New York, NY, USA, 2009. ACM.

[21] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 111–120, Austin, TX, 2010. FMCAD Inc.

[22] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 45–54. ACM, 2009.

[23] Girija J. Narlikar and Guy E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*, 21, 1999.

[24] Mike Rainey. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. PhD thesis, University of Chicago, August 2010.

[25] Supercomputing Technologies. Cilk 5.1 (beta 1) reference manual. *The Open Group Research Institute*, 1997.