Atomic Read-Modify-Write Operations are Unnecessary for Shared-Memory Work Stealing

Umut A. Acar, Arthur Charguéraud, Stefan Muller, Mike Rainey

Atomic Read-Modify-Write Operations are Unnecessary for Shared-Memory Work Stealing

Umut A. Acar
Carnegie Mellon University & Inria
umut@cs.cmu.edu

Arthur Charguéraud
Inria & LRI, Université Paris Sud, CNRS
charguer@inria.fr

Stefan Muller
Carnegie Mellon University
smuller@cs.cmu.edu

Mike Rainey
Inria
mike.rainey@inria.fr

Abstract

We present a work-stealing algorithm for total-store memory architectures, such as Intel's X86, that does not rely on atomic read-modify-write instructions such as compare-and-swap. In our algorithm, processors communicate solely by reading from and writing (non-atomically) into weakly consistent memory. We also show that join resolution, an important problem in scheduling parallel programs, can also be solved without using atomic read-modify-write instructions.

At a high level, our work-stealing algorithm closely resembles traditional work-stealing algorithms, but certain details are more complex. Instead of relying on atomic read-modify-write operations, our algorithm uses a steal protocol that enables processors to perform load balancing by using only two memory cells per processor. The steal protocol permits data races but guarantees correctness by using a time-stamping technique. Proving the correctness of our algorithms is made challenging by weakly consistent shared-memory that permits processors to observe sequentially inconsistent views. We therefore carefully specify our algorithms and prove them correct by considering a costed refinement of the X86-TSO model, a precise characterization of total-store-order architectures.

We show that our algorithms are practical by implementing them as part of a C++ library and performing an experimental evaluation. Our results show that our work-stealing algorithm is competitive with the state-of-the-art implementations even on current architectures where atomic read-modify-write instructions are cheap. Our join resolution algorithm incurs a relatively small overhead compared to an efficient algorithm that uses atomic read-modify-write instructions.

1. Introduction

As parallel computing becomes mainstream with the advent of multi- and many-core computers, techniques for writing and executing parallel programs have become increasingly important. By allowing parallel programs to be written at a high level and in a style similar to sequential programs, implicit parallelism, as elegantly exemplified by languages such as Cilk [16], Fork/Join Java [27], NESL [4], parallel Haskell [24], parallel ML [15], TPL [28], and X10 [8], has emerged as a promising technique for parallel programming. Programs written in these implicitly parallel languages generate a plethora of fine-grained parallel threads, requiring an efficient and scalable scheduler for execution. In the course of the last decade, the work-stealing algorithm [6, 7, 16, 17] has proved to be an effective scheduling strategy.

Work stealing can be expressed simply and elegantly at a high level when assuming a sequentially consistent memory model and low-cost synchronization operations such as locks. Each processor maintains a pool of ready threads. When it creates a new parallel thread, the processor adds the thread to its pool. When in need of a thread, it takes one from its pool. If the pool is empty, then the processor steals work from another processor's pool. To guarantee efficiency under a sequentially consistent memory model [6], it suffices 1) to maintain pools as doubly-ended-queues, i.e., *deque*s, 2) to choose randomly the processor to steal from, and 3) to steal the "largest" (top) thread from a deque.

In practice, however, synchronization is expensive and can prevent scalability. Furthermore, parallel architectures support only weakly consistent memory models, where certain basic operations such as memory writes are not atomic and making them atomic by use of synchronization operations such as memory fences can be expensive. There has therefore been much research on designing concurrent data structures and algorithms for efficient work stealing on weakly consistent memory. Much of this work focused on

- reducing the use of locks and other *atomic read-modify-write operations* (e.g., compare-and-swap and fetch-and-add) to improve efficiency and scalability, and
- reducing *memory fences* (e.g., a store-load barrier) so that the benefits of weak memory architectures can be realized without unnecessarily serializing memory operations.

Based on Dijkstra's mutual exclusion protocol, Frigo et al [16] present a work-stealing algorithm that uses locks only when performing steals and only in an edge case of local deque operations. Other papers [2, 9] present non-blocking work-stealing algorithms that use atomic read-modify-write operations instead of locks. Such non-blocking algorithms have been shown to perform well on certain existing hardware such as current multicores. Michael et al [29] show that it can be beneficial to eliminate all memory fences and atomic instructions from local deque operations in total-store order memory models. Their approach, however, weakens the semantics of work stealing by allowing a thread to be removed from a deque (and thus executed) multiple times, which in some cases can also adversely affect correctness and efficiency. More recent work [1] presents a work-stealing algorithm that eliminates all memory fences and atomic instructions from local deque operations on total-store-order memory models without weakening the semantics of work stealing or adversely affecting its performance.

While much progress has been made in reducing the use of atomic read-modify-write instructions and memory fences, the question of whether they can all be eliminated remains open. We find this question to be theoretically interesting, because it hints at the limits of computing with processors that communicate non-atomically via weakly consistent memory. The problem also appears to be of practical interest. In current architectures, atomic instructions are relatively cheap, because they can take advantage of the cache-coherence protocol implemented in hardware. It is not known, however, whether they can scale to computers with larger numbers of processors. For example, citing such concerns, recent work [25, 32] proposes hardware support for work stealing that can improve performance significantly by eliminating certain atomic and synchronization operations. In addition, future parallel hardware may not support at all the cache coherence protocols on which fast implementations of atomic read-modify-write operations rely (e.g., [21]). For example, Intel's SCC (Single Chip Cloud computer) architecture provides for a shared memory architecture but not for cache coherence nor for atomic read-modify-write instructions. Such architectures would require algorithms that do not use atomic read-modify-write instructions.

In their work on work-dealing, Hendler and Shavit also describe techniques that can be used to eliminate all atomic read-modify-write operations [18]. The idea would be to establish pairwise communication channels between processors implemented as producer-consumer buffers without using atomic read-modify-write instructions. To send a message to another, a processor would write to their dedicated channel; to receive a message, it would scan through its channels. Unfortunately, for P processors, this approach would require $\Theta(P^2)$ communication channels in total, and $\Theta(P)$ worst-case time for receiving messages.[1]

In this paper, we present a work-stealing algorithm for weak memory architectures that uses no atomic read-modify-write operations and no memory fences on *total-store order*, *TSO* for short, architectures[2] (Section 4). To ensure efficiency, our algorithm uses only two memory cells per processor and permits data races to take place, recovering from them to guarantee correctness. In addition, we present a *join-resolution* algorithm for fork-join parallel programs for determining when threads become ready without using any atomic read-modify-write operations or memory fences.

We carefully specify our algorithms and prove their correctness and termination properties (Section 5). Since such proofs require reasoning precisely about the memory operations, we consider a specific TSO architecture, Intel's X86, as formalized by the *X86-TSO* memory model [33]. The X86-TSO model, however, is not sufficient to reason about termination and efficiency. Inspired by the work of Dwork et al. [12] on the partial synchrony model of distributed computing, we therefore present a costed extension of X86-TSO that bounds the time for a memory write to become visible to other processors. We denote this bound Δ. Our algorithms do not assume knowledge of Δ. We refer to the costed model as *X86-ATSO*, short for *algorithmic X86-TSO*.

[1] Using atomic read-modify-write operations, $\Theta(P)$ time for receiving a message can be reduced to $O(\log P)$.

[2] Architectures that support weaker memory models such as ARM multicore chipsets can often emulate TSO, possibly by using some memory fences, which tend to require local re-ordering of operations performed by one processor.

Our correctness proof based on the X86-ATSO model establishes the invariants that our scheduler must observe and shows that they remain true during execution, including as writes performed by a processor become non-deterministically visible to other processors. We don't prove our algorithm to be theoretically efficient—such a proof seems to be another major undertaking (without making strong assumptions). We do, however, briefly describe why we believe that our algorithms satisfy important invariants for efficiency (Section 6) and provide experimental evidence that our algorithms are practical (Section 7).

We show that our algorithms can be implemented without significant difficulty and provide an empirical evaluation by considering a set of benchmarks, including standard Cilk benchmarks as well as more recently proposed graph benchmarks from the PBBS [3] suite (Section 7). For comparison, we use a compare-and-swap-based private-deques algorithm, the traditional Chase-Lev algorithm with concurrent deques [9], and CilkPlus [22]. We expect our algorithms to be especially beneficial in future large-scale systems where atomic read-modify-write operations are expensive or unsupported in hardware. Our experiments show that, even on current architectures, our work-stealing algorithm is competitive with the state of the art and that our join-resolution algorithm has a small but noticeable overhead, suggesting that further work may be needed on that problem.

Due to space constraints, we have included proofs and more detailed experiments in a separately submitted appendix.

2. Background and Overview

We briefly present an overview of the main ideas in our algorithms in the context of prior work. We make the ideas precise by presenting detailed specifications of the algorithms in Section 4.

Throughout the paper, we assume fork-join parallelism and restrict ourselves to degree-two joins for simplicity. Since arbitrary-degree joins can be represented with a collection of degree-two joins, this assumption causes no loss of generality.

2.1 Work stealing

Centralized scheduling. The simple approach to scheduling is to keep a centralized pool of threads shared by all processors. When a processor creates a new thread, it places the thread into the shared pool. When a processor goes idle, it removes a thread from the pool to run. This algorithm requires the operations on the shared pool to be atomic, making the approach inefficient and unscalable.

Work stealing with atomic read-modify-write operations. For improved efficiency and scalability, work stealing employs multiple pools instead of one and performs load balancing via steals to keep processors busy. More specifically, each processor is assigned a pool implemented as a doubly ended queue (deque). When a processor creates a thread, it pushes the thread into the bottom end of its deque. When it needs a thread, it pops a thread from the bottom. If the deque is empty and thus there is no thread to be popped, the processor attempts to steal a thread from the top end of another process's deque. Progressive advances in the last two decades have reduced the need for atomic instructions dramatically [1, 2, 9, 16, 29]. Existing approaches, however, all rely on atomic insructions to support steals.

Work-stealing without atomic read-modify-write operations. Our starting point is a recent work-stealing algorithm [1] that eliminates all atomic read-modify-write operations and memory fences from local deque operations. The idea behind that algorithm is to use a *private-deques architecture* for work stealing, where each processor owns a non-concurrent, private deque (instead of a concurrent deque as is traditionally used) and relies on communication to perform load balancing via steals; such communication can be performed via well-understood polling techniques. The algorithm uses atomic read-modify-write operations to support steals.

To see how we can eliminate all atomic operations (without adding memory fences) by building on the private-deques architecture, suppose that we create pairwise communication channels, a total of $P(P-1)$, between each pair of processors. We can use each channel as a pairwise producer-consumer buffer and implement it without using atomic read-modify-write primitives and memory fences [18, 26]. This algorithm would be correct but it has three shortcomings: 1) it requires $P(P-1) \in \Theta(P^2)$ communication channels, 2) a processor needs $\Theta(P)$ time for receiving a message, and 3) it violates a key invariant of work stealing: since an idle processor can receive multiple threads, one thread can remain "stuck" in a channel for some time during which it cannot be stolen; this breaks a key efficency invariant of work stealing—that the topmost thread in a deque is stolen first.

To regain efficiency, we reduce the number of channels and limit them so that they can hold only one message at a time. More specifically, instead of using P pairwise communication channels, in our algorithm, each processor owns just two cells: one *query* cell for receiving incoming steal queries and one *transfer* cell for receiving the threads sent to it; each cell can hold only one query or transfer object. When a processor wishes to steal from another target processor, it sends a message to the target processor by non-atomically writing to its query cell and waits for a response. When a processor receives a steal query, it sends the thread at the top of its deque by writing to the transfer cell of the querying processor; if the processor has no threads then it denies the request again by writing to the transfer cell of the querying processor.

This approach leads to at least two issues: 1) since the query is made non-atomically, a query can be overwritten by another processor's query in a data race, and 2) since all communication takes place via non-atomic write operations and no memory fences can be used, writes can be delayed arbitrarily, causing inconsistenciens. To solve these problems, our algorithm keeps a logical clock at each processor, counting uninterrupted rounds of work. Each round starts when a processor gets work, or satisfies (by sending work) a steal query, or declines (because it has no work to share) the steal query. To solve the first problem—the problem of steal queries overwritten in a data race—we use the logical clocks, called *round numbers*, as follows: when an idle processor a sends a steal query to a target processor b, it tags its query with the round number of b. When a's query is overwritten by the query of another processor c, b cannot see a's request but will eventually respond to c (or another processor) and increment its round number. Processor a can detect that its steal attempt has failed by observing that the round number of b has increased. The second problem—the problem of delayed writes— can cause the query of an idle processor to be overwritten by another query, which was made at an earlier round. The idle processor detects such an inconsistent write by observing the round number of the query cell of the target, and rewriting its query as needed.

2.2 Join Resolution

For work-stealing to be efficient, it is important for a thread to be made available for execution as soon as it becomes ready, i.e., all its ancestor threads are executed. Threads with a single parent are ready when created but *join threads* with two parents become ready only when both parents complete execution. Since parents can be executed by different processors, determining when join threads become ready can involve coordination between processors.

Join resolution with atomic read-modify-write operations. It is relatively straightforward to give a simple algorithm for join resolution when using atomic read-modify-write instructions. For example, each join thread can keep a counter of unfinished parent threads. The counter initially contains the number of parents and is atomically decremented by the processor that executes a parent. The join thread is ready when the counter reaches zero. This algorithm can be optimized to use atomic instructions only for *non-local joins* that involve two processors [16]. With this optimization, experimental evidence suggests that join resolution works efficiently on some modern parallel systems, because non-local joins are rare (one per steal) and only two processors contend for the same memory object. This optimization has been used in Cilk [16] as well as our own comparison implementations. Another approach to join resolution would be to keep a two-processor barrier. This approach has the disadvantage of holding up the processor that first completes a parent, breaking a crucial greedy property of work stealing.

Join resolution without atomic read-modify-write operations. To develop some intuition for the problem, we start by assuming that every memory write becomes visible to all processors in some known time Δ and that we know the two processors that will execute the parents of the join thread. We create pairwise communication channels between all processors and send and receive messages through them without using atomic instructions [18, 26]. Now, when processor i finishes a parent thread, it sends a `READY` message to the other processor j working on the other parent, which we know by assumption, and waits for a response for 2Δ time units. If j receives this message before it completes the parent thread, then it will execute the join thread. If, however, j finishes before hearing from i, then it sends `READY` to i, causing both processors to receive `READY`, indicating a tie, which can be broken arbitrarily. If i has not received `READY` from j after 2Δ units of time, processor i continues to obtain other work.

We now describe how to eliminate three strong assumptions made by this algorithm: 1) synchronous communication, 2) knowledge of the processors executing the parents, 3) $\Theta(P^2)$ buffers. We avoid using Δ by requiring each processor to acknowledge the receipt of a `READY` message by sending an `ACK` message back. Thus, after sending `READY`, a processor will wait until it eventually gets back either `READY`, indicating a tie, or `ACK`, indicating success, and may proceed accordingly in either case.

For the second problem, note it is impossible to know the processors participating in a join *a priori*. Figure 1 illustrates an example, where processor i starts to execute thread 0, which creates thread 1 and thread 4, which is then stolen by processor j. At this time, our only knowledge for thread 6, the join thread of 1 and 4, is that it might be decided between the processors i and j. When 1 executes, however, it forks the threads 2 and 3, which is stolen by processor k. Thus, the processor that executes the join thread of 2 and 3, either i or k, will actually participate in the resolution of thread 6 but there is no way to know which—the outcome will be non-deterministic.

We overcome the second and the third problems by allowing processors to communicate through the join threads. To allow for this, the data structures representing join threads must carry two communication channels for processors running the parent threads. These processors are said to be *watching* this join. A processor may be watching many joins at a time and must periodically check its communication channels for messages. To this end, each processor maintains a *watch list* of joins to be watched. Processors add a join to their watch lists

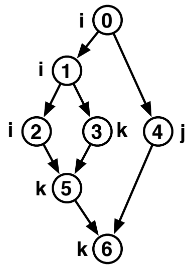

Figure 1. An example join resolution.

when a parent thread is stolen and remove it when it is resolved. Adding and removing joins can be done in constant time.

As the example in Figure 1 shows, watch lists must occasionally be transferred between processors when joins are resolved. For example, if the join on thread 5 is resolved such that processor k executes it, i must transfer its watch list to k so that k is aware of its responsibility to watch the join on thread 6. In general, the invariant that we maintain is that the branch of a join is watched by the leftmost processor (in tree order) that works on this branch. Thus, watch list transfers must occur when a left branch is completed and removed from the DAG, changing which processor is leftmost. It is a constant-time operation for a processor to append a transferred watch list to its own. However, since the transfer involves communication between processors, we avoid this operation when possible.

3. The Model: Algorithmic X86-TSO

For designing and reasoning about our algorithms, we consider the X86-TSO model [33] for *total-store-order* architectures (e.g., the X86 platform from Intel and AMD) and extend it with a cost model that assigns a cost to each operation. The cost model enables reasoning about termination and efficiency. We refer to this costed model as *algorithmic X86-TSO*, abbreviated as *X86-ATSO*.

Figure 2 illustrates the abstract X86-TSO machine model [33] for a 4-processor (core/CPU) specimen of the X86 archictecture with total-store order memory system and our cost model. The system consists of a shared memory and one first-in-first-out (FIFO) *store buffer* per processor, each mapping locations to values. When writing a location, the processor simply writes to its store buffer; the values stored in the buffer can propagate to memory any time. When reading a location, a processor first consults its store buffer and reads the value written by the most recent store into that location. If the location is not in the store buffer, then the processor accesses the memory.

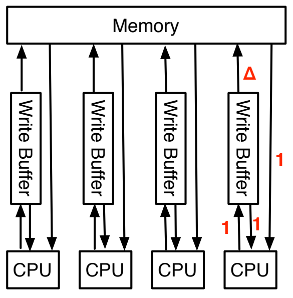

Figure 2. The X86-TSO model and the costs (in red).

Many X86 instructions operate locally without requiring synchronization with other processors but some, such as *memory fences* and LOCK'ed instructions, don't. *LOCK'ed* instructions include the atomic XCHG (compare-and-swap) instruction and the atomic versions of the many read-modify-write instruction (ADD, INC, etc) that, when preceded by the LOCK prefix, run atomically. A locked operation starts by taking a global read lock, which prevents all other processors from performing reads, and completes by flushing the store buffer and releasing the lock.

The X86-TSO model does not specify the cost of each operation, nor even termination. For example a memory-fence instruction may not return, or the values written into the store buffers may never propagate to memory. The lack of a cost specification may appear amiss, but it is in fact consistent with hardware because manufacturers do not specify whether the protocols implemented in hardware (e.g., cache coherency) are terminating and how frequently the buffers are flushed.

To design effective algorithms and to prove them correct, terminating, and efficient, a cost model is needed. Such a cost model should at least posit the termination of all operations including memory writes, because otherwise it would not be possible to reason about termination. Therefore, in *algorithmic X86-TSO*, memory reads and writes complete in a single time step, but writes can take up to an additional finite time, $\Delta \geq 1$, to become visible to all processors. Somewhat imprecisely, we illustrate the cost model in Figure 2 by assigning a cost of Δ for moving data from the store buffer to memory—Δ is an upper bound on the time for a write stored in a buffer to make it to memory.

While assuming that each operation takes at most Δ steps for some Δ may be realistic, we choose not to assume our algorithms are aware of the value of Δ for several reasons. First, since Δ can vary between different hardware and can be difficult to know, an algorithm designed for a particular Δ may not work well in all architectures. Second, the value of Δ might be large in the worst case but small in the common case, making an algorithm that uses Δ suboptimal. Therefore, our algorithms assume the existence of Δ but they don't use it. When reasoning about termination, we assume that some Δ exists, without explicitly quantifying it; in other words, we assume that each write eventually makes it to main memory.

4. The algorithms

Since we are primarily concerned with the design and the correctness of the algorithm on weak memory models, we present reasonably detailed and precise pseudo-code in C++-like syntax. To remain realistic, we do not assume that we are given a computation DAG to schedule but also construct the DAG dynamically du ring the execution. To express parallelism, we provide one primitive, `fork`, which can only be called at the end of a parallel thread, specifying the threads to be forked as well as their join. Given a work-stealing framework supporting some basic infrastructural data structures, these algorithms can be employed separately or together to perform load balancing with work stealing and perform join resolution. In fact, our implementation follows closely the pseudo-code and uses these algorithms paired with each other or other algorithms for experimental evaluation. We start by presenting the abstract data types and the data structures for work stealing along with the main scheduling loop and then describe the two algorithms.

4.1 The data structures

Several key data structures facilitate our algorithms. One key data structure is a doubly ended queue, a *deque* for short, for storing ready threads. Each processor owns a private deque that can be accessed only by that processor. Each processor operates on the deque following the work-stealing paradigm, by pushing new parallel threads at the bottom of the deque, and popping threads from the bottom when it completes the execution of a thread (with `push_bottom`, `pop_bottom` operations). Since deques are not concurrent, there is no need for memory fences, which would otherwise be needed at every `pop_bottom` operation.

Figure 3 specifies the other data types and structures used in our algorithm. The type `query` is used by the load-balancing algorithm for packing an identifier and a round number into a single machine word. A *query* consists of the identity of a processor and of a round number. For correctness, it is critical for processors to be able to make queries atomically. We therefore represent a query as a single 64-bit machine word (type `query`), using 24 bits for the identifier and 40 bits for the round number (which makes it possible to support computations up to 2^{40} rounds with up to 2^{24} processors). The function `q_make` constructs a query, and the functions `q_id` and `q_rnd` project out the ID and the round numbers of the offer, respectively.

The enumeration types `stat` and `side` are used by the join-resolution algorithm. The record `thread`, used to represent a thread, has three fields: the `status` field is an array of size two used by the join algorithm to keep track of the completion of the dependencies (if any) of this thread. These could be `ACK` indicating that the branch will complete last, `READY` indicating that the branch

```
// Query type with 40-bit round numbers.
type query = unsigned_int_64
query q_make(int i, int r)
  return r + (i << 40)
int q_id(query q)
  return (q >> 40)
int q_rnd(query q)
  return (q & ((1 << 40)-1))

// Constants for representation of threads.
type stat = | WORK
            | ACK
            | READY
            | TRANSFER { set<cont>* }
type side = int
const side LEFT = 0
const side RIGHT = 1
const side SINGLE = 2
```

```
// Representation of threads
type thread = {
  stat status[2]; // state of dependencies
  void run();     // body of the thread
  cont conti; }   // continuation

// Representation of continuations
type cont = {
  thread* join;      // end-point of the edge
  side branch; }  // position of the edge

// Global variables.
deque<thread*> deque[P] = {DEQUE_EMPTY, ..}
cont cur_cont[id]
set<cont> watching[P] = {SET_EMPTY, ..}
int round[P] = {1, ..}
query in_query[P] = {q_make(0,0), ..}
thread* received[P] = {NULL, ..}
```

Figure 3. Data types and global state.

has finished earlier than or concurrently with the other branch, WORK indicating that the branch has not been yet completed, and TRANSFER indicating the transfer of a watch list. These values, along with a pointer to the watch lists in the case of TRANSFER, can be packed into a single word and thus can be written atomically. The run method corresponds to the body of the thread, and the conti field describes the *continuation* of the thread. The continuation (type cont) consists of a pointer on a join thread, which has the current thread as a dependency, and a branch value, which indicates whether the current thread is the left, the right, or the single dependency of the join thread.

The shared state (global variables) include deques (called deque in the pseudo-code), current continuations (cur_cont), the watch lists (watching) defined as a set of items describing continuations that this processor must check for messages regarding join resolution. The other global variables (round, in_query and received) are used for work stealing. All these variables are processor-indexed arrays.

4.2 The main scheduling loop

Figure 4 shows the pseudo-code of the main scheduling loop, function main. Like all other functions of the algorithm, this function takes as first argument the ID of the processor calling it. If a processor finds its deque empty, then it calls acquire to obtain a thread. Otherwise, it pops the thread at the bottom of its deque and executes it, after remembering the continuation of this thread. After the thread completes, the processor handles the continuation using the function handle_cont, described below.

The fork function (Figure 4) takes as arguments the ID of the caller, the left branch thread, the right branch thread, and the join thread. When the currently-running thread calls fork, the current continuation is captured and used as a continuation of the join thread, while the current continuation is set to point to a NULL pointer. We set the status fields of the join thread to WORK in order to indicate that neither branch has completed execution. The left and right threads are then pushed onto the deque, after their continuation is set as the join thread and marked with the corresponding branch (LEFT or RIGHT). Note that the content of the status field of the branch threads is irrelevant because the threads are ready when created.

Figure 5 contains the code of the function handle_cont for handling continuations. If the continuation describes the end of the entire computation (END), the program exits. If the continuation carries a null pointer, indicating that the thread that has just run has forked, there is nothing to do. Otherwise, the continuation carries a valid thread pointer c.join and a branch value, which is one of SINGLE, LEFT or RIGHT. The value SINGLE captures the fact that the thread that has just run was the unique dependency on the join thread, so the join thread can be scheduled immediately. Otherwise, in the particular case where the thread that has just finished is a left branch and the corresponding right branch is still in the deque (it was not sent away), the remaining right branch can be safely converted into a single branch. In other cases, it must be that the left branch and the right branch are running on different processors, and therefore the join resolution algorithm needs to be invoked.

4.3 Our Work-Stealing Algorithm

To perform load balancing via steals without memory fences and atomic read-modify-write operations operations, our algorithm relies on explicit communication between processors and local operations on deques. To communicate, each processor periodically calls the function communicate. (Such periodic calls can be implemented using several known techniques (e.g., [1] and references thereof.) Since when exactly communication takes place does not affect correctness as long as it takes place within the main body of the scheduler (or by trapping into the scheduler from the user code), we do not discuss this any further.

For load balancing, each processor maintains a *round* number to differentiate between its own *phases*: starting from zero, the round number gets incremented every time the processor serves or declines a (steal) query. Each processor also uses two shared memory cells: an *in-query* cell (in_query) for receiving queries from idle processors, and a *reception* cell (received) for receiving thread pointers from busy processors.

The load-balancing algorithm (Figure 6) consists of two functions, called acquire and communicate. A processor calls acquire when its deque is empty. It starts by writing a null pointer in its reception cell, and then repeatedly attempts to acquire work from another randomly picked victim processor. To acquire work, it first checks that the victim admits queries by checking that the round number of the last query targeting the victim is less than victim's current round number. If so, then the processor writes a query containing its own identifier and the victim's round number into the victim's in-query field. The victim serves the query when it calls communicate. If a thread is delivered, the processor receives the thread, and updates its watch list to coordinate the resolution of the join with the victim. Otherwise the processor tries to steal again.

When a busy processor calls communicate (which it does periodically), it first reads its in_query field. If it finds a query whose round number matches its round number, then it means that (at least) one idle processor is waiting for an answer to its query. To serve the query, the processor checks if it has a thread that it can

```
void main(int i)
  while true do
    if deque[i].is_empty()
      acquire()
    thread* t = deque[i].pop_bottom()
    cur_cont[i] = t.conti
    t.run()
    handle_cont(i, cur_cont[i])
```

```
void fork(int i, thread* t1, thread* t2, thread* tj)
  tj.conti = cur_cont[i]
  cur_cont[i].join = NULL
  tj.status[LEFT] = WORK
  tj.status[RIGHT] = WORK
  t1.conti = cont { tj, LEFT }
  t2.conti = cont { tj, RIGHT }
  deque[i].push_bottom(t2)
  deque[i].push_bottom(t1)
```

Figure 4. Main loop and fork function.

```
// Called after completion of a thread
void handle_cont(int i, cont c)
  if c.join == END then exit(0)
  if c.join == NULL then return
  if c.branch == SINGLE
    deque[i].push_bottom(c.join)
  ...
  else if (c.branch == LEFT
           && not deque[i].empty())
    // Optimize the local right branch
    thread* t = deque[i].peek_bottom()
    t.conti.branch = SINGLE
  else
    resolve_join(i, c)
```

Figure 5. Handling of continuations.

```
// Called when a worker runs out of work
void acquire(int i)  // i = ID of caller
  received[i] = NULL
  while true
    int j = random ∈ {0, .., P-1}\{i}
    int r = round[j]
    // Test whether target accepts queries
    if q_rnd(in_query[j]) < r
      // Send a query
      in_query[j] = q_make(i, r)
      while round[j] == r
        // Resend same query if needed
        if q_rnd(in_query[j]) < r
          in_query[j] = q_make(i, r)
        block(i)
      // Receive the thread, if any
      thread* t = received[i]
      if t != NULL
        watching[i].push(t.conti)
        push_bottom(deque[i], t)
        round[i]++ // Accept queries
        return
    block(i)
```

```
// Called periodically by a busy processor
void communicate(int i) // i = ID of caller
  // Check for incoming queries
  query q = in_query[i]
  if q_rnd(q) != round[i]
    return
  // Process the incoming query
  if size(deque[i]) > 0
    && peek_top(deque[i]).conti.branch == RIGHT
    thread* t = pop_top(deque[i])
    cont conti = {t.conti.join, LEFT}
    watching[i].push(conti)
    received[q_id(q)] = t
  round[i]++ // Starts a new query phase

// Auxiliary function for blocking queries
void block(int i) // i = ID of caller
  int r = round[i]
  if in_query[i] != q_make(i,r)
    in_query[i] = q_make(i, r+1)
    round[i] = r+1
```

Figure 6. Load-balancing algorithm.

send. If so, it pops the thread from its deque, writes the corresponding thread pointer into the reception cell of the querying processor, and updates its watch list to coordinate with the thief. If not, the query cannot be served. In both cases, the processor increments its round number to implicitly notify all processors that made queries at the current round. A querying processor is thus able to detect whether its query was served or declined by testing whether its reception cell contains a non-null value.

The implementation of the function `acquire` involves two small complications. First, for efficiency, an idle processor should discourage other idle processors from making queries to it. To that end, an idle processor uses the auxiliary function `block` for making a query to itself (leaving it unanswered), thereby preventing other processors from making a query. Second, when an idle processor makes a query to a busy processor, it needs to ensure that, if its query gets overwritten by another query tagged with an out-of-date round number, then it resends its query. Without such a resend operation, the busy processor targeted would simply ignore the out-of-date query and never increment its round number.

4.4 Our Join-Resolution Algorithm

Figure 7 illustrates the pseudo-code for the join resolution algorithm. When a processor i completes executing a parent of a join thread, it calls the function `resolve_join` with the continuation for that join thread. The continuation consists of the join thread and an identification of the parent; `side = LEFT = 0` means the left parent and `side = RIGHT = 1` means the right parent. We start by checking the status field of the join for the parent that has just been completed. If the status field is `ACK`, then the other parent has completed and the join thread is ready for execution. We therefore push the thread into the bottom of the processor i's deque. If the status field is not `ACK`, then the parent has finished first or concurrently with the other parent. In this case, we set the status field of the join for this parent to `READY` and remove this continuation from the watch list of the processor. We then wait until we receive an `ACK` from the other processor watching the join or see that it is `READY`. If we receive an `ACK`, then this parent has finished first and thus the join has to wait for the other parent to complete. If this is the left branch, however, we first transfer the watch list of processor i to the other processor by making it available as part of the status field. If we receive a `READY`, then both parents completed at the same time and the processor that executed the left branch pushes the join thread into its deque (breaking the tie in this way avoids transfering the watch list.)

The function `watch`, which is periodically called by each processor, acknowledges the receipt of `READY` messages from other parents of the join that the processor i is watching, and transfers

```
// Called to resolve a non-local join
void resolve_join(int i, cont c)
  int b1 = c.branch
  int b2 = 1 - b1
  int* status = c.join.status
  if status[b1] == ACK // Finish second.
    deque[i].push_bottom(c.a.join)
  else // May have finished first
    status[b1] = READY
    watching[i].remove(c)
    while status[b2] == WORK
      noop // May call watch here
    match status[b2] with
    | ACK:
      if b1 == LEFT // Transfer watch list
        set<cont>* s = watching[i].copy()
        status[b1] = TRANSFER{s}
        watching[i].remove_all()
    | READY:
      if b1 == LEFT // Resolve race to left
        deque[i].push_bottom(c.join)
```

```
// Called periodically by a busy processor
void watch(int i)
  foreach (cont c) in watching[i]
    int b1 = c.branch
    int b2 = 1 - b1
    int* status = c.join.status
    if status[b2] == READY
      // Other branch done
      status[b1] = ACK
      watching[i].remove(c)
      if b1 == LEFT
        continue
      // Reception of the watch list
      while status[b2] == READY
        noop // May call watch
      match status[b2] with
      | TRANSFER{s}:
        watching[i].add_multiple(s)
```

Figure 7. Join resolution algorithm.

watch lists when necessary. To this end, it checks, for each join, whether the other parent is `READY`. If so, it acknowledges by writing `ACK` to the status field of the join, indicating that this parent will be responsible for the join and stops watching the join. If this parent in the right branch, it transfers the watch list from the left parent.

5. Correctness

To prove the correctness of our algorithms, we must first define what correctness means. We do this by building a formal, dynamic notion of a valid scheduler and then proving a correspondence between this formal notion and our concrete algorithm. Full details of the (12-page) proof appear in the appendix.

Assumptions. For the proof, we assume the X86-ATSO model described in Section 3. In particular, we assume that there exists a fixed (but unknown) upper bound Δ on the time in which writes are flushed to memory. We also make a (relatively weak) fairness assumption concerning the scheduling of our threads by the operating system: if a concrete execution does not terminate, then every processor takes infinitely many steps.

Proof Outline. We formalize a valid scheduler as an *abstract semantics.* The abstract semantics that we consider is presented as a reduction relation on the *abstract state*, which is an idealized view of the parallel computation to be performed. More precisely, the abstract state consists of a quadruple of the form (N, E, X, M). The pair (N, E) corresponds to the *computation DAG*, in which threads are represented as nodes and dependencies between threads are represented as edges. The map X maps each processor to the thread that it runs, if any. (Technically, X also includes the not-yet-run code of the running threads.) The variable M describes the *application state*. M is a concrete representation of memory that follows the X86-ATSO model, but contains only the application memory and not the data structures used by our scheduler.

The abstract semantics consists of transitions between abstract states, written $A \longrightarrow A'$. There are four possible transitions from a state (N, E, X, M). First, an idle processor (i.e. a processor bound by X to no thread) may be associated a ready thread (i.e., a node from N that has no incoming edge according to E). Second, a processor may execute one atomic instruction from its assigned thread; as a result, the application state M may be updated. The remaining two transitions describe the case of a processor that completes a thread and then becomes idle, after removing from N the corresponding node and removing from E all associated edges. Which of the two rules applies depends on whether the last operation performed by the thread is a fork operation. Formal rules for these transitions are given in the appendix.

The initial abstract state, called A_0, is defined as $(\{n_0\}, \emptyset, \emptyset, \emptyset)$, where n_0 is the initial thread. This thread describes the entire parallel computation in the sense that it will fork other threads, which themselves fork other threads which eventually build the entire computation DAG. When it terminates, an abstract execution reaches a state of the form $(\emptyset, \emptyset, \emptyset, M)$, where M, the final application state, typically contains the final result of the computation. We use the abstract semantics as a model for all valid schedulers.

The *concrete state* corresponds to the state of our program code (including scheduling code) at a given point in time. A concrete state C includes the code pointer of each of the processors and the content of all the shared memory. C includes the application memory, which is identical to the M component of the abstract state, as well as the data structures used by the scheduler such as the set of allocated thread objects, the state of the deques, etc. We project out the application memory component of C by writing $\mathrm{Mem}(C)$. The initial concrete state, written $C_0(n_0)$, describes a state where all deques are empty, except that of one processor, which holds the initial thread corresponding to the initial node n_0. Transitions on the concrete state, written $C \Longrightarrow C'$, correspond to execution of one atomic instruction by one of the processor or one flush from a write buffer into shared memory.

Our theorem, stated below, has two parts. The first half (correctness) asserts that if there is a reduction sequence for the concrete execution that terminates on a memory state C, then there exists a reduction sequence for the abstract state that reaches a state with application memory $\mathrm{Mem}(C)$. The second part (liveness) asserts that if the concrete execution may diverge, there there exists a reduction sequence for the abstract state that diverges as well.

Theorem 5.1 (Correctness and liveness)**.** *For any initial thread node n_0, and for any concrete state C in which all processors have run out of threads and are running* `acquire`*,*

$$C_0(n_0) \Longrightarrow^* C \quad \textit{implies} \quad (\{n_0\}, \emptyset, \emptyset, \emptyset) \longrightarrow^* (\emptyset, \emptyset, \emptyset, \mathrm{Mem}(C))$$
$$C_0(n_0) \Longrightarrow^\infty \quad \textit{implies} \quad (\{n_0\}, \emptyset, \emptyset, \emptyset) \longrightarrow^\infty$$

Note that, in the particular case of a *deterministic* terminating parallel program, our theorem implies that if the program reaches some application state in the abstract semantics, then it also reaches this application state in the concrete semantics.

The proof of the correctness result is based on a simulation lemma. This lemma shows that we are able to exhibit a relation R between abstract and concrete states, such that any action taken by the algorithm corresponds (in a way defined by the relation R) to an abstract transition, that is, to an allowed action of a scheduler.

Lemma 5.2 (One-step simulation)**.** *There exists a relation R for which the following implication holds: if $C \implies C'$ and $R\,C\,A$, then there exists A' such that $A \longrightarrow^* A'$ and $R\,C'\,A'$.*

The definition of R includes relations between the data structures of the algorithm code and the components of the abstract state, as well as all the invariants of the concrete state that are required to show that our algorithms are correct. These invariants are stated in terms of the main memory and the state of the write buffers associated with the processors.

To make this proof tractable and more modular, we separate the write buffers into one buffer per core and per memory location. Since there are situations where the order between the writes is crucial, we associate with each write a (logical) time-stamp. These time-stamps, local to each processor, allow us to express in our invariants comparisons between the times of writes by the same processor to two different locations. In addition to Lemma 5.2, we show that the flush of any write from a buffer to main memory preserves all the invariants.

6. Efficiency

At a high level, the behavior of our load balancing algorithm is very close to the compare-and-swap-based, receiver-initiated work stealing algorithm with private deques [1], which has been proved efficient. We believe that, under the assumption that the time taken by write operations to reach the main memory (the time bounded by Δ) follows a Poisson distribution, it would be possible to extend that proof to prove a similar result for our work-stealing algorithm. We find this assumption rather unrealistic, however. Without the assumption, it seems that a different proof technique might be needed. Setting up such proof is beyond the scope of this paper.

In this section, we discuss rather informally (mostly without proof) the key aspects regarding the efficiency of our algorithms. Following the presentation of the proof of the CAS-based algorithm [1], we let τ denote the average delay between calls to the functions `communicate` and `watch`. As established by the proof, for parallel computations exhibiting a reasonable amount of parallelism, τ can be set to a large value without adversely affecting performance. In particular, for most practical purposes, we can assume τ to be at least an order of magnitude greater than Δ, which is a rather small value.

Efficiency of the work stealing algorithm. A successful steal operation involves sending a query, which takes no more than Δ, waiting for the targeted processor to call `communicate`, which takes τ on average, and receiving the answer, which takes no more than Δ. Overall, instead of the τ bound on the successful steal operations in the CAS-based algorithm [1], we now have a bound of $\tau + 2\Delta$. This overhead, charged to the steals, is relatively small. Successful steals always steal the top task in the target's deque and the stolen task starts executing immediately afterwards.

Unsuccessful steals are slightly more delicate to analyze, because of the stale queries, which may appear to slow down the load balancing process. We argue that stale queries in fact do not harm performance. When a processor increments its round number, this number is seen by all other processors after a delay of Δ. Therefore, after a 2Δ delay, a processor cannot receive stale queries, unless it has incremented its round number in the meantime. In other words, unless the processor goes idle or serves a valid query, after a period of 2Δ, we know that this processor can only receive valid queries. Stale queries therefore can cause a small delay to successful steals. When $\tau \gg \Delta$, stale queries are even less likely. (Indeed, in our experiments, we rarely observed stale queries; the few that were observed could be due to swapping of workers by the operating system.)

Efficiency of the join-resolution algorithm. Our join resolution algorithm operates on watch lists in constant time, except for the watchlist traversal operation. We therefore show that items contained in all watchlists, summed over all P processors, does not exceed $P - 1$. This result suffices to show that the relative, amortized overhead associated with periodic checking of watchlists does not exceed $O(1/\tau)$ per processor. The proof of this result, detailed in Appendix C, relies on the intuition that the number of open interprocessor joins does not exceed $P - 1$. Intuitively, since the processors are always working on disjoint sub-DAGs of the computation DAG, the paths in the DAG from these sub-DAGs down to the final join form a reverse tree structure. This tree has P leaves (the sub-DAGs), so it cannot have more than $P - 1$ nodes of arity 2 (the open join nodes).

7. Experiments

Because atomic read-modify-write operations are quite efficient on current multicore computers, especially when they are rare as in the state-of-the art implementations of work stealing, we do not expect our algorithms to outperform state-of-the-art implementations on today's multicore computers. Our work is primarily motivated by the theoretical question of whether atomic read-modify-write operations may be eliminated completely and with the possibility that such operations may be absent from future hardware or may not be efficient. Nevertheless, we feel that an experimental evaluation on current hardware would be valuable to assess the implementabality and practicality of the algorithms. We report some of our findings here; the reader can find more details in the submitted appendix.

Implementation and experimental setup. We implemented our algorithms in the context of a C++ library for multithreading and evaluated them with a relatively broad range of benchmarks. Our implementation creates one POSIX thread (i.e., *pthread*) for each core available, and allows for dynamically selecting a scheduler of our choice, leaving the rest of the code the same. As basis for comparison, we used two other schedulers: first, a publicly available implementation of the CAS-based private-deques algorithm [1] and, second, our own implementation of the Chase-Lev [9] algorithm. In addition, we compare our work to Cilk Plus, which is an extension of GCC that implements an algorithm that is similar to Chase-Lev. Cilk Plus benefits from many years of careful engineering and sets a high standard for the performance of our algorithm. The private-deques algorithm, on which our algorithm is based, helps us isolate the cost of our steal protocol and compare it to the cost of performing steals with CAS instructions. Our own implementation of the Chase-Lev algorithm, even though it is not as optimized as Cilk Plus's implementation, helps us isolate the effect of the difference between our benchmarks and Cilk bencmarks, when needed.

For the parallel executions, all schedulers produce exactly the same computation DAG for all benchmarks, with the exception of the *sort* benchmark, for which Cilk Plus is using a different granularity control technique for the parallel loops (creating $8P$ subtasks). For our measurements, we used a 32-core, 2.0GHz Intel machine with 1Tb of RAM (our benchmarks use a fraction of the available memory). We consider just 30 out of the 32 total cores in order to reduce interference with the operating system. The baseline performance is, for each benchmark, from the single-core run time of the purely-sequential version of the code. To tame the variance observed in the measures when running with 30 cores (usually 5% to 10% noise), we averaged the measures over 20 runs.

	PD/CAS (speedup)	PD/CAS (sec)	Our WS (%)	Our WS+JR (%)
matmul	26.1	2.1	-0.0	+2.2
cilksort (exptintseq)	19.1	1.3	-0.1	+14.3
cilksort (randintseq)	22.2	1.5	-0.2	+8.1
fib	27.5	3.7	-1.9	+0.9
matching (eggrid2d)	19.4	1.6	+5.4	+13.6
matching (egrlg)	19.4	4.1	+0.7	+7.7
matching (egrmat)	18.9	3.6	-0.7	+7.2
MIS (grid2d)	15.7	0.8	-11.2	+9.8
MIS (rlg)	18.8	1.9	-0.9	+13.4
MIS (rmat)	19.0	2.1	-1.6	+12.6
hull (plummer2d)	17.3	2.3	-3.5	-2.4
hull (uniform2d)	18.3	4.4	-0.4	+1.5
sort (exptseq)	23.3	1.6	-0.9	+0.7
sort (randdblseq)	24.4	2.2	-1.3	+1.3

Table 1. Comparison with private-deques algorithm with CAS's and fetch-and-add (smaller % is better).

	Cilk Plus (speedup)	Cilk Plus (sec)	Chase Lev (%)	Our WS (%)	Our WS+JR (%)
matmul	23.3	2.4	+7.0	-10.8	-8.8
cilksort (exptintseq)	20.1	1.2	+7.1	+5.4	+20.5
cilksort (randintseq)	23.0	1.4	+4.6	+3.5	+12.1
fib	25.7	4.0	-5.6	-8.6	-5.9
matching (eggrid2d)	17.6	1.7	+7.2	-4.1	+3.4
matching (egrlg)	19.7	4.0	+9.5	+2.4	+9.4
matching (egrmat)	18.5	3.6	-1.8	-2.8	+5.0
MIS (grid2d)	16.8	0.7	+13.2	-5.2	+17.3
MIS (rlg)	18.2	2.0	+3.0	-3.9	+9.9
MIS (rmat)	17.3	2.3	+6.1	-10.4	+2.4
hull (plummer2d)	19.8	2.0	+8.0	+10.6	+11.8
hull (uniform2d)	18.9	4.2	+3.6	+2.8	+4.8
sort (exptseq)	15.0	2.5	-38.5	-36.0	-35.0
sort (randdblseq)	15.0	3.5	-40.1	-39.2	-37.7

Table 2. Comparison with Cilk Plus and Chase Lev (smaller % is better).

Benchmarks. As benchmark, we consider three classic programs ported from Cilk and a number of programs from Blelloch et al.'s recent problem-based benchmark suite (PBBS) [3]. Cilk benchmarks include *cilksort*, which is based on a parallel version of merge-sort; *matmul*, which multiplies two dense matrices in place using a cache-efficient, divide-and-conquer algorithm [16]; and *fibonacci*, which computes Fibonacci number using the exponential algorithm. (This last benchmark is useful to perform analyses without observing interference from the memory.) PBBS benchmarks include internally-deterministic parallel programs targeting Cilk: *matching*, which computes the maximal matching of an undirected graph; *MIS*, which computes the maximal independent set of an undirected graph; *hull*, which computes a 2-dimensional convex hull; and *sample-sort*, which is a low-depth, cache-efficient version of the classic sample sort algorithm. Note that two of the benchmark programs, namely *matching* and *MIS*, make use of atomic CAS instructions [3], but the others do not.

Terminology. We use the following terminology: ``Our WS'' refers to our work-stealing algorithm that uses no atomic read-modify-write instructions for stealing work but uses fetch-and-add operations for join resolution; ``Our JR'' refers to our join-resolution algorithm with no atomic read-modify-write operations. We use our join-resolution algorithm only with our work-stealing algorithm. ``Our WS+JR'' refers to this algorithm. ``PD/CAS'' refers to the private-deques algorithm with CAS-based steals. ``Chase Lev'' refers to our implementation of the Chase-Lev algorithm. ``Cilk Plus'' refers to Cilk Plus. All these approaches (``PD/CAS'', ``Chase Lev'', and ``Cilk Plus'' use the optimized fetch-and-add based join resolution (Section 2).

Work Stealing. To understand the overheads of our work-stealing algorithm (described in Section 4.3), we can compare its performance to the private-deques algorithm with CAS's, which differs from our algorithm only in how the steals are performed. Table 1 reports on the absolute speedups of these algorithms and on the the relative differences between the three. Our algorithm is consistently the same or slightly faster, except for one case where it is 5% slower. Our algorithm is on average 1.2% faster. These measurements show that even on current architectures, where CAS operations are cheap, there is no measurable overhead to eliminating them from work stealing.

Table 2 gives the speedup and absolute run time of Cilk Plus and gives the relative value of the run time of the other algorithms: the Chase-Lev algorithm, our work-stealing algorithm, and our work-stealing algorithm with our join-resolution algorithm. Compared with Cilk Plus, our work-stealing algorithm (the second column from the last) is in some case faster and in some cases slower, but never more than 10% slower. Compared to Chase Lev, our algorithm is almost always faster—on average, 6.9% and 5.6% faster compared to Cilk and Chase Lev respectively. For sorting benchmarks *sort*, the run time of Cilk Plus is quite poor. As measurements with Chase-Lev confirms, this is due to the different treatment in Cilk Plus of the granularity in parallel loops. These measurements indicate that our work-stealing algorithm is competitive with the state of the art CAS-based work-stealing algorithms.

Our join-resolution algorithm. As can be seen in Tables 1 and 2, by comparing the last two columns, our atomic-read-modify-write-free join resolution algorithm is slower, by at most 15% but usually less than 5%, than the optimized fetch-and-add algorithm for join resolution employed by the other schedulers. To understand why it is slower, we examined in more detail the statistics of the worst case, namely cilksort(exptintseq). From the statistics, we were able to rule out factors such as higher latency or larger total-parallel-work. Few potential factors remain. One is the overhead of maintaining watch lists. These findings suggest that 1) join resolution with atomic read-modify-write operations performs well on current architectures, and 2) that join resolution without atomic read-modify-write instructions can be noticeably slower, which might become an important issue in future, larger-scale architectures.

8. Related work

While there is a lot of work on scheduling for parallel computations on hardware shared memory machines, all of the previous work relies on synchronization operations such as locks, memory fences, and atomic read-modify-write operations [2, 5, 9, 16, 31]. Some research has looked into eliminating memory fences, sometimes by weakening the correctness guarantees of work stealing [29], some in the general case [14, 15, 18, 20, 32, 34]. Recent work shows that memory fences can be eliminated from work stealing without detrimentally effecting efficiency [1].

The cost of synchronization in concurrent algorithms has been an important subject of study. Non-blocking algorithms and data structures [19] disallow the use of locks but rely instead on atomic read-modify-write operations such as compare-and-swap. The algorithms that we propose appear to be non-blocking due to the absence of any locks and other synchronization operations but they are not wait-free because an idle worker can wait for another busy worker indefinitely if that worker is suspended by the operating system. This problem, however, appears not too difficult to remedy.

To eliminate memory fences and atomic read-modify-write operations, our algorithms use explicit communication between processors. Our algorithms therefore have some similarities to distributed algorithms, which communicate by message passing. There has been much work on distributed scheduling algorithms considering both the receiver- and sender-initiated approaches [1, 10, 11, 13, 30]. In the sender-initiated approach, busy processors periodically share work with other idle (low-load) processors; in the receiver-initiated approach idle (low-load) processors demand work from busy (high-load) processors. Both approaches perform quite well though certain specifics regarding delays and system load can make one preferable over the other. Our algorithms differ from distributed algorithms because they use shared memory for storing the shared state (e.g., threads) between processors. In addition, for efficient steals, our work-stealing algorithm allows certain data races; for efficient join resolutions, it carefully creates and manages message channels.

9. Conclusion

This paper shows that it is possible to perform work-stealing and join-resolution without atomic read-modify-write operations and memory fences. In our algorithms, as in distributed algorithms, processors communicate explicitly. Unlike in distributed algorithms, our algorithms take advantage of shared memory by, for example, sharing state and permitting and taking advantage of data races. Our algorithms seem practically efficient, and we informally reason, but do not prove, that they are efficient in theory. Such proofs, left to future work, seem non-trivial especially because of the partial-synchrony-like assumptions required to account for the effects of the weak memory model.

References

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2013.

[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] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 181–192, New York, NY, USA, 2012. ACM.

[4] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*, pages 213–225. ACM, 1996.

[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 *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 207–216, 1995.

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

[7] 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.

[8] 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.

[9] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, 2005.

[10] Sivarama P. Dandamudi. The effect of scheduling discipline on dynamic load sharing in heterogeneous distributed systems. *Modeling, Analysis, and Simulation of Computer Systems, International Symposium on*, 0:17, 1997.

[11] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –8, march 2007.

[12] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35:288–323, April 1988.

[13] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Perform. Eval.*, 6(1):53–68, 1986.

[14] Marc Feeley. A message passing implementation of lazy task creation. In *Parallel Symbolic Computing*, pages 94–107, 1992.

[15] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011.

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

[17] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 9–17. ACM, 1984.

[18] 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.

[19] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[20] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based load balancing. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 55–64. ACM, 2009.

[21] Jason Howard, Saurabh Dighe, Sriram R. Vangal, Gregory Ruhl, Nitin Borkar, Shailendra Jain, Vasantha Erraguntla, Michael Konow, Michael Riepen, Matthias Gries, Guido Droege, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek K. De, and Rob F. Van der Wijngaart. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. *J. Solid-State Circuits*, (1):173–183.

[22] Intel. Cilk Plus. `http://software.intel.com/en-us/articles/intel-cilk-plus/`.

[23] Intel. Intel Xeon Processor X7550. Specifications at `http://ark.intel.com/products/46498/Intel-Xeon-Processor-X7550-(18M-Cache-2_00-GHz-6_40-GTs-Intel-QPI)`.

[24] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, 2010.

[25] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. *SIGARCH Computer Architecture News*, 35:162–173, June 2007.

[26] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, April 1983.

[27] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000.

[28] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, 2009.

[29] 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)*, pages 45–54, 2009.

[30] R. Mirchandaney, D. Towsley, and J.A. Stankovic. Analysis of the effects of delays on load sharing. *Computers, IEEE Transactions on*, 38(11):1513 –1525, nov 1989.

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

[32] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 311–322, New York, NY, USA, 2010. ACM.

[33] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53:89–97, July 2010.

[34] Seiji Umatani, Masahiro Yasugi, Tsuneyasu Komiya, and Taiichi Yuasa. Pursuing laziness for efficient implementation of modern multithreaded languages. In *ISHPC*, pages 174–188, 2003.

A. Experiments

Detailed description of test machine. Our test machine hosts four eight-core Intel Xeon X7550 [23] chips with each core running at 2.0GHz. Each core has 32Kb each of L1 instruction and data cache and 256 Kb of L2 cache. Each chip has an 18Mb L3 cache that is shared by all eight cores. The system has 1Tb of RAM and runs Debian Linux (kernel version 3.2.21.1.amd64-smp). We consider just 30 out of the 32 total cores in order to reduce interference with the operating system. All of our code is compiled by the CilkPlus GCC (v4.8.0 20120625) with the `-O2` option.

Thread granularity. For simplicity, we make the assumption that the granularity of threads is controlled appropriately, so that it suffices to check between every two threads whether the time has come to call the communication functions. (Otherwise, local interrupts can be used, as described in [1].)

Input data. The input sizes of the benchmarks are as follows: *cilksort*: random and exponentially-distributed, 240m integers, *matmul*: square matrix of size 3500, *fibonacci*: n = 50, *matching* and *MIS*: 3-d grid with 140m nodes, random graph with 500m nodes and 5b edges, and rMat graph with 500m nodes and 5b edges, *hull*: uniform and plummer with 800m points, *sample-sort*: random and exponentially-distributed, 240m doubles.

In-depth analysis of our results. By comparing with the performance of the Chase-Lev algorithm, and by investigating detailed statistics on the runs, we can identify several causes of the difference in performance of our algorithm with respect to Chase-Lev and Cilk Plus. Tables 8 and 9, and Table 10 show the three key statistics: number of steals, total parallel work, and relative idle time. Total parallel work is measured by summing the running times of all the sequentialized leaves of the call tree. The statistics show that the two algorithms which use concurrent deques achieve a slightly better utilization of the cores by performing quicker steals. However, better utilization does not always pay off because in a few cases there is a corresponding increase in the number of steals, cache misses, and in a few cases CAS conflicts. We see the impact of the cache misses and CAS conflicts by examining the total parallel work measures, which show that, for example for matching(eggrid2d), poorer utilization correlates with less total parallel work. In this case, better utilization of cores effectively slows down the application by creating a bottleneck on the memory. Another cause of their relative poor performance is that the concurrent deques algorithms have to pay for the cost of the fence, which we have measured, when running with a single processor, to represent almost 5% of overhead.

	`Our WS` (nb. steals)	`Our WS + Our JP` (%)	`Chase Lev` (%)	`PD/CAS` (%)
matmul	2199	+0.9	+1.8	-3.1
cilksort (exptintseq)	3169	-13.0	+2.6	-1.3
cilksort (randintseq)	3127	-9.4	+4.8	-7.4
fib	833	-14.1	+7.8	-7.3
matching (eggrid2d)	63449	-25.7	+15.4	+2.9
matching (egrlg)	74326	-21.2	+13.4	+2.1
matching (egrmat)	74118	-21.5	+13.3	+2.5
MIS (grid2d)	31328	-29.5	+13.9	+0.4
MIS (rlg)	40223	-27.0	+13.6	+3.1
MIS (rmat)	40840	-26.3	+12.5	+1.7
hull (plummer2d)	8780	-21.4	+8.9	+2.9
hull (uniform2d)	10915	-16.0	+5.8	-0.7
sort (exptseq)	3852	-14.1	+10.5	+3.3
sort (randdblseq)	3953	-13.4	+9.1	-1.8

Figure 8. Number of steals.

	`Our WS` (total seq.)	`Our WS + Our JP` (%)	`Chase Lev` (%)	`PD/CAS` (%)
matmul	64.3	-2.7	+17.4	-1.9
cilksort (exptintseq)	36.1	-0.4	-0.5	-0.9
cilksort (randintseq)	41.5	+0.6	-0.8	-0.2
fib	107.0	-0.0	-1.0	+0.9
matching (eggrid2d)	42.0	-2.6	+9.8	+2.4
matching (egrlg)	108.0	-0.0	+25.7	-0.7
matching (egrmat)	92.8	-0.7	+11.6	+7.4
MIS (grid2d)	16.0	-2.0	+13.4	+0.2
MIS (rlg)	50.5	-7.8	-1.7	-9.7
MIS (rmat)	53.0	-2.2	+11.9	+5.7
hull (plummer2d)	28.5	-0.4	-2.4	+0.9
hull (uniform2d)	65.7	-2.1	-0.8	-1.8
sort (exptseq)	22.9	-0.5	+0.0	-0.6
sort (randdblseq)	37.0	+0.5	-0.4	-0.3

Figure 9. Total parallel work (i.e., total time spent in processing the sequentialized leaves of the computation tree; smaller % is better).

	`Our WS` (rel. idle %)	`Our WS + Our JP` (%)	`Chase Lev` (%)	`PD/CAS` (%)
matmul	0	+12.6	-77.6	-8.9
cilksort (exptintseq)	1	-32.1	-89.2	-32.1
cilksort (randintseq)	0	+34.2	-81.8	-14.0
fib	0	+73.7	-67.3	+79.0
matching (eggrid2d)	5	+82.0	-73.8	-3.3
matching (egrlg)	3	+81.4	-80.4	+5.6
matching (egrmat)	3	+93.8	-77.3	+2.6
MIS (grid2d)	5	+104.2	-71.2	-4.4
MIS (rlg)	2	+121.4	-76.5	+2.1
MIS (rmat)	2	+104.4	-78.7	-4.2
hull (plummer2d)	6	+4.1	-14.2	+25.7
hull (uniform2d)	1	+29.4	-28.0	-14.9
sort (exptseq)	2	+3.1	-63.0	+37.5
sort (randdblseq)	1	+1.7	-66.6	+47.7

Figure 10. Relative idle time (smaller % is better).

B. Correctness proof

B.1 Overview of the proof

The structure of the proof is as follows. First, we describe the abstract semantics, which gives a high-level description, in terms of nodes and edges, of valid scheduling behavior. Second, we describe the relation between the concrete semantics, which corresponds to the pseudo-code given in the paper, and the abstract semantics. This relation captures in particular all the invariants of work stealing executions. The invariants on the concrete state include constraints on the private and shared variables, including contents of the buffers for shared variables. Then, we prove that the load balancing algorithm and join resolution algorithm are correct in that they preserve the relation between the concrete state and an appropriate abstract state. This shows that the concrete semantics never displays behavior that is invalid according to our abstract definition. We also argue for the liveness of our entire scheduling algorithm. Putting all the pieces together, we are able to conclude that Theorem 5.1 is correct.

In order to show Lemma 5.2, we must show that the invariants we define hold under any transition of the concrete state. These transitions are:

1. One processor executes any atomic instruction from the scheduler or program code.
2. The last item in one processor's write buffer is *flushed* to main memory. An invariant that holds under this kind of transition is called *stable*.

In order to prove that the invariants are stable under flush operations, we present a technique to reason modularly about the x86-TSO shared memory. We decompose the store buffer of each processor as a collection of store buffers, one per memory cell. When needed (and only when needed) we state constraints relating the times at which a same processor has pushed two different writes in its store buffer.

Notation. For a memory location x, we write X the value of this location in the shared memory, and we write $\bar{X}^i$ the list of values that correspond to the write requests stored in the buffer of processor i for the location x. We follow the convention that the more recent request is at the head of the list. Moreover, for convenience, we introduce two other pieces of notation related to shared memory locations. We write $\vec{X}^i$ for the concatenation of the list $\bar{X}^i$ with the value X. We write X^i the value at the head of the list $\vec{X}^i$. This value corresponds exactly to the value value returned when processor i reads at location x. Finally, we introduce notation to refer to local (non-shared) variables in the pseudocode, when the processor whose stack contains the variable is clear from context. The notation x_n refers to the value of variable x at line n in the pseudocode.

Outline. We first present formal definitions of abstract and concrete states. Abstract states and their transitions are defined in Appendix B.2. The defintion of abstract states is extended with additional information and invariants relating to work stealing in Appendix B.3. The relation R between abstract and concrete states is defined in Appendix B.4. Lemma 5.2, which states that all concrete transitions preserve this relation, is then proved in stages. Appendix B.5 proves the stability of invariants related to the load balancing algorithm, and proves that all invariants hold under atomic transitions in the load balancing algorithm. Appendix B.6 gives similar proofs for the join resolution algorithm. Appendix B.7 puts the pieces together to prove Lemma 5.2. Liveness is proven in Appendix B.8, which sets up the proof of Theorem 5.1 in Appendix B.9

B.2 Abstract semantics

We first describe an abstract semantics which describes abstractly what we consider to be valid behavior of any scheduler (not necessarily a work-stealing scheduler.) An abstract state gives the state of the abstract scheduler at any point in time. The abstract semantics describes how valid abstract states transition to one another.

Definition 1 (Representation of the abstract state)**.** An abstract state A is represented as a tuple (N, E, X, M) and consists of:

– N: a set of nodes, representing threads to be scheduled.

– E: a set of edges (n_1, n_2), where $n_1, n_2 \in N$. If $(n_1, n_2) \in E$, then the thread corresponding to n_2 depends on n_1 and cannot be scheduled until n_1 completes.

– X: a map that binds a processor id either to $\mathsf{NO\_CODE}$ or to a pair (n, k) where $n \in N$ and k describes the remaining computation to be performed.

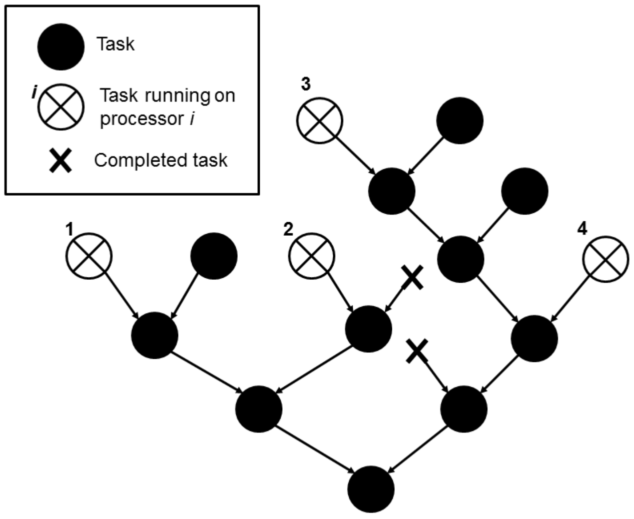

Figure 11. An example of an abstract state

START
$$\frac{X[i] = \mathsf{NO\_CODE} \quad n \text{ ready with respect to } (N,E,X,M) \quad X'[i] = (n, \mathit{BodyOf}(n))}{(N,E,X,M) \longrightarrow (N,E,X',M)}$$

STEP
$$\frac{X[i] = (n,k) \quad \mathit{Step}(k,M) = (k',M') \quad X'[i] = (n,k')}{(N,E,X,M) \longrightarrow (N,E,X',M')}$$

FORK
$$\frac{X[i] = (n, \mathit{fork}(n_1,n_2,n_j)) \quad X'[i] = \mathsf{NO\_CODE} \quad N' = N \setminus \{n\} \uplus \{n_1,n_2,n_j\} \quad E_1 = \text{replace } n \text{ by } n_j \text{ everwhere in E} \quad E' = E_1 \cup \{(n_1,n_j),(n_2,n_j)\}}{(N,E,X,M) \longrightarrow (N',E',X',M)}$$

END
$$\frac{X[i] = (n, \mathit{noop}) \quad X'[i] = \mathsf{NO\_CODE} \quad N' = N \setminus \{n\} \quad E' = \{(n_1,n_2) \in E \mid n_1 \neq n\}}{(N,E,X,M) \longrightarrow (N',E',X',M)}$$

Figure 12. Transitions for abstract states

– M: the abstract shared memory (under the x86-TSO weak-memory model).

In future definitions, we let n range over nodes, e range over edges, and i over processor identifiers.

The abstract state allows us to determine certain properties of threads, which will be defined here.

Definition 2 (Running nodes)**.** In a state (N,E,X,M), a node $n \in N$ is *running* if there exist i and k such that $X[i] = (n,k)$.

Definition 3 (Ready nodes)**.** In a state (N,E,X,M), a node $n \in N$ is *ready* if n has no incoming edges, that is, $\forall n' \in N.\ (n',n) \notin E$.

Figure 11 represents an abstract state as a graph. This graph appears to have the structure of a tree, but with edges reversed (pointing toward the root.) Indeed, Lemma B.1 will show this to be the case in general. The "leaves" of the reverse tree (nodes with no incoming edges) are the ready nodes. Certain ready nodes n are labelled with a processor i such that $X[i] = (n,k)$ for some k. These are the running nodes.

We now define abstract transitions, which complete the abstract semantics by providing a set of rules by which the scheduler can change from one abstract state to another.

Definition 4 (Abstract transitions)**.** Figure 12 shows the definition of the relation $A \longrightarrow A'$, which describes the allowed transitions for abstract states. Compositions of these transition rules capture all of the behaviors which we allow of our abstract scheduler. The auxiliary function Step executes one (atomic) instruction from the code, returning the new state of abstract memory and the remaining code. The auxiliary function BodyOf returns the code associated with a node. Rule START starts executing a ready node n on an idle processor. Rule STEP performs an atomic step in a running node and updates the continuation and memory. Rule FORK removes a forking node, replacing it with nodes n_1, n_2 and n_j, where n_j depends on both n_1 and n_2 and n_j inherits any outgoing edges of n. Rule END removes a running node n without forking.

Definition 5 (Accessible abstract state)**.** An abstract state A is accessible if it can be reached by a sequence of transitions from an abstract initial state, that is, if $(\{n_0\}, \emptyset, \emptyset, \emptyset) \longrightarrow^* A$ holds for some n_0.

We can now prove by induction on the relation that the graph (N, E) maintains the structure of a reverse tree, as described informally above.

Lemma B.1 (Reverse tree)**.** *If (N, E, X, M) is an accessible abstract state, then it forms a reverse tree in the sense that (N, E^R), where $E^R = \{(n_2, n_1) \mid (n_1, n_2) \in E\}$, is a tree with nodes of arity at most 2.*

Proof. By induction on the reduction sequence, maintaining the property that only ready nodes can be running (i.e., assigned by X). We consider each of the rules that may apply.

- Rule START: does not change the graph (N, E), and ensures that the assigned thread is ready.
- Rule STEP: does not change the graph (N, E), and thread not change the set of running threads.
- Rule FORK: replacing the node n by n_j in the tree preserves its structure; adding two new nodes n_1 and n_2 with associated edges pointing to n_j also preserves the tree structure.
- Rule END: removes a ready node and its associated edges; because the node is ready, it corresponds to a leaf, so removing it preserves the reverse tree structure.

□

B.3 Work stealing states

The definition of the abstract scheduler enforces rules that must hold for any valid scheduler (for example, processors may execute only one thread at a time and may only start ready threads) but makes no mention of the scheduling mechanism. In particular, a *work-stealing* scheduler imposes additional invariants on the state. We extend abstract states with additional information relating to work stealing, and strengthen the induction hypothesis of the proof with the additional work-stealing invariants. We refer to these extended abstract states as *work stealing states*.

The state of a work-stealing scheduler must include an assignment of ready threads to processors (the threads in each processor's deque). It also includes information about the threads on which a non-ready thread depends (its *parents*.) Since these parent threads arose from a forking thread, we refer to these as the *branches* of a join, and label them as the left and right branches, keeping with prior literature on work stealing. The right branch may be stolen by another processor, resulting in a *remote join* or may remain at the original processor in a *local join*. For reasons relating to the proof, we add an additional designation to represent single branches of join threads which currently have only one parent. If the right branch of a remote join finished before the left, we will re-designate the left branch as a single branch. For technical reasons, we need not do this in the case of a left branch finishing first. In a local join, the left branch will always finish first because it is pushed onto the queue first.

We also assign some nodes an *owner*, if it is clear that the node has exactly one processor "responsible for it." This includes nodes in a processor's deque, but also non-ready nodes both of whose parents are owned by the same processor.

The following definitions formalize branch and ownership labeling.

Definition 6 (Branch labeling)**.** In an accessible state (N, E, X, M), a *branch labeling* is a map that binds each edge $e \in E$ to a value from the set $\{\mathsf{LEFT}, \mathsf{RIGHT}, \mathsf{SINGLE}\}$, in such a way that the following two conditions are satisfied:

- if $\mathit{Branch}(n_1, n_j) = \mathsf{SINGLE}$, then the edge (n_1, n_j) must be the only incoming edge of n_j,
- if $\mathit{Branch}(n_1, n_j) = \mathsf{LEFT}$, then there must exist an edge $(n_2, n_j) \in E$ such that $\mathit{Branch}(n_2, n_j) = \mathsf{RIGHT}$,
- if $\mathit{Branch}(n_1, n_j) = \mathsf{RIGHT}$, and if there exists another edge $(n_2, n_j) \in E$, then $\mathit{Branch}(n_2, n_j) = \mathsf{LEFT}$.

Note: if a branch is tagged LEFT, then the corresponding right branch must belong to the tree; however, if a branch is tagged RIGHT, then the corresponding left branch might be absent. If $(n_1, n_2) \in E$ and $\mathit{Branch}(n_1, n_2) = b$, we will sometimes write $(n_1, n_2, b) \in E$.

Definition 7 (Ownership labeling)**.** In an accessible state (N, E, X, M), an *ownership labeling* is a map that binds each node in N to either a processor id or to the constant NO_OWNER, in such a way that, if we call this map Owner and define $\mathit{OwnedBy}(i)$ as $\{n \in N \mid \mathit{Owner}(n) = i\}$, the following three conditions are satisfied:

- no leaf node n has $\mathit{Owner}(n) = \mathsf{NO\_OWNER}$.
- for every processor i, the set of nodes in $\mathit{OwnedBy}(i)$ and the edges between them form a subtree of (N, E).
- if $i \neq j$ then the subtrees $\mathit{OwnedBy}(i)$ and $\mathit{OwnedBy}(j)$ are completely disjoint.

Figure 13 shows the work stealing information for the tree of Figure 11. Each edge is labeled with its branch labeling, and the $\mathit{OwnedBy}(i)$ sets are circled. Note that the circled sets of nodes and the edges between them form subtrees of the large tree.

Definition 8 (Work stealing state)**.** We say that A is a *work stealing state* if A is an accessible abstract state such that there exists a branch labeling Branch and an ownership labeling Owner for this state A. In this case, we write $\mathit{Workstealing}(A, \mathit{Branch}, \mathit{Owner})$.

B.4 Relation between abstract and concrete states

The following definitions operate on the abstract state and are helpful to characterize the parts of the abstract state corresponding to the content of certain data structures involved in the concrete states.

Definition 9 (Deque)**.** In a work stealing state $(A, \mathit{Branch}, \mathit{Owner})$, we define $\mathit{Deque}(i)$ to be the list of leaf nodes in the subtree $\mathit{OwnedBy}(i)$ listed in the order in which they appear in the tree, from left to right.

Recall that open remote joins must be watched by the two processors responsible for resolving that join. That is, if $(n_1, n, b_1), (n_2, n, b_2) \in E$, then n_j is a join thread that must be watched by two processors. Since each processor must know whether it is watching the left or right branch of the join, it is sometimes convenient to instead talk about processors watching *edges*. This hints at part of the relation between

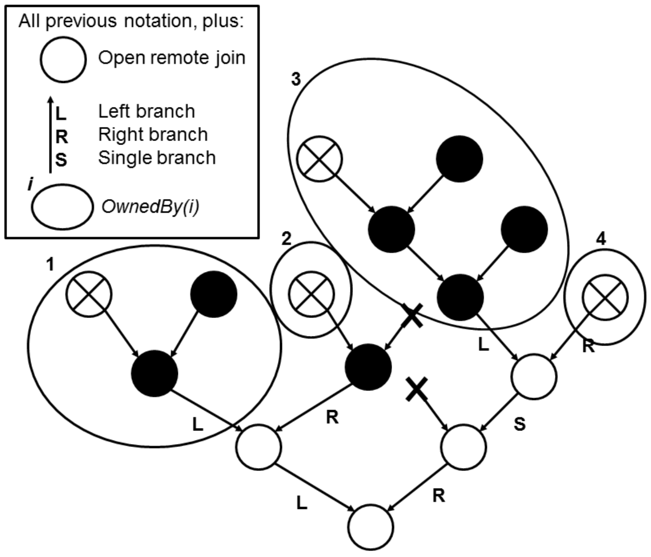

Figure 13. An example of a work-stealing state

the abstract and concrete states: while nodes in the abstract state correspond to concrete threads, edges in the abstract state correspond to continuations. Since concrete watch lists contain continuations, abstract watch lists will contain edges. That is, one processor, i, will watch (n_1, n, b_1) and another, j, will watch (n_2, n, b_2). We write $\mathit{Watcher}(n_1) = i$ and $(n_1, n, b_1) \in \mathit{Watchlist}(i)$, and similarly for n_2 and j. These notations are defined below.

Definition 10 (Watcher processor)**.** In a work stealing state $(A, \mathit{Branch}, \mathit{Owner})$, we define $\mathit{Watcher}$ to be a map that binds each node $n \in N$ to the identifier of the owner of the leftmost leaf accessible from n. The formal definition is by recursion on the tree. If $\mathit{Owner}(n)$ is equal to some processor id i (this is always the case when n is a leaf) then we let $\mathit{Watcher}(n) = i$. Otherwise, if $\mathit{Owner}(n)$ is equal to NO_OWNER then we let $\mathit{Watcher}(n) = \mathit{Watcher}(n')$ where n' is the left parent of n if it exists, or the unique parent otherwise.

Definition 11 (Watch list)**.** In a work stealing state $(A, \mathit{Branch}, \mathit{Owner})$, we define $\mathit{Watchlist}$ to be a map that binds each processor to the set of edges that it is responsible for watching periodically. Intuitively, $\mathit{Watchlist}(i)$ corresponds to the edges whose source node is owned by no processor but watched by i, and which are not the only incoming edge into the destination node. Formally,

$$\mathit{Watchlist}(i) = \{(n_j, b_1) \mid \exists n_1, n_2, b_1, b_2. \quad \begin{array}{ll} & n_1 \neq n_2 \\ \wedge & \mathit{Owner}(n_j) = \text{NO\_OWNER} \\ \wedge & (n_1, n_j, b_1) \in E \\ \wedge & (n_2, n_j, b_2) \in E \\ \wedge & i = \mathit{Watcher}(n_1) \end{array} \}$$

In Figure 14, the running example tree from the previous subsections is annotated with information relating to deques and watchers. The deque of each processor is circled. Note that the running thread is the leftmost thread of each deque and is specially marked. For each open remote join, its two incoming edges are marked with the number of the processor watching that continuation. In all cases, this is the owner of the leftmost leaf of the subtree rooted at the edge's source node.

We are now ready to define the relation R that relates an abstract state with a concrete state. This relation is built from a large number of invariants, some enforcing properties of either the abstract or concrete states, and some enforcing relations between them. Intuitively, the relation need only ensure that the data structures of the concrete state (the deques of the processors, the dependencies recorded in the continuation structures, etc.) reflect the structure of the abstract state. As long as this relation holds, the scheduler can be considered valid.

However, we must include more information in the relation in order to ensure that it is preserved over concrete transitions. This is similar to strengthening the induction hypothesis of an inductive proof. In particular, we require that the abstract state is a work-stealing state, and include in the large relation R some sub-relations that mention the branch and ownership assignments and the corresponding data structures of the concrete state.

Definition 12 (Relation between abstract and concrete states)**.** Given an abstract state A and a concrete state C, the relation $R\,A\,C$ holds if there exists Branch and Owner such that $(A, \mathit{Branch}, \mathit{Owner})$ is a work stealing state, if there exists a bijection ϕ between nodes of the concrete state A and thread objects allocated in the concrete state C, and if all the invariants listed below are true. In the statement of these invariants, we let (N, E, X, M) be the components of the abstract state A. Moreover, we let D_i denote the array of deque objects (`deque[i]`) let W_i denote the array of watch lists (`watchlist[i]`), let K_i denote the array of current continuations (`cur_cont[i]`), let $\mathrm{Mem}(C)$ denote the concrete application state, all of which are allocated in the concrete state C. L_i denotes the next line of code (numbered based on the psuedocode figures presented in this paper) to be executed by processor i. We denote continuations by $\{t, b\}$, where t is the continuation task and b is the branch.

We introduce the invariants by category, starting each category with an informal description of those invariants.

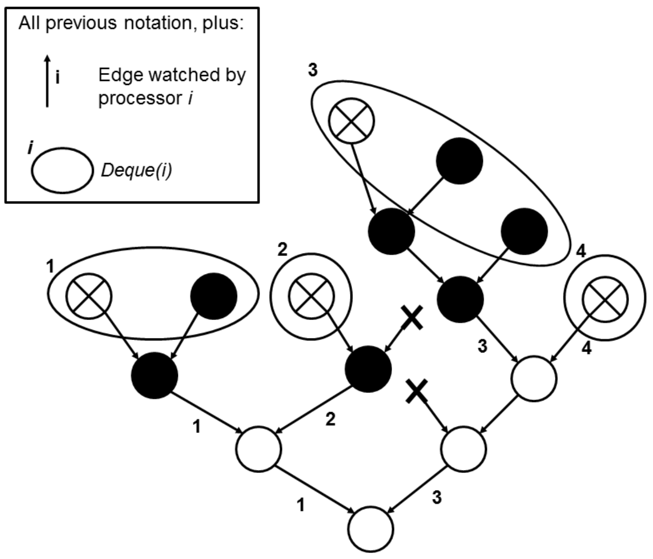

Figure 14. The example annotated with deques and watcher processors labeled

- **Deques**: The abstract set $Deque(i)$ matches the concrete deque D_i, using ϕ to map nodes to threads. If a thread is currently running or being stolen, it will appear in the abstract state but not the concrete deque. These threads are concatenated to the deque to complete the bijection.

– R_D: For each processor i, the list obtained by mapping ϕ over $Deque(i)$ is equal to the concatenation of $RunningThread(i)$ and ${D_i}^i$ (items taken from the bottom of the deque to the top) and $MigratingThread(i)$, where $MigratingThread$ is defined in the load balancing protocol and where $RunningThread(i)$ is the singleton list made of $\phi(n)$ if $X[i] = (n, k)$ for some k, is the singleton made of $\mathtt{t}_{41}$ if $L_i \in [42; 43]$ (between pop and run), and is the empty list otherwise.

- **Threads**: The bijection ϕ correctly maps nodes and edges of the abstract state to corresponding concrete structures: the run method of the thread is preserved (i.e. the node and thread have the same code), and the continuation of $\phi(n)$ corresponds to the outgoing edge of n.

– R_{T_1}: The bijection ϕ maps nodes to thread objects with corresponding run methods, in the sense that for any $n \in N$, we have $BodyOf(n) = \phi(n)$.run.

– R_{T_2}: The continuations of the threads implement the edges of the abstract state in the following sense: for any $n \in N$, if $(n, n') \in E$, then $\phi(n)$.conti $= \{\phi(n'), Branch(n, n')\}$, where this equality holds in the eye of processor i if $Owner(n)$ is i, and holds in the eye of all processors if $Owner(n)$ is NO_OWNER.

- **Current continuations**: For all processors i, the continuation of the thread assigned to i is correctly stored in the current-continuation variable of i, in the following sense:

– R_{C_1}: If $X[i] = (n, k)$ for some $k \neq fork(\cdot)$ or if $L_i \in [45, 47]$ (beginning of the fork function), then $K_i^i = \phi(n)$.conti.

– R_{C_2}: If $X[i] =$ NO_CODE, then $K_i^i =$ NULL $\vee L_i = 43$ (after assigning the current continuation but before running the run method).

- **Running threads**: For all processors i, the thread assigned to i in the abstract state matches that running on i in the concrete state, in the following sense:

– R_{X_1}: $L_i \in \phi(n)$.run for some $n \in N \Rightarrow X[i] = (n, k)$ for some $k \neq noop$.

– R_{X_2}: $L_i \in [45, 53] \Rightarrow X[i] = (n, fork(n_1, n_2, n_j))$ for some n, n_1, n_2, n_j such that $\mathtt{t1}_{45} = \phi(n_1), \mathtt{t2}_{45} = \phi(n_2), \mathtt{tj}_{45} = \phi(n_j)$.

– R_{X_3}: otherwise, $X[i] =$ NO_CODE.

– R_{X_4}: $X[i] = (n, k)$ for some $n, k \neq noop \Rightarrow L_i \in \phi(n)$.run. Remark: by invariant R_{T_1}, the code $\phi(n)$.run corresponds to k.

– R_{X_5}: $X[i] = (n, noop) \Rightarrow L_i \in \{44\} \cup [55, 132]$ (running the `handle_cont` function) Remark: $X[i]$ could also be NO_CODE when L_i is in this range.

– R_{X_6}: $X[i] = (n, fork(n_1, n_2, n_j))$ for some $n, n_1, n_2, n_j \Rightarrow L_i \in [45, 53] \wedge \mathtt{t1}_{45} = \phi(n_1), \mathtt{t2}_{45} = \phi(n_2), \mathtt{tj}_{45} = \phi(n_j)$ (running the `fork` function with matching arguments.)

- **Application State**: The application state (which is stored as a concrete representation of memory even in the abstract state) must match in A and C.

– R_M: $M = \mathrm{Mem}(C)$

- **Watchlists**: The concrete watch list for each processor i contains all and only the continuations corresponding to edges in $Watchlist(i)$, except when watch lists are being transferred. We let $Wltrans_i$ denote a watch list being transferred to i and enforce that all of the

continuations in this watch list belong in the abstract watch list of i. We also define the shorthand $W(i, n, b)$ to indicate that (n, n_j, b) is in i's abstract watch list for some n_j. Formally:
Let $W(i, n, b) = \exists n_j.(n, n_j, b) \in \mathit{Watchlist}(i) \wedge \phi(n).\text{status[b]}^i = \mathsf{WORK} \wedge \neg B_{i,\phi(n)}$ (the property B will be defined in the load balancing section).
Let $\mathit{Wltrans}_i = \mathsf{if}\ L_i \in [141, 150] \wedge t.\text{status[b2]}^{\bar{i}} = \mathsf{TRANSFER}\{S\}\ \mathsf{then}\ S\ \mathsf{else}\ \emptyset\}$, where $\bar{i}$ is the processor that "owns" t.status[b2]. This concept will be defined formally in the proof of the join resolution protocol.
Let $\mathit{Wlinc} = \{87, 103, 121, 142\}$. These are lines of code at which the state of the watch lists may be inconsistent.

– R_{W_1}: For all n, $\{\phi(n), b\} \in W_i \Rightarrow W(i, n, b) \vee L_i \in \mathit{Wlinc}$.
– R_{W_2}: $W(i, n, b) \Rightarrow \{\phi(n), b\} \in W_i \vee \{\phi(n), b\} \in \mathit{Wltrans}_i$.
– R_{W_3}: $\{\phi(n), b\} \in \mathit{Wltrans}_i \Rightarrow W(i, n, b)$.

The above are the main invariants relating the abstract and concrete states. At any point in time, they are sufficient to hold that the abstract state is a valid model of the concrete state of the scheduler. However, the proofs that the load balancing and join resolution protocols maintain the relation will require additional invariants, which will be introduced in the appropriate sections of the proof.

- **Load balancing invariants**: See Appendix B.5.
- **Join resolution invariants**: See Appendix B.6.

B.5 Correctness of the load balancing algorithm

Notation We introduce single-letter variables to represent the variables from the code. R denotes the round numbers (`round`). Q denotes the query fields (`in_query`). T denotes the reception fields (`received`). We subscript these variables with the index of a processor, and we use the notation for describing the contents of store buffers and of the shared memory cell associated with the variables. We extend the notation x_n from the previous section, and use the notation j^i_{72}, r^i_{73} and t^i_{100} to refer to the variables in the stack of processor i when these variables are in its execution scope (these are the three specific variables that will be of special important to the proof, and the line numbers refer to the lines on which each variable is assigned the relevant value.)

Definition 13 (Load balancing auxiliarly variables)**.** We introduce several auxiliary definitions to help in stating the invariants.

$A_i \equiv L_i \in [71; 90]$
(i is in the main loop of the acquire function)

$B_i \equiv A_i \wedge (\forall k \neq i.\ \bar{T}^k_i = \mathsf{nil}) \wedge (\bar{T}^i_i = \mathsf{NULL} :: \mathsf{nil} \vee (\bar{T}^i_i = \mathsf{nil} \wedge T_i = \mathsf{NULL}))$
(i has set its reception field and is in the main loop of acquire)

$C_{i,j,r} \equiv L_i \in [78; 82] \wedge j = \mathsf{j}^i_{72} \wedge r = \mathsf{r}^i_{73} \leq R^j_j$
(i has made an answer to j at round r and is waiting for an answer)

$E_{i,j,t} \equiv L_i \in [78; 87] \wedge i \neq j \wedge j = \mathsf{j}^i_{72} \wedge (\forall k \neq j.\ \bar{T}^k_i = \mathsf{nil}) \wedge ((\bar{T}^j_i = \mathsf{t} :: \mathsf{nil} \wedge \mathrm{H}) \vee (\bar{T}^j_i = \mathsf{nil} \wedge T_i = t))$
where H asserts that the last write in $\bar{T}^j_i$ occured before the write of any value greater than r^i_{73} in $\vec{R}^j_j$
(i has obtained the thread t from j but has not yet pushed it its in deque)

$E_i \equiv \exists j, t.\ E_{i,j,t}$

Definition 14 (Migrating threads)**.** We define $\mathit{MigratingThread}$ to be a map that binds each processor to a list of length zero or one, containing the thread that has just been popped from the deque and not yet sent, or the thread has just been received and not yet pushed to the deque. More precisely,

– $\mathit{MigratingThread}(i)$ is the singleton list made of t^i_{100} if $L_i \in [101; 103]$ (i has just popped a thread and not yet sent it),
– $\mathit{MigratingThread}(i)$ is the singleton list made of t if $E_{i,j,t}$ is true (i has just obtained a thread t from j but has not yet pushed it to its deque),
– $\mathit{MigratingThread}(i)$ is the empty list otherwise.

Remark: this definition asserts that the ownership transfer (in the sense of the ownership labeling Owner) happens at the moment where the sender writes the thread pointer into the reception field of the target.

We now define several invariants which are part of R, but only relate to the load balancing protocol.

Definition 15 (Load balancing Invariants)**.** The following invariants are satisfied at any time during the execution.

$\mathcal{I}_{A_1} \equiv$ If A_i is false then $\forall k.\ \bar{T}^i_k = \mathsf{nil}$.

$\mathcal{I}_{A_2} \equiv$ If A_i is false and if $\vec{Q}^i_i$ contains a query with id i, then round number of this query is less than R^i_i.

$\mathcal{I}_{R_1} \equiv$ $\forall j \neq i.\ \bar{R}^i_j = \mathsf{nil}$ (meaning that processors update only their own round numbers).

$\mathcal{I}_{R_2} \equiv$ $\vec{R}^i_i$ is a list of stricly decreasing values (meaning that round numbers only increase through time).

$\mathcal{I}_{Q_1} \equiv$ All the queries in $\bar{Q}^i_j$ have an id equal to i (meaning that queries always contain the id of their sender).

$\mathcal{I}_{Q_2} \equiv$ All the queries in $\vec{Q}^i_j$ have a round number no greater than R^j_j,
with one exception: the case where $L_j = 111$ (block function) and the query has round number $R^j_j + 1$.

$\mathcal{I}_{Q_3} \equiv$ If a query in $\vec{Q}^k_j$ has a round number equal to R^j_j and an id i with $i \neq j$, then $C_{i,j,r}$ is true for $r = R^j_j$
and this query is the last write in $\vec{Q}^i_j$ and it occured after the last write in $\bar{T}^i_i$.

$\mathcal{I}_{T_1} \equiv$ If A_i is true (i in the main loop of acquire) then either B_i or E_i is true
(in the first case, T^i_i is NULL, whereas in the second case T^i_i is or will soon become not NULL).

$\mathcal{I}_{T_2} \equiv$ If $L_i \in [84; 85]$ (just about to read the reception field) then $\mathsf{r}^i_{73} < R_j$, where $j = \mathsf{j}^i_{72}$.

$\mathcal{I}_{T_3} \equiv$ If $L_i \in [86; 85]$ (just read a non-null pointer) then E_i is true and T_j is the thread pointer read, where $j = \mathsf{j}^i_{72}$.

We prove all of these invariants stable under flushes from store buffers to main memory.

Lemma B.2 (Stability of the invariants)**.** *The flush of any value from the tail of a store buffer to the shared memory is an operation that preserves the invariant.*

Proof. Observation 1: variables of the form X^i are never affected by a flush of a write into the memory cell X. Observation 2: a property that holds of all the values in a vector of the form $\bar{X}^i$ is preserved by flushes of writes taken from this buffer. Observation 3: a property that holds of all the values in a vector of the form $\vec{X}^i$ is preserved by flushes of writes taken from this buffer, if i is the only processor writing in X. Observation 4: a property that holds of all the values in all the vectors of the form $\vec{X}^i$ (for all i) is preserved by flushes of writes taken from this buffer. Observation 5: an assertion that a buffer $\bar{X}^i$ is empty is preserved by flushes because no flush can be performed.

– $\mathcal{I}_{A_1}$: Preserved, by observation 5.
– $\mathcal{I}_{A_2}$: Preserved, by observation 2 and the fact that round numbers only increase.
– $\mathcal{I}_{R_1}$: Preserved, by observation 5.
– $\mathcal{I}_{R_2}$: Preserved, thanks to $\mathcal{I}_{R_1}$ and because a flush would just trim the list $\vec{R}^i_i$, perserving the fact that it is ordered in decreasing order.
– $\mathcal{I}_{Q_1}$: Preserved, by observation 2.
– $\mathcal{I}_{Q_2}$: Preserved, by observation 4 and the fact that round numbers only increase.
– $\mathcal{I}_{Q_3}$: We use observation 3. Assume that the premise of $\mathcal{I}_{Q_3}$ was true. If the round number of j increases, then the premise becomes false, so the statement holds. Otherwise, we use the fact that $C_{i,j,r}$ is stable, which is true because round numbers can only increase. Also, the constraint on the order of the writes gets preserved.
– $\mathcal{I}_{T_1}$: We show that both B_i and E_i are stable. For the stability of B_i, there are two cases. If $k \neq i$, we use observation 5. Otherwise, the flush originates from $\bar{T}^i_i$, which must therefore be non empty. In this case, the NULL value gets transfered to T_i and $\bar{T}^i_i$ becomes empty. For the stability of E_i, we use a similar argument for the flush of the thread pointer t. Also, the assertion H is stable because it is an ordering constraint.
– $\mathcal{I}_{T_2}$: Preserved, by the fact that round numbers only increase.
– $\mathcal{I}_{T_3}$: Preserved, using the stability of E_i established before.

□

Lemma B.3 (Increment of the round number)**.** *The incrementation of the round number of processor i at any time preserves all invariants, except when i has just sent a thread, in which case it must be the case that the thread pointer was sent just before.*

Proof. Straightforward, because all invariants except $\mathcal{I}_{Q_3}$ always assert that a value is less than a round number. For $\mathcal{I}_{Q_3}$, the comparison to the round number appears in a premise, so changing the round number preserves the invariant, as explained in the previous lemma. The particular constraint on the thread pointer is needed to preserve invariant E_k, where k is the processor to which i sends a thread. □

The following lemmas show that the invariants are preserved under atomic instructions executed in the functions of the load balancing algorithm.

Lemma B.4 (Correctness of the acquire function)**.** *Let A be an abstract state and C be a concrete state. If $R\ A\ C$, and $C \Longrightarrow C'$, where this transition is accomplished by processor i executing an atomic instruction in the* `acquire` *function, then either $R; A; C'$ or there exists A' such that $A \longrightarrow A'$ and $R; A'; C'$.*

Proof. We consider one by one each of the lines that perform a write operation and thereby modify the shared memory, and also all the lines that are mentioned as boundries in the invariants.

– Line 70. Before the execution of this line, A_i is false, so the conclusions of $\mathcal{I}_{A_1}$ and $\mathcal{I}_{A_2}$ are true, so $(\forall k. \bar{T}_k^i = \mathsf{nil})$. After the execution of this line, the premise A_i of $\mathcal{I}_{T_1}$ becomes true. We show that this invariant is satisfied by proving B_i to be true. Indeed, we have $(\forall k \neq i.\ \bar{T}_i^k = \mathsf{nil})$ and $\bar{T}_i^i = \mathsf{NULL} :: \mathsf{nil}$.

– Line 77. We have to show that $\mathcal{I}_{Q_1}, \mathcal{I}_{Q_2}$ and $\mathcal{I}_{Q_3}$ remain true. Let $j = \mathsf{j}_{72}^i$ denote the target processor. $\mathcal{I}_{Q_1}$ remains true because the id associated with the query sent is i. $\mathcal{I}_{Q_2}$ remains true because the round number r_{73}^i associated with the query sent is the round number of j as seen be i, that is, the value R_j. This value is, by invariant $\mathcal{I}_{R_2}$, less than R_j^j. $\mathcal{I}_{Q_3}$ remains true because after the write, $C_{i,j,r}$ holds: $L_i \in [78; 82]$ and and $\mathsf{r}_{73}^i \leq R_j^j$ as we have just explained.

– Line 81. Preserves invariants for exactly the same reasons as the send on line 77.

– Line 84 (when reaching this line). First, if we exit the loop, then it means that the round number of j read by i, that is, R_j, has become greater than r_{73}^i. So, $\mathcal{I}_{T_2}$ is established. Second, we have to show $\mathcal{I}_{Q_3}$ to be preserved. Consider a query whose round number was equal to R_j^j before existing the loop. When in the loop, we had $\mathsf{r}_{73}^i = R_j$. Because we exit the loop, it must be now the case that $\mathsf{r}_{73}^i < R_j$. Since $R_j \leq R_j^j$, the premise is now false, so the invariant is satisfied.

– Line 85. The thread pointer T_i^i is tested. By $\mathcal{I}_{T_1}$, either B_i is true or E_i is true. Assume that the value T_i^i is not NULL. This implies that B_i is false. So E_i must be true. We thus establish $\mathcal{I}_{T_3}$.

– Line 86. See the proof of the join protocol.

– Line 87. Global invariant R_D is maintained because $\mathcal{I}_{T_5}$ is now true because after this line E_i becomes false since the function exits (the writing of the round number can be safely considered to happen before the push in the deque, since the deque is private).

– Line 88. As proved separately in Lemma B.3, incrementing the round preserves all invariants.

□

Lemma B.5 (Correctness of the communicate function)**.** *The* `communicate` *function preserves R in the same way as defined in Lemma B.4.*

Proof. We consider one by one each of the lines that perform a write operation. Note that `communicate` is not called during `acquire`, so we can assume A_i to be false.

– Line 103. Let $k = \mathtt{q\_id(q)}$ and let t be the thread popped from the deque. When reaching this point in the code, Q_i^i is a query such that its round number is equal to R_i^i. By invariant $\mathcal{I}_{A_2}$, queries in $\vec{Q}_i^i$ with id i all carry a round number less than R_i^i. So, it must be the case that $k \neq i$. By $\mathcal{I}_{Q_1}$, it must be the case that $Q_i^i = Q_i$. At this point, by $\mathcal{I}_{Q_3}$ applied to processor k and this particular query, processor k is such that $C_{k,i,r}$ is true for $r = R_i^i$, moreover, because this query has reached the shared memory, it must be the case that $\bar{T}_k^k$ is empty. By definition of $C_{k,i,r}$, it must be the case that $i = \mathsf{j}_{72}^k$. After the write in the reception field of k, E_k becomes true, so $\mathcal{I}_{T_1}$ is preserved. At this point, we update the *Owner* map: the node n such that $\phi(n) = t$ is now mapped to processor k instead of processor i.

– Line 102. See proof of the join resolution protocol.

– Line 104. By Lemma B.3, incrementing the round preserves all invariants, using for $E_{i,j,t}$ the fact that the write of the round number occurs after the write in the reception field of the target.

□

Lemma B.6 (Correctness of the block function)**.** *The* `block` *function preserves R in the same way as defined in Lemma B.4.*

Proof. The `block` function is only called from inside the `acquire` function, so we can assume A_i to be true.

– Line 110 We have to show that $\mathcal{I}_{Q_1}, \mathcal{I}_{Q_2}$ and $\mathcal{I}_{Q_3}$ remain true. $\mathcal{I}_{Q_1}$ remains true because the id associated with the query sent is i. $\mathcal{I}_{Q_2}$ remains true because the next line is 111. $\mathcal{I}_{Q_3}$ remains true because for this new query, the assumption $j \neq i$ is false.

– Line 111. Invariant $\mathcal{I}_{Q_2}$ is now fixed, as the round number has catched up on the query. By Lemma B.3, incrementing the round number preserves all invariants.

□

Lemma B.7 (Correctness of load balancing)**.** *Assume that an abstract state A and a concrete state C are related in the sense that relation $R\,A\,C$ holds. Then, for any transition $C \Longrightarrow C'$ in the concrete state that involves a step in the acquire function, the block function or the communicate function, the relation $R\,A\,C'$ holds.*

Proof. By definition of $R\,A\,C$, we have $(A, \mathit{Branch}, \mathit{Owner})$. When performing load balancing, the computation graph does not change, so there is no transition on the abstract state A. The branch labeling does not change either. The ownership labeling is only updated at the time when a thread gets sent in the communicate function, as explained in the lemma associated with the verification of this functions. □

B.6 Correctness of the join resolution algorithm

The lemmas in this section show that the global invariants and relation between abstract and concrete states are preserved under transitions relating to the join algorithm. We first present helper definitions and invariants relating specifically to the join algorithm.

Definitions In the invariants and the proof, we use the following additional helper definitions. $P_k(n)$ is true if and only if processor k has executed line n in the current round of the join algorithm (with the current continuation.) $F(l, v)$ states that location l contains, or will eventually contain, the value v and will not have any other value written to it later.

$$\begin{aligned} P_k(n) &\equiv L_k \text{ has held the value } n \text{ with the current continuation } \mathsf{c}_{55}. \\ F(l,v) &\equiv (l = v \wedge \overline{l}^i = \mathsf{nil}\ \forall i) \vee (l = \mathsf{WORK} \wedge \overline{l}^i = \{v\}\ \textit{for some } i \wedge \forall j \neq i, \overline{l}^j = \mathsf{nil}) \end{aligned}$$

In showing that invariants involving $F(l, v)$ are preserved, it will be important to show that this condition is stable, that is, it is not violated when writes are flushed from write buffers into shared memory. This is proven in Lemma B.8.

Lemma B.8. *The invariant $F(l, v)$ is stable under any flush of a write buffer to shared memory.*

Proof. Either $l = v$ or $l = \mathsf{WORK}$. If $l = v$, then, for all i, $\overline{l}^i = \mathsf{nil}$ and no flush to l can occur. Suppose $l = \mathsf{WORK}$. Then, there exists a processor i such that $\overline{l}^i = \{v\}$. Since for all $j \neq i$, $\overline{l}^i = \mathsf{nil}$, the flush to l must come from the write buffer of i, resulting in $l = v$ and $\overline{l}^i = \mathsf{nil}$. This maintains the invariant. □

Definition 16 (Ownership of status fields)**.** We say S is a status field belonging to k if $S = t.\text{status}[\text{b}]$ where $\{t, b\}$ is either in W_i or is $\phi(n).\text{cont}$ for a node n in the deque of i or already executed by i.

Lemma B.9 (Uniqueness of status field ownership)**.** *A status field S belongs to at most one processor.*

Proof. Deques are disjoint by the correctness of the load balancing algorithm, so no two processors can own the same thread, and the structure of the reverse tree ensures that distinct threads have distinct continuations. Watch lists are disjoint because $\mathit{Watcher}$ is uniquely defined. A continuation $\phi(n).\mathit{cont}$ cannot be in one processor's deque and another processor's watch list because this would imply $\mathit{Owner}(n) \neq \mathsf{NO\_OWNER}$ but $\mathit{Owner}(n) \neq \mathit{Watcher}(n)$. □

We next present the set of invariants specific to the join algorithm.

Invariants

Definition 17 (Permanent join algorithm invariants)**.** $\mathcal{I}_{J1}$ enumerates four cases into which all join threads fall. The remaining invariants concern writes to status fields: only the owner of a status field may write to it, and may only write certain values depending on the value of the corresponding field. $\mathcal{I}_{J4}$ states that, under certain conditions, the value read by a processor was the last value written to it.

$\mathcal{I}_{J1} \equiv$ For all $n_j \in N$, one of the following cases applies:

1. There doesn't exist n_1 such that $(n_1, n_j) \in E$, and processor k is running `resolve_join` with a continuation c such that $c.\text{a.join} = \phi(n_j)$.
2. $(n_1, n_j) \in E$, there does not exist $n_2 \in N$ such that $(n_2, n_j) \in E$, and no processor other than $\mathit{Watcher}(n_1)$ is running `resolve_join` or `handle_cont` with a continuation c such that $c.\text{join} = \phi(n_j)$.
3. $(n_1, n_j, \mathsf{LEFT})$ and $(n_2, n_j, \mathsf{RIGHT}) \in E$
4. There doesn't exist n_1 such that $(n_1, n_j) \in E$, and no processor is running `handle_cont` or `resolve_join` with a continuation c such that $c.\text{a.join} = \phi(n_j)$.

In the following invariants, let k be any processor, let S_k be a status field belonging to k and let $S_{\overline{k}}$ refer to the other status field of the same thread, that is, $S_{\overline{k}} = t.\text{status}[1 - \text{b}]$.

$$\begin{aligned} \mathcal{I}_{J2} &\equiv \forall m \neq k, \bar{S_k}^m = \mathsf{nil}. \\ \mathcal{I}_{J3} &\equiv \vec{S_k}^k \text{ consists of at most one instance of } \mathsf{WORK} \text{ possibly preceded by:} \\ & \mathsf{READY} \text{ (if } S_{\overline{k}}^k = \mathsf{WORK} \vee S_{\overline{k}}^k = \mathsf{NULL}) \\ & \mathsf{READY} \text{ possibly preceded by } \mathsf{TRANSFER}\{\cdot\} \text{ (if } S_{\overline{k}}^k = \mathsf{ACK}) \\ & \mathsf{ACK} \text{ or } \mathsf{READY} \text{ (if } S_{\overline{k}}^k = \mathsf{READY} \text{ or } S_{\overline{k}}^k = \mathsf{TRANSFER}\{\cdot\}) \\ \mathcal{I}_{J4} &\equiv S_{\overline{k}}^k \neq \mathsf{WORK} \wedge S_{\overline{k}}^k \neq \mathsf{NULL} \wedge S_k{}^k \neq \mathsf{ACK} \Rightarrow S_{\overline{k}}^k = S_k{}^k \end{aligned}$$

Definition 18 (Invariant for case 1)**.** The following invariant applies only in case 1 and ensures that this processor will not schedule the join thread, since the other processor did so.

$$\mathcal{I}_{J5} \equiv L_k \neq 118 \wedge L_k \neq 132 \wedge S_k{}^k \neq \mathsf{ACK} \wedge (F(S_{\overline{k}}, \mathsf{ACK}) \vee \mathit{Branch}(n_1, n_j) = \mathsf{RIGHT}) \wedge X[k] = \mathsf{NO\_CODE}$$

Definition 19 (Invariant for case 2)**.** The following invariant applies only in case 2 and ensures that the remaining processor will schedule the join thread. Let $t = \phi(n_1)$, let $j = \mathit{Watcher}(n_2)$ and let $S_k = \phi(n_j).\text{status}[\mathit{Branch}(\text{n}_1, \text{n}_\text{j})]$ (if $\mathit{Branch}(n_1, n_j) \neq \mathsf{SINGLE}$).

$$\mathcal{I}_{J6} \equiv \mathit{Branch}(n_1, n_j) = \mathsf{SINGLE} \vee S_k{}^k = \mathsf{ACK} \vee (F(S_{\overline{k}}, \mathsf{READY}) \wedge \mathit{Branch}(n_1, n_j) = \mathsf{LEFT}) \vee L_j = 53$$

Definition 20 (Invariants for case 3)**.** The following invariants apply only in case 3, and rule out invalid states of the status fields: both processors may not write ACK, a processor only writes READY in `resolve_join` and a processor only writes ACK after reading READY. Let $t_1 = \phi(n_1)$, let $t_2 = \phi(n_2)$, and let a be the activation record such that $\phi(n_1).\text{c.a} = \phi(n_2).\text{c.a} = a$. Let $i = \mathit{Watcher}(n_1)$, let $j = \mathit{Watcher}(n_2)$, let $S_i = a.\text{status}[0]$ and let $S_j = a.\text{status}[1]$. k refers to either i or j and $\overline{k}$ refers to the other.

$$\begin{array}{lcl}\mathcal{I}_{J7} & \equiv & \neg({S_i}^i = \mathsf{ACK} \land {S_j}^j = \mathsf{ACK}) \\ \mathcal{I}_{J8} & \equiv & \neg P_k(120) \Rightarrow {S_k}^k \neq \mathsf{READY} \\ \mathcal{I}_{J9} & \equiv & {S_k}^k \neq \mathsf{READY} \Rightarrow {S_{\overline{k}}}^{\overline{k}} \neq \mathsf{ACK}\end{array}$$

In the proof of some invariants, we use the additional fact that if, at some point in the past, ${S_k}^{\overline{k}}$ held a value other than WORK or READY in some cases, it still holds that value. This fact follows from invariants $\mathcal{I}_{J2}$ and $\mathcal{I}_{J3}$.

The next two lemmas prove that any transition in the concrete state preserves the invariants relating to the join protocol.

Lemma B.10 (Stability of the invariants)**.** *The flush of any value from the tail of a store buffer to the shared memory is an operation that preserves the invariants.*

Proof. Note that we only need to consider the stability of invariants that involve ${S_k}^i$ for $i \neq k$. This is because all other references to shared memory cells involve either write buffers, which aren't affected by flushes to main memory, or ${S_k}^k$, which, by $\mathcal{I}_{J2}$, can't be affected by writes from any write buffers other than that of k. This fact, together with Lemma B.8 implies the stability of all remaining invariants.

– $\mathcal{I}_{J3}$. This invariant can only be invalidated by a flush to shared memory if
  - WORK or ACK is flushed to $S_{\overline{k}}$ and $\vec{S_k}^k$ contains ACK. However, this would mean that $S_{\overline{k}}$ previously held READY or TRANSFER$\{\cdot\}$ and, by $\mathcal{I}_{J2}$, before the flush, ${S_{\overline{k}}}^{\overline{k}}$ held WORK or ACK ahead of READY or TRANSFER$\{\cdot\}$ or
  - WORK, READY or TRANSFER$\{\cdot\}$ is flushed and $\vec{S_k}^k$ contains TRANSFER$\{\cdot\}$. In this case, $S_{\overline{k}}$ previously held ACK and, before the flush, ${S_{\overline{k}}}^{\overline{k}}$ held WORK or READY or TRANSFER$\{\cdot\}$ ahead of ACK.

In either case, $\mathcal{I}_{J3}$ would have been violated before the flush.

– $\mathcal{I}_{J4}$. If ${S_k}^k \neq$ ACK, then by $\mathcal{I}_{J2}$ and $\mathcal{I}_{J3}$, ${S_k}^{\overline{k}} \neq$ ACK, so if there is a write to $S_{\overline{k}}$, it must have been pulled from $\bar{S}_{\overline{k}}^{\overline{k}}$ and there are no other such writes in $\bar{S}_{\overline{k}}^{\overline{k}}$. Therefore, the write flushed to main memory is the one value in $\vec{S_k}^k$, and is now the value that will be read from $S_{\overline{k}}$ by both k and $\overline{k}$.

□

Lemma B.11. *Let A be an abstract state and C, C' be concrete states, Suppose $R\,A\,C$ and $C \Longrightarrow C'$. If the transition $C \Longrightarrow C'$ takes the form of processor k executing an atomic instruction in* $[61, 67] \cup [113, 150]$ *then $R\,A\,C'$ or there exists an abstract state A' such that $R\,A'\,C'$ and $A \longrightarrow A'$.*

Proof. The proof is broken into the cases described in $\mathcal{I}_{J1}$. The proof then follows from $\mathcal{I}_{J1}$. We need not consider case 4 because a relevant transition cannot occur in case 4. Let $A = (N, E, X, M)$. Unless an A' is specified, we show that $R\,A\,C'$.

Case 1

– Line 117. By invariant $\mathcal{I}_{J5}$, this test fails and so the else branch will be taken.

– Lines 126, 129. If control exits `resolve_join` at either of these points, all invariants are preserved since $X[k]$ is already NO_CODE. n_j is now in case 4 of $\mathcal{I}_{J1}$.

– Line 131. If control reached this point, by $\mathcal{I}_{J5}$, $\mathit{Branch}(n_1, n_j) =$ RIGHT, so this test will fail and control will exit `resolve_join`. As above, this preserves all invariants.

Case 2

– Lines 59, 118 and 132. Since $(n_1, n_j) \in E$, n_j isn't already ready and isn't in a deque. Let $A' = (N \setminus \{n_1\}, E \setminus \{(n_1, n_j)\}, X[k \mapsto \mathit{None}], M)$. n_j has no incoming edges so is ready in A'. Since t_1 was executing, by R_D and the definition of $\mathit{Deque}(\cdot)$, n_1 was the leftmost leaf of the subtree formed by $\mathit{OwnedBy}(i)$, and thus n_j is now the leftmost leaf of the new subtree formed by $\mathit{OwnedBy}(i)$. Since n_j is added to the bottom of D_i, R_D is preserved. The removal of a node and its outgoing edge maintains $\mathit{Workstealing}(A', \mathit{Branch}, \mathit{Owner})$ and the preservation of the remaining invariants implies $R\,A'\,C'$. $A \longrightarrow A'$ by END. n_j is now in case 4 of $\mathcal{I}_{J1}$.

– Line 120. Since this branch was taken, ${S_k}^k \neq$ ACK, so writing to ${S_k}^k$ cannot invalidate $\mathcal{I}_{J6}$.

– Line 126. Since this branch was taken, ${S_k}^k \neq$ ACK, so by $\mathcal{I}_{J6}$, $F(S_{\overline{k}}, \mathsf{READY})$. Control exited the while loop, so ${S_{\overline{k}}}^k \neq$ WORK and thus ${S_{\overline{k}}}^k =$ READY. This means control will not reach this line.

– Lines 131. By $\mathcal{I}_{J6}$, $\mathit{Branch}(n_1, n_j) =$ LEFT so control will not exit `resolve_join`.

– Line 65 By assumption, the other parent of t_j has been run, so $D_k =$ nil and this line will not run, so control will not exit from `handle_cont` here.

Case 3

– Line 141. $\mathcal{I}_{J2}$ is maintained because this line writes only to S_k. $\mathcal{I}_{J3}$ is maintained because R_{W_1} and R_{W_2} imply that ${S_k}^k = \mathsf{WORK}$, which by $\mathcal{I}_{J3}$ means that only WORK has been written into ${\bar{S}_k}^k$. $\mathcal{I}_{J7}$ is maintained because the success of the test ensures ${S_{\overline{k}}}^k = \mathsf{READY} \neq \mathsf{ACK}$. By $\mathcal{I}_{J4}$, ${S_{\overline{k}}}^{\overline{k}} \neq \mathsf{ACK}$, so $\mathcal{I}_{J9}$ is maintained.

– Line 118. n_j has incoming edges, so it isn't ready and isn't in a deque. Let $A' = (N \setminus \{n \mid (n, n_j) \in E\}, \{(n, n') \in E \mid n' \neq n_j\}, X[k \mapsto \mathsf{NO\_CODE}], M)$. n_j has no incoming edges so is ready in A'. $R\,A'\,C'$ and $A \longrightarrow A'$ as above for case 2. Suppose n_j is now in case 1 of $\mathcal{I}_{J1}$ and processor $\overline{k}$ is executing `resolve_join` on the corresponding continuation. Then modify A' such that $X[\overline{k}] = \mathsf{NO\_CODE}$. Since ${S_k}^k = \mathsf{ACK}$, $\mathcal{I}_{J7}$ gives ${S_{\overline{k}}}^{\overline{k}} \neq \mathsf{ACK}$. Invariant $\mathcal{I}_{J2}$ implies that ${S_k}^{\overline{k}} \neq \mathsf{READY}$, and these two results give that $L_{\overline{k}} \neq 118, 132$, so invariant $\mathcal{I}_{J5}$ is preserved. Otherwise, n_j is now in case 4 of $\mathcal{I}_{J1}$.

– Line 120. Since the test on Line 117 failed and $\mathcal{I}_{J8}$ applies, ${\vec{S_k}}^k$ previously had no instance of READY or ACK. By $\mathcal{I}_{J3}$, then, ${S_{\overline{k}}}^k = \mathsf{WORK}$. READY is added, and invariant $\mathcal{I}_{J3}$ is preserved. $\mathcal{I}_{J4}$ must be preserved since no writes have been flushed to S_k and ${S_k}^{\overline{k}}$ must be WORK. Invariant $\mathcal{I}_{J8}$ is preserved since $L_k = 120$, so $\neg P_k(120)$ is now false.

– Lines 126, 129 and 131. Suppose control exits from `resolve_join` and `handle_cont` on one of these lines. If $X[k] = (n, \mathit{noop})$, then let $A' = (N \setminus \{n\}, \{(n_1, n') \in E \mid n_1 \neq n\}, X[k \mapsto \mathit{None}], M)$. $A \longrightarrow A'$ by End. This coincides with t_k being removed from T in C'. n_j is not yet ready and is in case 2 since it still has an edge from $t_{\overline{k}}$. If the line is 126 or 129, then ${S_{\overline{k}}}^k = \mathsf{ACK}$. By $\mathcal{I}_{J4}$, ${S_{\overline{k}}}^{\overline{k}} = \mathsf{ACK}$. If the line is 131, then $\mathit{Branch}(n_k, n_j) = \mathsf{RIGHT}$ and ${S_{\overline{k}}}^k = \mathsf{READY}$. We then know that $\mathit{Branch}(n_{\overline{k}}, n_j) = \mathsf{LEFT}$ and, by $\mathcal{I}_{J4}$, ${S_{\overline{k}}}^{\overline{k}} = \mathsf{READY}$. In either case, $\mathcal{I}_{J6}$ holds for the remaining edge to n_j, and $R\,A'\,C'$. The removal of a node and its edge preserves the fact that $\mathit{OwnedBy}(i)$ is a subtree of (N, E) disjoint from all other $\mathit{OwnedBy}$ sets, so $\mathit{Workstealing}(A', \mathit{Branch}, \mathit{Owner})$.

– Line 132. See Line 118 above. Suppose n_j is now in case 1 of $\mathcal{I}_{J1}$ and processor $\overline{k}$ is executing `resolve_join` on the corresponding continuation. Then, since ${S_k}^k = \mathsf{READY}$, $\mathcal{I}_{J2}$ implies ${S_{\overline{k}}}^{\overline{k}} \neq \mathsf{ACK}$. Since control reached this branch, the definition of Branch means that the corresponding continuation has branch RIGHT, and these two results give that $L_{\overline{k}} \neq 118, 132$, so invariant $\mathcal{I}_{J5}$ is preserved. Otherwise, n_j is in case 4.

– Line 65 By the standard deque invariant for work-stealing schedulers, if the deque is non-empty, the thread at the bottom of the deque was the right branch of this thread. Let $A' = (N \setminus \{n_1\}, E \setminus \{(n_1, n_j)\}, X[i \mapsto \mathsf{NO\_CODE}], M)$. $A \longrightarrow A'$ by End. This coincides with t_k being removed from T in C'. The join thread is not yet ready since it still has an edge from $t_{\overline{k}}$. Update $\mathit{Branch}(n_2, n_j)$ to be SINGLE. R_{T_2} is preserved since $t.c.\text{branch} = \mathsf{SINGLE}$, there does not exist an edge $(n_1', n_j) \in E \setminus \{(n_1, n_j)\}$ and control is not in `resolve_join`. This means n_j is in case 2 of $\mathcal{I}_{J1}$. $\mathcal{I}_{J6}$ is preserved since the branch is $\mathit{Branch}(n_2, n_j) = \mathsf{SINGLE}$. The removal of a node and its edge preserves the fact that $\mathit{OwnedBy}(i)$ is a subtree of (N, E) disjoint from all other $\mathit{OwnedBy}$ sets, so $\mathit{Workstealing}(A', \mathit{Owner})$.

– Line 59. Since both branches exist, $c.\text{branch} \neq \mathsf{SINGLE}$, so this line is not reached.

□

Lemma B.12. *Let $A = (N, E, X, M)$ be an abstract state and C, C' be concrete states, with $\mathit{Workstealing}(A, \mathit{Owner})$ and $R\,A\,C$. If $C \Longrightarrow C'$, then C' satisfies the conjunction of invariants $\mathcal{I}_{J1}$ through $\mathcal{I}_{J9}$.*

Proof. The stability of these invariants has already been shown. We show cases not covered by Lemma B.11

– Line 52. Invariant $\mathcal{I}_{J6}$ now applies, and is true since the added thread is owned by k and $L_k = 53$.

– Line 53. Invariants $\mathcal{I}_{J7}$ through $\mathcal{I}_{J9}$ now apply to t_1, t_2 and $t_j = \phi(n_j)$. The status fields have been initialized on the previous lines such that these invariants are true.

– Lines 86 and 87. These actions might change ownership of a status field $S = \phi(n).\text{status}[\mathsf{RIGHT}]$ from j to k for some n, b. Invariants will be preserved unless $S^k \neq S^j$, that is, if $\bar{S}^j \neq \mathsf{nil}$. If $\bar{S}^j$ consists only of WORK, then since $\phi(n)$ must have been initialized before the offered thread and the offered thread must have been initialized before it was offered, this write has already been flushed to main memory. Processor j could not have written READY into S since this is only done after the right parent of n is finished, so i could only have written ACK in `watch`, but this would imply that the thread was already migrated.

– Line 150. Invariants will be preserved unless $\bar{S}^j \neq \mathsf{nil}$ where $S = t.\text{status}[\text{b}]$ for some $\{t, b\}$ transferred from j to k. However, all writes to such fields by j must have occurred before j cleared its watch list and wrote $\mathsf{TRANSFER}\{s\}$. Since this write was flushed to memory, all writes to S must have been flushed as well.

□

B.7 Proof of the correctness lemma

Before proving the correctness lemma, we prove that atomic transitions preserve the watch list invariants of R, which have so far not been covered by any previous lemmas.

Lemma B.13. *Any atomic transition preserves the watch list invariants.*

Proof. We now consider all atomic transitions in the pseudocode that affect the watch lists or deques.

– Lines 86, 102, 120 and 141. Establish Wlinc_i, preserving R_{W_1}.

– Lines 121 and 142. $\mathsf{c}_{55}.\text{join}.\text{status}[\mathsf{c}_{55}.\text{branch}]^i \neq \mathsf{WORK}$, so R_{W_2} is preserved.

– Line 87. Let $t_j = \mathtt{t}_{84}.\text{cont.join}$. $\mathit{Owner}(t_j)$ is now NO_OWNER since there exists an n_1 such that $(n_1, n_j) \in E$ such that $\mathit{Owner}(n_1) = j \neq i$, so since $\mathit{Owner}(\mathtt{t}_{84}) = i$ and $\mathtt{t}_{84}.\text{cont} \in \mathit{Watchlist}(i)$, and line 86 preserved the invariants.
– Line 103. Let $t_j = \mathtt{t}_{100}.\text{cont.join}$. $\mathit{Owner}(t_j)$ is now NO_OWNER since there exists an n_2 such that $(n_2, n_j) \in E$ and $\mathit{Owner}(n_2) = j \neq i$, so since $\mathit{Owner}(t) = i$ and $\{t_j, \mathsf{LEFT}\} \in \mathit{Watchlist}(i)$, and line 102 preserved the invariants. The invariants on $\mathit{Watchlist}(j)$ are not violated since $B_{j,\mathtt{t}_{100}}$.
– Line 129. By Defintion 7, $\mathit{OwnedBy}(i)$ becomes empty, so there can be no $n \in N$ such that $\mathit{Watcher}(n) = i$, and clearing the watch list preserves the watch list invariants. Any thread n that previously had $\mathit{Watcher}(n) = i$ now has $\mathit{Watcher}(n) = j$, where j is the processor that won this race. After line 128, $S_i{}^i = \mathsf{TRANSFER}\{s\}$ where s contains all continuations $\{n', b\}$ such that $W(i, n', b)$ was previously true, for which $W(j, n', b)$ is now true. This preserves R_{W_2} and R_{W_3}.
– Line 150. This line transfers $\mathit{Wltrans}_i$ to W_i, preserving R_{W_2}. R_{W_3} implies R_{W_1}.

□

We now recall and prove Lemma 5.2.

Lemma 5.2. There exists a relation R for which the following implication holds:
if $C \Longrightarrow C'$ and $R\,C\,A$, then there exists A' such that $A \longrightarrow^* A'$ and $R\,C'\,A'$.

Proof. We verify the transitions not covered by previous lemmas. That is, those outside the load balancing and join resolution algorithms.

– Line 41. L_i becomes 42, so $\mathit{RunningThread}(i)$ becomes $\mathtt{t}_{41}$. Since this was previously the bottom element of $D_i{}^i$ with $\mathit{RunningThread}(i) = \emptyset$, R_D is preserved.
– Line 42. R_{C_2} is preserved.
– Line 43. Let $A' = (N, E, X[i \mapsto (n, \mathit{BodyOf}(n))], M)$, where n is the node such that $\phi(n) = \mathtt{t}_{41}$. This corresponds to calling the run method of t in the concrete state, so R_{X_1} and R_{X_5} are preserved. The corresponding transition in the abstract state is allowed since $n \in \mathit{Deque}(i)$ and so n is ready. Since $\phi(n) = \mathtt{t}_{41}$, $\mathit{RunningThread}(i)$ doesn't change and R_D is preserved. The previous line set $K_i{}^i$ to $\phi(n).\text{cont}$, preserving R_{C_1}.
– End of `fork`. If $X[i] = (n, \mathit{fork}(n_1, n_2, n_j))$, then let $A' = (N \cup \{n_1, n_2, n_j\}, E \cup \{(n_1, n_j), (n_2, n_j), (n_j, n')\}, X[i \mapsto \mathsf{NO\_CODE}], M)$, where $\phi(n') = \phi(n).\text{cont.join}$. Since the continuation field of t_j was set when L_i was 46, by R_{C_1}, the current continuation at that point was $\phi(n).\text{cont}$, and so t_j has the correct continuation. Extend ϕ such that $\phi(n_1), \phi(n_2)$ and $\phi(n_j)$ are t_1, t_2 and t_j respectively. Let $\mathit{Owner}(n_1) = \mathit{Owner}(n_2) = \mathit{Owner}(n_j) = i$. Since n was previously at the bottom of the deque, R_D states that it was the leftmost leaf owned by i. It has been replaced by n_1 and n_2 whose corresponding threads have been pushed, in order, onto D_i, so R_D is maintained. The invariant on $\mathit{OwnedBy}(i)$ is maintained because n has been replaced by another subtree contained in $\mathit{OwnedBy}(i)$. The transition $A \longrightarrow A'$ is allowed.
– The program code takes an atomic step. Let $A' = (N, E, X[i \mapsto (n, k')], M')$, where $\mathit{Step}(k, M) = (k', M')$. By definition, $\mathrm{Mem}(C') = M'$, so R_M is preserved.

□

B.8 Proof of liveness

The following lemmas lead to a proof of the liveness part of Theorem 5.1. Lemma B.14 shows that, under the assumption of fairness stated in Section 5, all processors call `watch` periodically during any infinite reduction of a concrete state.

Lemma B.14 (Periodic watch list operations)**.** *Let A be an abstract state, and let i be any processor. Let $C_0, C_1, \ldots$ be concrete states. If $R\,A\,C_0$ and $C_0 \Longrightarrow C_1 \Longrightarrow C_2 \Longrightarrow \ldots$ and $c \in \mathit{Watchlist}(i)$, then there exists some n such that in C_n, $L_i = 135$ with $\mathtt{c}_{135} = c$ (running the outer loop of the `watch` function with c).*

Proof. We assume that `watch` is called periodically by busy processors, so consider processors i not currently in `run` methods of threads. Either i will enter a `run` method at some C_m or it remains in one of the `while` loops in `acquire`, `resolve_join` or `watch` in all C_m for $m > n$ for some n. In the latter two cases, `watch` may be safely called and we assume the implementation will do so periodically. In the first case, since i is running `acquire`, $\mathit{OwnedBy}(i) = \emptyset$, so there is no $n \in N$ such that $\mathit{Watcher}(n) = i$, and $\mathit{Watchlist}(i) = \emptyset$.

We therefore know that there exists some n_1 for which i calls `watch` in C_{n_1}. There then exists some C_n for $n > n_1$ such that in C_n, $L_i = 135$ with $\mathtt{c}_{135} = c$, unless i becomes stuck in the inner `while` loop. However, if this happens, the continuation that caused it to become stuck has been removed from $\mathit{Watchlist}(i)$. By induction, there exists some C_{n_2} for $n_2 > n_1$ at which `watch` will be called again with the smaller watch list. Eventually, c will be reached in the outer loop or `watch` will be called with a watch list containing only c, at which point it will follow immediately that $\mathtt{c}_{135} = c$. □

Lemma B.15 states that if a processor i is in the `while` loop in `resolve_join`, a processor is watching or has already written into the corresponding status field. This shows that i will eventually receive a reply to the READY message it has written.

Lemma B.15. *Let A and C be abstract and concrete states, respectively, such that $\mathit{Workstealing}(A, \mathit{Branch}, \mathit{Owner})$ for some Owner and $R\,A\,C$. If $L_i \in [122, 123]$ with $\mathtt{c}_{113} = \{\phi(n), b\}$ for some $n \in N$, then there exists some processor j and node n' such that $(n', n, 1 - b) \in \mathit{Watchlist}(j)$ or $\phi(n).\text{status}[1 - \text{b}]^j \neq$ WORK.*

Proof. Proceed in the first three cases given by $\mathcal{I}_{J1}$ for $\phi(n)$. In case 1, the result follows from $F(S_{\overline{k}}, \mathsf{ACK})$ in $\mathcal{I}_{J5}$. In case 2, since this branch was reached, it must be the case that $F(S_{\overline{k}}, \mathsf{READY})$. In case 3, there exists an $n' \in N$ such that $(n', n, 1-b) \in E$, and so $(n', n, 1-b) \in \mathit{Watchlist}(\mathit{Watcher}(n'))$. □

Lemma B.16 asserts that the concrete state makes progress relative to the abstract state. That is, there is no infinite reduction of concrete states that does not result in a change in the abstract state.

Lemma B.16 (Liveness)**.** *Let A_0 be an abstract state with an ownership labelling Owner_0 such that $\mathit{Workstealing}(A_0, \mathit{Owner})$. Let $C_0, C_1, \ldots$ be concrete states in which ready threads exist. If $R\,A_0\,C_0$ and $C_0 \Longrightarrow C_1 \Longrightarrow C_2 \Longrightarrow \ldots$, then there exist some A_1, Owner_1 and n such that $R\,A_1\,C_n$ and $A_0 \longrightarrow A_1$.*

Proof. If any processor is in a `run` method of a thread, then, given the fairness assumption of Section 5, that processor will eventually take a step $C_{n-1} \Longrightarrow C$, and the abstract state will take the corresponding step $A_0 \longrightarrow A_1$ with rule STEP. We may therefore assume that no processors are running threads and are instead in one of the functions of the scheduler. However, a processor in one of these functions will eventually proceed through and enter the `run` method of a thread unless it remains in one of the `while` loops in `acquire`, `resolve_join` or `watch` in all C_m for $m > n$ for some n. Therefore, such an infinite reduction sequence can only exist if all processors are in one of these loops. If all processors are in `acquire`, then $\mathit{Deque}(i) = \mathsf{nil}$ for all processors i, and the computation is finished. Thus, there exists at least one processor i such that one of the following applies.

- $L_i \in [122, 123]$. In this case, Lemma B.15 implies that there is a processor j such that $\mathsf{status}_{122}[b2]^j \neq \mathsf{WORK}$, in which case i will eventually see this write and exit the loop, or $\mathit{Watchlist}(j)$ contains the corresponding continuation to c_{113}. In this second case, since $\mathsf{status}_{122}[b1]^i = \mathsf{READY}$, this write will eventually be seen by j. By Lemma B.14, j will run the outer `watch` loop on this continuation at some point after the flush of this write, and will write ACK into $\mathsf{status}_{122}[b2]$, which will cause i to enter the loop when this write is flushed.
- $L_i \in [146, 147]$. In this case, $\mathsf{b1}_{114} = \mathsf{RIGHT}$ and there exists a processor j that has executed line 120 with $\mathsf{b1}_{114} = \mathsf{LEFT}$ and written READY into $\mathsf{status}_{146}[b2]$. Similarly, i has written ACK into $\mathsf{status}_{146}[b1]$. This write will eventually be seen by j, which will exit the loop and write $\mathsf{TRANSFER}\{s\}$. This write will eventually be seen by i, which will exit its loop.

□

B.9 Proof of Theorem 5.1

Recall Theorem 5.1:

Theorem 5.1 (Correctness and liveness)**.** For any initial thread node n_0 and concrete state C in which all processors have run out of threads and are running `acquire`,

$$\begin{array}{rlll} & C_0(n_0) \Longrightarrow^* C & \text{implies} & (\{n_0\}, \emptyset, \emptyset, \emptyset) \longrightarrow^* (\emptyset, \emptyset, \emptyset, \mathrm{Mem}(C)) \\ \text{and} & C_0(n_0) \Longrightarrow^\infty & \text{implies} & (\{n_0\}, \emptyset, \emptyset, \emptyset) \longrightarrow^\infty \end{array}$$

Proof. We first show that if $A = (N, E, X, M)$ is any abstract state such that $R\,A\,C$, then $A = (\emptyset, \emptyset, \emptyset, \mathrm{Mem}(C))$. Since all processors have run out of threads, for all processors i, it must be the case that $D_i = \mathsf{nil}$. By R_D, we must have that $\mathit{Deque}(i) = \mathsf{nil}$. Thus, $\bigcup_i \mathit{OwnedBy}(i) = \emptyset$ and, since all leaf nodes must have an owner and (N, E) must be a reverse tree, $N = E = \emptyset$. If all processors are running `acquire`, then X is the trivial map from all processors to NO_CODE. This gives the desired result. The correctness part can then be shown by inductive application of Lemma 5.2. Liveness is shown by inductive application of Lemma B.16. □

C. Bound on the number of open remote joins

Lemma C.1. *Let A be a work stealing state. Let Owner be the corresponding ownership labelling. Let P_b be the number of busy processors (processors i such that $X[i] \neq$ NO_CODE). There are at most $P_b - 1$ threads t_j such that $(t_1, t_j), (t_2, t_j) \in E$ and $\mathit{Owner}(t_j) =$ NO_OWNER and $\mathit{Watcher}(t_1) \neq \mathit{Watcher}(t_2)$.*

Proof. Since for each busy processor i, $\mathit{OwnedBy}(i)$ consists of a subtree of (N, E), we may consider a reverse tree constructed from (N, E) by collapsing each of these P_b subtrees into a single (leaf) node. The desired threads consist of the non-leaf nodes of this reverse tree, each of which has exactly two parents. The number of non-leaf nodes in such a tree is at most one less than the number of leaves, so the number of such threads is at most $P_b - 1$. □