Greedy Sharing: Load Balancing on Weakly Consistent Memory

(Regular Submission)

Umut A. Acar
umut@mpi-sws.org

Arthur Charguéraud
charguer@mpi-sws.org

Mike Rainey
mrainey@mpi-sws.org

Max-Planck Institute for Software Systems

Abstract

An efficient online scheduler is crucial for balancing irregular parallel computations in a multiprocessor system. Over the last two decades, variants of the work-stealing scheduler have emerged as a popular choice for hardware shared-memory systems. The state-of-the-art work-stealing algorithms can guarantee near-optimal asymptotic complexity by relying on simple yet powerful techniques to balance total load among processors. Implementations of work stealing algorithms, however, continue to rely on synchronization operations, such as atomic read-write operations (e.g., compare and swap), to guarantee correctness of concurrent accesses to shared task pools. Furthermore, since work-stealing algorithms are traditionally designed by assuming a sequentially-consistent memory model, their implementations use additional memory fences on modern multiprocessor machines. Memory fences and atomic-read-write operations are known to be expensive in general, especially because they can require exclusive access to shared resources, such as the memory.

In this paper, we present the *greedy-sharing algorithm* for load balancing on weakly-consistent hardware shared-memory architectures, such as modern multicore computers. Greedy sharing combines ideas from work-sharing and work-stealing algorithms to eliminate all synchronization operations. As in work sharing, busy processors perform load balancing by sharing their work; as in work-stealing, tasks migrate only from busy to idle processors. In greedy sharing, data races can occur when multiple processors target the same idle processor. To recover safely from such data races, we design a protocol for task sharing that makes use of particular time-stamping technique, which is attributed to Lamport. We present a specification of the algorithm, prove its correctness on the X86-TSO weak memory model, and prove an upper bound for its execution time. We have implemented the algorithm as part of our C++ library for parallel programming. We present experiments to show that the algorithm is practical.

Keywords: Load balancing, work stealing, work sharing, weak-memory model, X86-TSO, synchronization operations, atomic read-write, memory fences, correctness proof, efficiency proof.

1 Introduction

In parallel applications, an online scheduler is crucial for balancing the load between processors. In the seventies, Brent [6] showed that greedy schedulers, which assign a task to a processor as soon as the processor becomes idle and a task becomes available, perform within factor two of the optimal schedule. But one important cost is absent from Brent's theorem: the cost of scheduling. Indeed, naive implementations of greedy scheduling that use a centralized task queue offer poor scalability due to the high cost of synchronization during scheduling. To reduce synchronization costs, researchers therefore developed distributed scheduling algorithms, where each processor owns a pool of ready tasks and communicates with other processors to balance load. Typically, distributed scheduling algorithms perform some combination of *work sharing* or *work stealing*. In work sharing, each processor actively shares its load with other processors [13]. In work stealing [8, 17], busy processors do not actively participate in load balancing, relying instead on idle processors to steal tasks from busy processors. Many variants of work-stealing and work-sharing algorithms, which differ in when and how they migrate tasks, have been proposed and studied.

By avoiding task migration except when a processor is idle and by prioritizing migration of large tasks, work stealing minimizes tasks migration (i.e., steals), and achieves asymptotic bounds close to those of a greedy scheduler, even when including the cost steals [2, 5]. Perhaps for this reason, work stealing is a popular algorithm for scheduling fine-grained parallel applications. Implementing work stealing on modern parallel and multicore computers, however, turns out to be challenging because of the careful engineering required to minimize use of synchronization operations. Broadly speaking, work-stealing implementations use two kinds of *synchronization operations*: 1) *locks* and *atomic read-write operations*, such as compare-and-swap, and 2) and *memory fences*. The first class of synchronization operations is used by the scheduler to prevent race conditions in steal operations and the second class to enforce key invariants by enforcing proper relative ordering of certain memory reads and writes.

Synchronization operations are known to be expensive, and because they require exclusive access to memory, they may also limit scalability. There has therefore been much work on eliminating synchronization from work stealing. Earlier implementations of the work-stealing algorithm [4] guarded access to deques with locks. Subsequent work reduced [16], and eventually eliminated locks [2] by offering a non-blocking algorithm that use atomic read-write operations (e.g., compare-and-swap). Chase and Lev [9] generalized the non-blocking algorithm to support growable deques; this sophisticated algorithm comes with a 48-page correctness proof [10]. Perhaps due to this complexity, non-blocking work stealing can be difficult to adapt for different load distribution strategies. For example, Hendler and Shavit present a non-blocking algorithm that supports the "steal-half" strategy, which can be important in fine-grained parallel applications (e.g., graph algorithms), but with quite complex modifications to the original algorithm [18].

Although they can eliminate locks, the aforementioned algorithms require memory-fence operations to ensure correctness on multicore computers with weak memory models. Memory fences are expensive synchronization operations, in part because they usually require flushing the write buffer. For example, Frigo et al observe that in an Intel multiprocessor, Cilk-5's THE protocol, which processors use to obtain work, spends half of its time executing a memory fence [16]. This fence, which ensures that each task is executed exactly once, is performed every time a processor accesses its deque. In more recent work, Michael et al [26] present idempotent work stealing which improves performance by 15%-20% and sometimes up to 50% by avoiding this particular memory fence. Idempotent work stealing, however, only works when tasks can be safely executed multiple times. While generalizing such an improvement would be highly desirable, recent results by Attiya et al [3] prove it to be impossible to eliminate synchronization from work stealing by showing that its shared deques belong to a class of concurrent data structures which demand memory fences on weak memory models (even on x86-TSO, which guarantees a total order on writes to shared memory [29]).

In this paper, we present a scheduling algorithm that completely eliminates the use of synchronization

operations in x86 multicore architecture with the total-store memory model.[1] Our algorithm (Section 3.1), which we call *greedy sharing*, combines ideas from work stealing and work sharing algorithms and carefully controls data races to eliminate synchronization. In greedy sharing, processors share tasks but only with idle processors. As in work stealing, each processor owns a deque (a doubly ended queue) for storing ready tasks. Each deque, however, is private and accessed only by the processor that owns it. As in work sharing, each busy processor periodically shares its tasks with idle processors by randomly polling other processors and sharing its work. Processors use several shared memory cells for communication. As in work stealing, when sharing a task, a processors sends the task at the top of its deque. Since its deques are private, greedy sharing is flexible. As we describe, for example, it can be easily adapted to support the "steal half" strategy.

Our greedy-sharing algorithm circumvents the impossibility result of Attiya et al [3] by avoiding shared deques. Since deques are private, no atomic read-write operations are needed to coordinate concurrent accesses to a processors deque. The memory cells used by processors to communicate, however, can be accessed concurrently, e.g., when multiple processors share a task with a processor at the same time. We show that accesses to the shared cells can be coordinated to guarantee correctness and efficiency without using synchronization operations. One crucial realization behind our algorithm is that data races that occur when multiple processors are trying to share with the same idle processor are not harmful, because it suffices for one processor to succeed in order for the idle processor to receive a task. Data races therefore do not harm efficiency. They could, however, harm correctness, as tasks can get overwritten and thus lost. We therefore design a two-phase *offer protocol* by which processors keep track of the tasks that they share with other processors and recover lost tasks due to data races. Our offer protocol utilizes a well-known timestamping technique due to Lamport [24].

Although our algorithm is relatively simple, the underlying invariants of correctness and efficiency are quite complex. We therefore prove that our proposed approach performs load balancing correctly by showing a key property of load balancing: that no task is lost and no task is duplicated (Section 3.2). We also show that our algorithm is efficient, by proving that the time for a parallel run can be bounded by the work and depth of the computation as in traditional work stealing. The major complexity in our proofs comes from the weak memory model, x86-TSO, that we assume.

We have developed a C++ library for general-purpose fine-grained parallelism that employs greedy sharing for scheduling (Section 3.6). Our experience suggests that it is relatively straightforward to implement our techniques, especially because of the lack of synchronization operations. We perform a preliminary evaluation of greedy sharing by assessing its performance and comparing it to a state-of-the-art work stealing algorithm. Our experiments shows (Section 3.7) that the algorithm is practical in practice and performs competitively with work stealing on modern multicore architectures with a relatively small number of cores.

2 Preliminaries: the x86-TSO weak memory model

To provide efficient access to memory, modern multiprocessors implement *weak memory models* that do not guarantee sequential consistency. In this paper, we consider the *x86-TSO* memory model [29]. This model describes the behavior of shared-memory multiprocessors with local *store buffers* (also called *write buffers*), where store operations performed by a processor are kept in a local store buffer for some time before being *flushed* to the shared memory. Only after the flush does the store operation become visible to all other processors. When a processor reads a memory cell, the store buffer is first checked to fetch the most-recent write; if there is no such write, the shared memory is accessed. At any point in time, a store request may be pulled out of the store buffer and applied to the shared memory once. Thus, in the x86-TSO model, there exists a total ordering of all store operations as reflected on the shared memory. Moreover, this total order is

[1] Light-weight fences such as load-load and store-store fences are needed in more relaxed memory models such as that of PowerPC which can locally reorder memory operations.

compatible with the order of the store operations requested by any given processor.

For establishing the efficiency of our algorithm, we must make the assumption that there is an (unknown) upper bound, written ϵ, on the delay between the time at which a store operation is performed by a processor and it is flushed to the shared memory. This reasonable assumption, which appears essential for shared memory, is guaranteed for AMD chipsets, but not officially guaranteed for Intel chipsets [29].[2] Neither our algorithm nor our correctness proof use or rely on the value of the bound ϵ. We use this bound only to establish the time-complexity bounds. In this sense, our model is similar to the partially-synchronous model of Dwork, Lynch, and Stockmeyer [12], where message delivery, a concept that is analogous to our store operations, is guaranteed to take place with some fixed but unknown delay, and where algorithms are not allowed to use this delay as a parameter.

3 Greedy Sharing, its Correctness, and Efficiency

3.1 The algorithm

In greedy sharing (pseudo code shown in Figures 1 and 2), each processor owns a *deque*, or doubly-ended queue, and operates on both ends of the deque with `push_bottom`, `pop_bottom`, `push_top`, `pop_top` operations. Each processor privately operates on its deque on both ends. Processors operate in phases of *working* and *acquiring* where they are working on tasks and trying to acquire tasks respectively. Each processor maintains a *round* number to differentiate between its own phases; starting from zero, the round number is equal to the number of transitions of a given worker from an acquiring to a working phase.

Busy processors perform load balancing by offering tasks to idle processors. An *offer* consists of the identity of a processor and of a round number. The algorithm guarantees two invariants that are key both to correctness and efficiency: 1) at any time, each processor has at most one outstanding offer, and 2) each processor accepts exactly one offer at any given round. To communicate, each processor uses several shared memory cells. The *out-offer* cell describes the outstanding offer of the processor (if any) by storing the identity of the target processor and the round of the target processor at the time when the offer was made. The *in-offer* cell is used by a processor to receive an offer: it contains the identity of the processor making the offer and the round associated with this offer. The *offered* cell contains a pointer on the task associated with the outstanding offer; this field can be accessed by the processor to which the offer is made.

For correctness, it is critical for processors to be able to make offers atomically. Packing the ID of the processor and the round number in a word makes this possible without needing further synchronization operations. We thus represent an offer as a single (64-bit) machine word (type `t_offer`). The function `o_make` constructs an offer, and the functions `o_id` and `o_rnd` project out the ID and the round numbers of the offer respectively. For the purpose of this paper, we define offers to include 23-bit ids, which makes it possible to support up 2^{23} processors, and 40 bit round numbers, making it possible computations up to 2^{40} rounds. The 64^{th} bit serves to represent negative values, which we use to represent special states: `WAITING`, which indicates that a processor is waiting for offers, and `RECEIVED`, which indicates the acceptance of an offer. Initially, all processors start out waiting, that is, with the value `WAITING` as content of their *in-offer* field. When a computation starts, the initial task is given to one particular processor, and its *in-offer* field is set to `RECEIVED`.

The main work of scheduling is performed by two functions, called `acquire` and `communicate`, each of which takes as argument the ID of the processor calling it. A processor calls `acquire` when it finds its deque empty. Each processor calls `communicate` periodically with some fixed frequency; we

[2]Note that if a hardware that does not guarantee a bound on the delay for store operations to reach shared memory, then it must provide an instruction to allow programs to flush the store buffers, because otherwise processor may not be able to communicate. In this case, we wan insert such an instruction at the appropriate place in our algorithm so as to ensure efficiency.

```
1  // Offer type with 40-bit round numbers
2  type t_offer = int64
3  t_offer o_make(int i, int r)
4    return r + (i << 40)
5  int o_id(offer o)
6    return (o >> 40)
7  int o_rnd(t_offer o)
8    return (o & ((1 << 40)-1))
```

```
9  // Global variables.
10 const t_offer RECEIVED = -1
11 const t_offer WAITING = -2
12 int round[P] = {0, 0, ..}
13 t_deque deque[P] = {DEQUE_EMPTY, ..}
14 t_offer in_offer[P] = {WAITING, ..}
15 t_offer out_offer[P] = {RECEIVED, ..}
16 t_task* offered[P] = {NULL, ..}
```

Figure 1: Greedy sharing: offers and the global state.

```
17 void acquire(int j) // j = ID of caller
18   offer o
19
20   // Complete current offer.
21   repeat o = out_offer[j]
22   until o == RECEIVED
23      or round[o_id(o)] ≠ o_rnd(o)
24
25   // If current offer cancelled, return.
26   if o != RECEIVED
27     communicate(j) // Take offer back.
28     return
29
30   // Else wait for an offer.
31   repeat
32     in_offer[j] = WAITING
33     repeat o = in_offer[j] until (o > 0)
34   until (o_rnd(o) == round[j])
35
36   // Accept offer.
37   push_bottom(deque[j],offered[o_id(o)])
38   out_offer[o_id(o)] = RECEIVED
39   round[j]++
```

```
40 // Called periodically by a busy processor
41 void communicate(int i) // i = ID of caller
42   // Check outstanding offer.
43   offer o = out_offer[i]
44   if o != RECEIVED
45     int j = o_id(o)
46     if round[j] == o_rnd(o)
47       // Check and resend if needed.
48       if in_offer[j] == WAITING
49         in_offer[j] = o_make(i, o_rnd(o))
50       return
51     else if out_offer[i] != RECEIVED
52       // Take offer back.
53       push_top(deque[i],offered[i])
54       out_offer[i] = RECEIVED
55
56   // Make offer, if possible, and needed.
57   if size(deque[i]) <= 1 then return
58   int j = random ∈ {0, .., P-1}\{i}
59   if in_offer[j] != WAITING then return
60   offered[i] = pop_top(deque[i])
61   int r = round[j]
62   out_offer[i] = o_make(j, r)
63   in_offer[j] = o_make(i, r)
```

Figure 2: Greedy sharing: `acquire` and `communicate` functions.

write δ for the size of the intervals between calls to `communicate`. The function `communicate` can offer a task to an idle processor. When made, an offer remains outstanding until it is *accepted*, that is, executed by the target processor, or until it is *rejected* by the target processor. Rejection of an offer occurs implicitly when the round number of the target processor exceeds that of the offer.

When inside the function `acquire`, an out-of-work processor first checks if it has an outstanding offer. If so, the processor waits for the outstanding offer to complete. If the offer is rejected, then the processor takes back its task and start working on the task by calling `communicate`. Otherwise, the processor sets its in-offer to `WAITING` in order to indicate that it is idle. The processor then repeatedly checks its in-offer field until an offers appears. If a received offer has a round number that is less than that of the processor, then the processor discards the offer and continues to wait. When the processor receives and offer whose round number matches its current round number, the processor accepts the offer by reading the `offered` field of the processor that sent the offer, and acknowledges the offer by setting the out-offer of the sender processor to `RECEIVED`. Having found work, the processor, then increments its round number and starts working on its deque. By incrementing its round number, the processor rejects all other outstanding offers that were made to it during that round.

Upon entering the function `communicate`, a busy processor first checks the status of its outstanding offer. (Recall that each processor has at most one outstanding offer). If the offer is received, then the

processor proceeds to make another offer. If not, then the processor probes the target of the outstanding offer to determine the offer's status. The probe reads the round number of the target processor, and examines the round to detect one of the two following cases: (1) If the target processor's round matches that of the outstanding offer then the offer is incomplete. In this case, the processor checkes whether the target is idle or not. If not idle, then the processors returns back to the work, waiting for the offer to complete. If idle, then the processor deduces that its offer was lots in a data race (otherwise it would have been received) and thus it resends its offer and goes back to work. (2) If the round number is not the same as that of the offer, then the target processor has found some work. Thus, the processor checks whether its offer was accepted by reading (again) its out-offer. If out-offer is set to `RECEIVED`, then the offer is complete and the processor proceeds to make another offer. Otherwise, the processor takes back its offer (since the target has increases its round number, any outstanding offer cannot be accepted), and proceeds to make another offer. Note that the out-offer is read twice within the `communicate`. This is important because, its value can change while the processors is executing `communicate` (i.e., between line 46 and 51).

After the processor completes an outstanding offer its attempts to make a new offer if its deque has at least one extra task that can be offered (processors work on the task at the bottom of the deque themselves). To make an offer, the processor picks a random target and checks if the target is idle. If so, it makes an offer as follows. First, it pops the task from the top of its deque and stores it into its `offered` field. Second, it stores in the `out-offer` the ID of the target processor and the round number of this target. Third, it sends the offer to the target processor by writing in the `in-offer` field of this target its own ID and the round number of the target.

Our synchronization-free greedy algorithm also naturally leads to a very simple algorithm that is lightly synchronized. We describe this algorithm in Appendix A.1, which eliminates the fence from the access path of local deque operations but uses a compare-and-swap for sharing tasks.

3.2 Correctness

We have proved that, when executed on a machine that implements the x86-TSO model, our load balancing algorithm is correct, in the sense that it does not drop nor duplicate any task. We present the full proof in Appendix A.2. For the proof, we formulate a global invariant that captures the fact that the set of all the ready tasks in the system is preserved through load balancing actions. We express this invariant in terms of the state of the shared memory, in terms of the contents of all the store buffers, and in terms of the current state of each processor (line number and value of the local variables).

In the x86-TSO model, the store buffers are represented as sequences of store operations. For reasoning about a program, however, it is more convenient to view the store buffer of one given processor as a set of *mini-buffers*, each of which being associated with one particular memory location, and to allow for the specification of a partial order between the store operations contained in different mini-buffers associated with the same processor. This approach simplifies the statement of the invariant, as we are able to independently reason about different memory locations, while nevertheless being able to capture the fact that particular store operations are performed before others, at the few places in the proof where this is needed.

The proof consists of three parts. First, we state the global invariant. Second, we show that this invariant is preserved by any *flush-transition*, which consists of taking out the oldest store operation from one arbitrary mini-buffer and executing this store operation to update the shared memory. This part of the proof accounts for the fact that the flush transitions can happen at any time during the execution. Third, we show that any atomic instruction performed by any given processor preserves our invariant. This shows that our algorithm runs correctly for any interleaving of the execution of the different processors.

3.3 Run-Time Analysis

While our algorithm is quite different than work stealing on the surface, its behavior and therefore its performance is quite similar to it. In this section, we state and discuss a run-time bound on the performance of our algorithm that is similar to the work-stealing bound. For the theorem, we assume that the `communicate` function is called with probability $\frac{1}{\delta}$ at every step of the execution by each processor independently. Under this assumption, the expected number of steps between two calls to `communicate` is δ steps. We further assume for simplicity that each node in the DAG has outdegree at most two. For the analysis, we consider a unit-time model, where each processor executes one instruction every unit of time.

Theorem 3.1 (Performance of greedy sharing) *Let T_1 and T_∞ be the work (sequential execution time) and the depth (minimal execution time with infinite processors) of a computation, and let H be the maximal number of forks (nodes with outdegree two) in any path of the computation. We prove that, for sufficiently large δ, the expected execution time for the computation with greedy sharing on P processors is*

$$\mathbb{E}[T] \;\leq\; 1.01\,\frac{T_1}{P} \;+\; 1.20\,T_\infty \;+\; 3.43\,\delta H.$$

The execution time is equal to Brent's bound $\frac{T_1}{P} + T_\infty$ plus the scheduling overhead, which takes the form $0.01\,\frac{T_1}{P} + 0.20\,T_\infty + 3.43\,\delta H$. The factor 0.01 comes from the fact that the calls to the function `communicate` incur some overhead, which we bound to be 1% for sufficiently large δ. The rest of the scheduling overhead consists of a term depending on T_∞, and another term depending on the product δH. This suggests that, when the maximal number of forks in a path is relatively small, then it is possible to increase the value of δ before observing any effect on the execution time regardless of the actualy depth of the computation.

We present a more general and precise version of this theorem and its proof in Appendix A.3 that considers the case for when δ is small. We also bound the expected number of task migrations, called S, as $\mathbb{E}[S] \;\leq\; 2P \cdot (T_\infty/\delta \;+\; 2.87\,H)$. This bound shows that the number of task migrations decreases slightly with δ. Compared to known bounds on work stealing, where there is no delay between steal attemps, our bound is tighter in its dependence on the depth, depending instead on H; the best known bound for work stealing as established by Tchiboukdjian et al [30] is $\mathbb{E}[T] \;\leq\; \frac{T_1}{P} + 3.65\,T_\infty$. We expect that our proof could also be adapted to establish a tighter bound for traditional work stealing.

Our proof reuses several ideas from known proofs of work stealing. The main novelty is the use of a carefully designed potential function that separates the potential due to the maximal length of a path from that due to the maximal number of forks in a path. Another novelty that simplifies the proof, compared with that of Tchiboukdjian, is the use of an induction over the set of possible configurations of the algorithm instead of an induction over the value of the potential.

We briefly describe an intuitive correspondence between greedy sharing and work stealing, which helps develop a high-level in understanding the performance of greedy sharing. This correspondence, however, is not meant for precise analysis but for simply developing intuition. Let's assume a variant of greedy sharing where every processor enters a communication phase every δ time steps and a variant of work stealing where it takes δ time steps to complete a steal attempt. Consider some processor A and the time at which the processor becomes idle. In work stealing, A goes steal a task from another processor and succeeds with probability P'/P where P' is the number of busy processors which have at least one task in their deque. This steal attempt takes δ steps and may be repeated if the steal attempt is not successful. Consider now work sharing. When A becomes idle it will simply wait for a task to be delivered. Within the next δ time steps, each of the P' busy processor will enter the communicate loop and will attempt to find and deliver a task to A. The probability that each processor will succeed in finding A is $1/P$ and thus the probability that at least one processor will find A is P'/P. Thus, within the next δ time steps, A will succeed in obtaining

work with probability P'/P, which is identical to the case with work stealing. The same argument applies to subsequent δ-intervals, because each outstanding offer will complete after δ time stamps (assuming $\delta > 2\epsilon$). The two algorithms therefore have the same parallel time complexity. Previous work [1] shows that the variant of work stealing where steal attempts take δ steps yields $O(T_1/P + \delta T_\infty)$ parallel time. This is not as tight as the bound that we obtain but asymptotically close when the depth of the computation and the maximal number of forks along a path are of similar order.

3.4 Handling Joins

The greedy-sharing algorithm (Section 3.1) performs load balancing without using any synchronization instructions. The algorithm, however, does not attempt to eliminate synchronization elsewhere in a parallel computation. Determining when a *join task*, a task with multiple incoming edges, becomes ready is another important source of synchronization. A naive way to determine when a join task becomes ready is to assign each join task a *join counter* that processors decrement via atomic read-write operations, such as fetch-and-add, as they execute the source of incoming edges. To reduce the overhead of synchronization, it is possible to replace an atomic decrement operations with a plain, non-atomic decrement operation if we can guarantee that all parents of a task execute on the same processor. The *clone-translation* technique achieves this [16], and eliminates all atomic decrement operations except for the tasks with parents that are stolen (migrated) and thus execute on different processors.

We outline an algorithm or protocol that we call *optimistic joins* that requires no synchronization operations when determining the readiness of join tasks. The algorithm is relatively straightforward; we therefore do not present a detailed description or a proof. The idea behind the algorithm is to use optimistically non-atomic decrement operations and rely on a message-passing protocol (via shared memory) to recover from data races. In optimistic joins, we assign each task an *owner* which is set to be the processor that creates the join and two join counters: an *owner-counter* and a *shared-counter*. Every time a processor executes a task, it signals each child of the task by decrementing the shared-counter. If the processor is also the owner of the child, then it also decrements the owner-counter. Whenever a processor (owner) finds the shared-counter (owner-counter) to be zero, it executes that task, setting the shared-counter to -1 (the shared counter of an executed task will always -1). After decrementing its counter, if a processor does not execute a join task, then it sends a message to the task's owner, acknowledging that it had executed its parent.

Since each task is executed exactly by one processor, there will be no race in setting the shared counter to zero: either there is no race, in which case we know that one and only one processor will see the shared-counter reaching zero; or there is a race, and the shared-counter will get stuck on a positive value. In optimistic joins, tasks whose parents are not stolen are guaranteed to be executed by the processor who owns the task. If a parent is stolen, then the shared-counter may not ever reach zero due to data races. Therefore, as an owner receives an acknowledgement message from another processor, it decrements the owner-counter and schedules the task when the owner-counter reaches zero (and if the task is not already executed, which can be determined by checking that the shared counter is positive).

Optimistic joins hinges on a method for exchanging messages without synchronization. The idea is to assign $P-1$ *producer-consumer buffers* (*PCB's*) to each processor, one reserved for each other processor. It is known that PCBs can be implemented without synchronization instructions (using either lists of circular buffers). The main limitations of this approach is that each processor has to poll $P-1$ buffers during the `communicate` function. One way to reduce the cost of polling is to use a simple one-cell cache to keep the index of the last message, and poll only one PCB, in each call to `communicate`. Since the messages are only needed when a parent is stolen, and when data race takes place, the cost of finding them will likely be small in practice.

3.5 Implementation and Experiments

We implemented a C++ library, called PASL, to provide a framework for evaluating our greedy-sharing algorithm on multicore machines. For handling join tasks, we use our optimistic-joins algorithm. For the purpose of comparison, we implemented the Chase-Lev algorithm [9], denoted `cl`, extended with the minimal required set of fences for ensuring correctness on x86-TSO [23]. We tuned our `cl` implementation until its performance got within a factor of two of the performance of Intel's TBB [21], which provides a highly-tuned, shared-deque implementation of work stealing for C++. An interesting property of our algorithm is its flexibility: greedy sharing can easily be adapted to support different load distribution strategies. As an example, we implement the "steal-half queues" strategy specialized for light-weight task-queue data structures.

3.6 Implementation

PASL creates for each worker thread a POSIX thread (i.e., `pthread`) to act as host for the worker on a particular core. We use the `hwloc` library to pin worker threads to particular cores [7] and always take care to prevent false sharing on critical scheduling data structures by using proper alignment. PASL does not require the programs to be written in a particular style, allowing instead any arbitrary computation DAG to be expressed directly [32]. For fork-join programs, however, PASL offers a special, optimized mode, which avoids the overheads of the general DAG-based model. We use the fork-join mode in one of our benchmarks to obtain an accurate estimate of the cost of memory fences in `cl`.

Our implementation of the greedy-sharing algorithm follows closely the pseudo-code presented in Section 3.1. To support the optimistic-joins algorithm, we check PCBs for incoming messages at every call to `acquire` and `communicate`. Observe that the pseudocode leaves it open how exactly the `communicate` function is called. One possibility is to use the OS signaling mechanism, which is often too expensive because triggering and handling of signals require invoking an OS routine. Because we assume that most tasks are small in front of δ, we can resort to a simpler approach, where each worker calls `communicate` only after the period of delay δ has passed since the previous call. In PASL, we implement this behavior by accessing the Time-Stamp Counter (TSC), which is a 64-bit register provided by the x86 architecture that counts the number of cycles since the processor began executing (or since the last integer overflow). The TSC is accurate enough for us to set delays at just tens of microseconds, yet cheap enough to not degrade performance noticeably. Otherwise, if we were not willing to assume that most tasks are small compared to δ, we could resort to *software polling*, where the compiler inserts into the program operations to check for incoming messages [14, 15], or to some form of hardware down-counter triggered interrupts. However, both approaches are relatively expensive and require care to amortize on large chunks of work.

3.7 Experiments

Test machine. Our test machine hosts four eight-core Intel Xeon X7550 [20] 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 2.6.32.50.2.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 GCC (v4.4.5) with the `-O2` option. Unless otherwise stated, the delay parameter δ in `gs` is set to the value which we found yields good performance on our machine, which is 50 microseconds.

Benchmarks. To determine how synchronization costs in the scheduler affect performance, we perform a limit study using a synthetic benchmark, which we call `array_write`. This benchmark allocates an

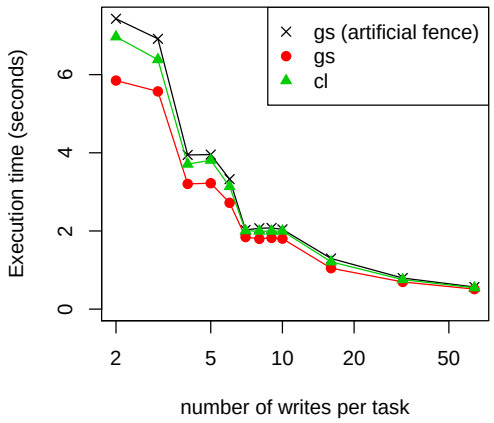

Figure 3: Effect of the memory fence on seven-core execution time of `array_write` (x-axis on log scale).

800MB array and then generates a total of four-billion writes to random cache lines in the array by recursively dividing the array into equal sized pieces. By varying the recursion cutoff, we control the size of leaves and the number of writes executed by each task. We ran this benchmark with PASL's fork-join-only mode to obtain a more precise measure of the synchronization overheads.

We follow the limit study with a comparison of the greedy-scheduling algorithm to existing the state-of-the-art Chase-Lev algorithm across three work loads with varying amounts of parallelism: scarce (`bellman_ford`), moderate (`cilksort`), and abundant (`matmul`). Our `bellman_ford` benchmark implements a parallel version of the Bellman-Ford algorithm for computing the single-source shortest paths in a weighted digraph. The graph is represented by an array of edges, where an edge consists of 32-bit integer source and destination IDs and a 32-bit integer weight. We parallelized the inner loop, which performs a relaxation step over all the edges. For the medium and large work loads, we chose two classic programs from the Cilk benchmark suite [31]. The `cilksort` benchmark implements a parallel mergesort on an array. The program applies a divide-and-conquer style that switches to a serial quicksort on an array whose size falls below a cutoff of our choosing. Merging is performed in parallel as well. The `matmul` benchmarks multiplies two dense matrices in place using a cache-efficient, divide-and-conquer algorithm.

Finally, we evaluate the flexibility of our algorithm in handling different task distribution strategies by considering the benchmark `dfs`, which implements a depth-first search of an arbitrary digraph. The digraph is represented as an array of edge lists, and the edge lists themselves are represented as a flat vectors of 32-bit ints. For input, we use a large graph, called cage15, that arises from DNA electrophoresis [33] and has been used recently to benchmark another parallel graph-connectivity algorithm [25]. The graph has 5,154,859 vertices, 99,199,551 edges, and its diameter is 50.

A summary of our benchmark input sizes and execution times can be found in Table 1.

Results: the limit study. Figure 3 shows the running time with our `array_write` benchmarks at different levels of granularity, which in turn controls the number of writes per leaf task. We ran this experiment on just seven of the cores in order to avoid NUMA effects, which would be observable if we ran on more cores. The results show that the Chase-Lev algorithm performs up to 15% worse than greedy-sharing for small tasks but catches up with the latter as the task sizes increase along the x-axis (exponential scale). To understand the effect of the extra fence that the Chase-Lev algorithm executes at every access to the deque, we inserted artificial fence operations into our algorithm in front of deque operations. Experiments show that this algorithm performs worse than both algorithms but closely tracks the performance of Chase-Lev otherwise. This experiment suggests that our greedy-sharing algorithm eliminates the fence without additional scheduling costs.

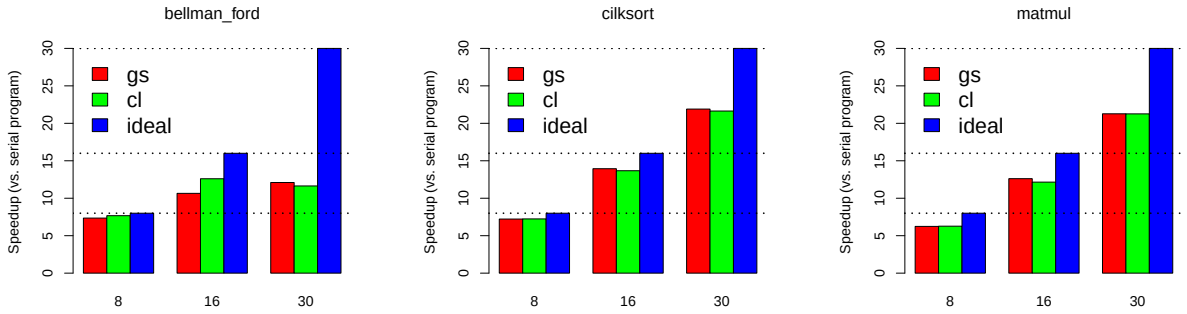

Figure 4: Comparing greedy sharing (`gs`) and Chase-Lev (`cl`) using eight, sixteen, and thirty cores.

Results: large granularity. Figure 4 shows a comparison between the performance of our greedy-sharing algorithm and the Chase-Lev algorithm. The main result is that our greedy sharing algorithm performs always within 5% of the Chase-Lev algorithm, except in one particular case where we have many forks in the critical path and where the results vary a lot for both algorithms depending on the cutoff size being used. For the two other programs with small number of forks in the critical path, the algorithms behavior is practically identical.

Results:steal-half deques and depth-first-search. Graph connectivity algorithms represent a challenging domain for parallel programming because of the irregularity of the computations and because tasks tend to be very small (e.g., tens of reads and writes per task). As demonstrated by our memory-fence experiment, at this granularity, memory-fence instructions in the scheduler degrade performance noticeably. One approach to this issue, called "idempotent work stealing", is a variant of work stealing where tasks are allowed to execute multiple times [26]. Thanks to this relaxation, fence instructions can be safely eliminated from the local-deque operations (though, fences are still required for remote deque operations). In spite of the performance advantage afforded by eliminating fences, this algorithm suffers from being confined to idempotent computations, which seems to be a small (though important) class of parallel algorithms. It is well known that a steal-half policy, where workers steal *half* of the nodes from the victim's local task pool, accelerates the performance of irregular graph computations [11]. The effort required to extend `gs` to support this policy was small thanks to its private-deque design. The results shows that our `dfs` benchmark achieves good scalability up to and including thirty cores. In particular, the speedup at 8, 16, and 30 cores are 5x, 8.55x and 11.58x respectively.

Results: sensitivity to delay. One concern is how sensitive our results are to different settings of δ. Recall from Section A.3 that, for programs having a small maximal number of forks (H), we can easily set δ to a very large value without degrading performance because the third term in the time bound of our algorithm, $2.68\delta H$, remains small. This prediction is confirmed by the results in Figure 5, which show execution time as a function of delay. From the first two plots we see that we can choose huge values for δ, such as five milliseconds, without noticeably degrading performance. For programs with large H, such as `bellman_ford`, we have to use a smaller value of δ, as we have a tradeoff between the cost of scheduling and the cost induced by total idle time. We can see from the right-most curve in Figure 5 that the performance degrades with a relatively small increase to δ. This behavior is expected because the parallel loop generates a few small tasks and by increasing the delay, we prevent idle processors from getting the available tasks.

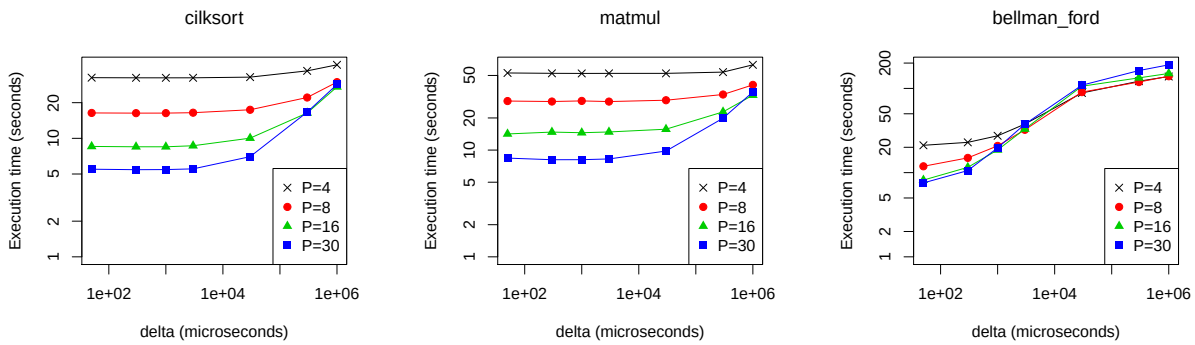

Figure 5: Execution time versus delay δ for various numbers of processors, on a log scale.

4 Related work

We have discussed closely related work in the introduction; here, we briefly review other more remotely related work.

Software support. Feeley was the first to study a work-stealing algorithm, which he dubbed MP, for weakly-consistent shared-memory architectures [14]. MP employs private deques and relies on message passing to migrate tasks between processors, but, unlike our greedy-sharing algorithm, MP requires synchronization operations to implement stealing. Furthermore, the study lacks proofs showing correctness and efficiency.

Flexibility. Feeley shows that, thanks to avoiding concurrency control on deque access, two features that appear in a number of langauges, namely dynamic scoping and first-class continuations, are much easier to implement in the private-deque model than in a shared-deque model. Manticore, an implementation of Parallel ML, employs a private-deque variant of work stealing [27]. The private-deque approach is useful for implementing lazy promotion, which is a scheme for lazily copying heap objects from processor-private heaps to a shared heap. In Section 3.5, we showed that we can implement graph-connectivity problems efficiently via a minor extension to our scheduling algorithm, whereas to do the same in the shared deque approach would require idempotent work stealing [26].

5 Conclusion

We presented techniques for performing load balancing on weakly consistent memory architectures—the total-store-order model specifically—without using synchronization operations. The crucial ideas behind our algorithm include the use of private deques for storing parallel tasks, the use of a simple message cell to provide for efficient and effective communication, and a protocol for work sharing that allows for data races without harming correctness. We prove that our algorithm is correct and bound the efficiency of P-processor executions with a bound similar to that of work stealing. Our implementation and preliminary experiments suggest that private deques simplify implementation and enable implementing other distribution policies, such as steal half policy, by giving processors exclusive access to their deques. Our experiments show that synchronization costs in the scheduler can degrade performance of fine-grain tasks and that our greedy-sharing algorithm can prevent such degradation. For moderate-to-large grain tasks, greedy sharing performs essentially the same as work stealing on current multicore architectures with relatively small number of cores. We leave it to future work to investigate whether our synchronization-free algorithm improves the scalability of load balancing with moderate-grain parallelism on massively parallel systems consisting of hundreds of cores.

References

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

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129. ACM Press, 1998.

[3] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 487–498, 2011.

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

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

[6] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[7] François Broquedis, Jérôme Clet Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a generic framework for managing hardware affinities in HPC applications. In *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

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

[9] 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] David Chase and Yossi Lev. Dynamic circular work-stealing deque. Technical report, Sun Microsystems, 2005.

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

[12] 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 (extended abstract). In *Proceedings of the 1985 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '85, pages 1–3. ACM, 1985.

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

[15] Marc Feeley. Polling efficiently on stock hardware. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 179–187, New York, NY, USA, 1993. ACM.

[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. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 280–289, New York, NY, USA, 2002. ACM.

[19] 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.

[20] 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)`.

[21] Intel. Intel threading building blocks. 2011.

[22] 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.

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

[24] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[25] Charles E. Leiserson and Tao B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 303–314, New York, NY, USA, 2010. ACM.

[26] 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.

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

[28] 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.

[29] 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.

[30] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part II*, volume 6507 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2010.

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

[32] Mike Rainey Umut A. Acar, Arthur Charguéraud. Efficient primitives for creating and scheduling parallel computations. In *Declarative Aspects of Multicore Programming*, January 2012. Appears in the short-paper track.

[33] A. van Heukelum, G. T. Barkema, and R. H. Bisseling. DNA electrophoresis studied with the cage model. *Journal of Compututational Physics*, 180:313–326, July 2002.

```
type state = task*
const state WAITING = (state) 0
const state INCOMING = (state) 1
state states[P] = { INCOMING, .. }

void acquire(scheduler* sched)
  task*& st = states[sched->id]
  st = WAITING
  while (st == WAITING || st == INCOMING)
    sched->wait_a_bit()
  sched->remote_push(st)
```

```
void communicate(scheduler* sched)
  if not sched->remote_has() then return
  int j = random ∈ {0, .., P-1}\{sched->id}
  if states[j] != WAITING then return
  bool s = compare_and_swap(& states[j],
                            WAITING, INCOMING)
  if not s then return
  states[j] = sched->remote_pop()
```

Figure 6: Sender initiated work stealing implemented using synchronization

A Appendix

A.1 Lightly Synchronized Greedy Sharing

We describe a variant of greedy sharing that utilizes synchronization operations for performing task migrations but still eliminates the need for synchronization from local deque operations. This algorithm and its correctness is very simple; the algorithm also delivers the performance very similar to the (synchronization-free) greedy sharing algorithm under the assumption that synchronization operations take constant time (an assumption that is not needed for the synchronization-free algorithm). More precisely, the synchronization overhead of this lightly-synchronized algorithm compared with our synchronization-free one is the execution of no more than two compare-and-swap instructions per offers.

Figure 6 shows the pseudo-code for this lightly synchronized algorithm. The algorithm uses a single state field for each processor. This field is intended to receives task pointers. When the processor becomes idle, it writes the special constant `WAITING` into the field, and then waits until it sees a task pointer being written in it. When a task arrives, the processor starts working on the task. When a busy processor invokes the communicate method, it picks a target at random and looks up its state. If the state is `WAITING`, then it tries to send a task to this target. To that end, the processor first compare-and-swaps the constant `INCOMING` into the state field of the target. If this compare-and-swap succeeds, then the processor sends the task at the top of its deque (using a conventional write) into the state field of the target. Otherwise, if the compare-and-swap instruction fails (meaning that the target has received a concurrent offer), then the processors returns to work. We here use a two-step approach to sending a task to avoid a redundant deque operation in the case where compare-and-swap eventually fails, but note that it would also be possible directly compare-and-swap the task pointer and take the task back in case of failure.

A.2 Proof of correctness

Before we present the proof, we summarize the most important invariants, in informal words. (1) The set of ready tasks is equal to the union of the content of the private deques, plus the content of the `offered` fields for the offers that have not been received yet. (An offer is received if the target processor has issued a write of the value RECEIVED to the out-offer field of the processor making the offer.) (2) The values written by a processor i in the in-offer field of a processor j refer to the ID i and to round numbers that increase through time, while always remaining less than or equal to the round of processor j. Moreover, if processor i has made an offer to j at round r, and if the round of j is exactly r, then the out-offer field of processor i contains the ID j and the round r. (3) If there is a pending store operation by a processor j to the out-offer field of processor i, meaning that the offer is accepted by j but this accept is not yet seen by i, then: (a) this store operation consists of writing the value RECEIVED, (b) there is no other pending store operation to the out-offer field of processor j, (c) the current value of this out-offer field in the shared memory is a stamp referring to the ID j and to a round that is strictly smaller than the current round of j (or will become so after the next instruction of processor j), and (d) processor j has incremented its round number only after writing to the out-offer field of the processor j from which it received the offer.

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, it is convenient to introduce two other pieces of notation. 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 $\bar{X}^i$. This value corresponds exactly to the value value returned when processor i reads at location x.

Variables For processor i, we let Q_i denote its private deque, R_i denote its round field, I_i denote its in-offer field, O_i denote its out-offer field, T_i denote its offered task. Moreover, we let L_i be the label of the next line that processor i will execute. We write $\mathcal{S}(i, r)$ a stamp associated with a processor i and a round r, that is, the result of calling the function `o_make` on i and r. Thereafter, we always use the index i for the sender, j for the receiver.

Overview We next summarize the most important invariants, in informal words. The details of the proof can be found in the appendix. (1) The set of ready tasks is equal to the union of the content of the private deques, plus the content of the `offered` fields for the offers that have not been received yet. (An offer is received if the target processor has issued a write of the value RECEIVED to the out-offer field of the processor making the offer.) (2) The values written by a processor i in the in-offer field of a processor j refer to the ID i and to round numbers that increase through time, while always remaining less than or equal to the round of processor j. Moreover, if processor i has made an offer to j at round r, and if the round of j is exactly r, then the out-offer field of processor i contains the ID j and the round r. (3) If there is a pending store operation by a processor j to the out-offer field of processor i, meaning that the offer is accepted by j but this accept is not yet seen by i, then: (a) this store operation consists of writing the value RECEIVED, (b) there is no other pending store operation to the out-offer field of processor j, (c) the current value of this out-offer field in the shared memory is a stamp referring to the ID j and to a round that is strictly smaller than the current round of j (or will become so after the next instruction of processor j), and (d) processor j has incremented its round number only after writing to the out-offer field of the processor j from which it received the offer.

Definitions V_i captures whether the task T_i should be considered as a valid ready task or not. $A_{i,j,r}$ describes whether there exists an active offer by processor i to processor j at round r. $B_{i,j,r}$ describes a state where j is currently receiving a task from i at round r. $C_{i,j}$ describes the particular state where j has just grabbed the task pointer from i but not yet updated its round number. $D_{i,j}$ indicates whether j has acknowledged reception of the task coming from i.

$$
\begin{array}{lcl}
V_i & \equiv & (O_i^i \neq \mathsf{RECEIVED} \wedge L_i \neq 54 \wedge \forall j \neq i.\, \bar{O}_i^j = \mathsf{nil} \wedge \neg C_{i,j}) \vee (L_i \in \{61, 62\}) \\
A_{i,j,r} & \equiv & i \neq j \ \wedge \ r = R_j^j \ \wedge \ (\mathcal{S}(i,r) \in \vec{I}_j^i \ \vee \ (O_i^i = \mathcal{S}(j,r) \wedge L_i \neq 63)) \\
B_{i,j,r} & \equiv & r = R_j \wedge \bar{R}_j^j = \mathsf{nil} \wedge i \neq j \wedge O_i = \mathcal{S}(j,r) \wedge (\forall j'.\, \bar{O}_i^{j'} = \mathsf{nil}) \wedge (\forall j' \neq j.\, \neg C_{i,j'}) \\
C_{i,j} & \equiv & L_j = 38 \text{ with variable i} = i \\
D_{i,j} & \equiv & i \neq j \wedge \bar{O}_j^i \neq \mathsf{nil}
\end{array}
$$

Invariant The invariant of the program is stated as the conjunction of the several invariants below.

$$
\begin{array}{lcl}
\mathcal{I}_N & \equiv & \forall j \neq i.\ \bar{R}_j^i = \mathsf{nil} \wedge \bar{T}_i^j = \mathsf{nil} \\
\mathcal{I}_T & \equiv & \text{The set of ready tasks is } \bigcup_i \left(Q_i \cup (\mathsf{if}\ V_i\ \mathsf{then}\ T_i^i\ \mathsf{else}\ \emptyset)\right) \\
\mathcal{I}_R & \equiv & \vec{R}_j^j \text{ is a list of stricly decreasing values} \\
\mathcal{I}_I & \equiv & I_j = \mathsf{WAITING} \vee (\exists ir.\ I_j = \mathcal{S}(i,r) \wedge i \neq j) \\
\mathcal{I}_W & \equiv & \bar{I}_j^j \text{ is a list whose elements are all equal to } \mathsf{WAITING} \\
\mathcal{I}_O & \equiv & O_i^i = \mathsf{RECEIVED} \vee \left(\exists jr.\ O_i^i = \mathcal{S}(j,r) \wedge i \neq j \wedge r \leq R_j\right) \\
\mathcal{I}_P & \equiv & \forall j \neq i.\ \exists n \geq 0.\ \exists r_1 \ldots r_n.\ \bar{I}_j^i = \mathcal{S}(i,r_1) :: \ldots :: \mathcal{S}(i,r_n) :: \mathsf{nil} \\
 & & \wedge (R_j \geq r_1 > \ldots > r_n) \\
 & & \wedge\ (\forall r.\ I_j = \mathcal{S}(i,r) \Rightarrow (n = 0 \wedge R_j \geq r) \vee (n > 0 \wedge r_n > r)) \\
\mathcal{I}_A & \equiv & A_{i,j,r} \wedge L_j \neq 39 \Rightarrow \exists t. \\
 & & O_i^i = \mathcal{S}(j,r) \\
 & & \wedge\text{the last write, if any, in } \bar{O}_i^i \text{ occured at time } < t, \text{ and similarly for } \bar{T}_i^i \\
 & & \wedge \left((\bar{I}_j^i \neq \mathsf{nil} \wedge \text{ last write in } \bar{I}_j^i \text{ at time t}) \vee (\bar{I}_j^i = \mathsf{nil} \wedge t = 0)\right) \\
\mathcal{I}_D & \equiv & D_{i,j} \Rightarrow O_i = \mathcal{S}(j,r) \wedge \bar{O}_i^i = \mathsf{nil} \wedge \exists t. \\
 & & \bar{O}_i^j = \mathsf{RECEIVED} :: \mathsf{nil}, \text{ with this write occuring at a time } < t \\
 & & \wedge \begin{pmatrix} (r = R_j \wedge \bar{R}_j^j = \mathsf{nil} \wedge L_j = 39 \text{ with variable i = } i) \\ \vee\ (\bar{R}_j^j \text{ contains a write of } (r+1) \text{ requested at time } t) \end{pmatrix}
\end{array}
$$

Invariant, continued:

$$
\begin{array}{lcl}
\mathcal{I}_{34} & \equiv & L_j = 34 \Rightarrow r = R_j^j \Rightarrow B_{i,j,r} \\
\mathcal{I}_{37} & \equiv & L_j \in [37,38] \Rightarrow B_{i,j,r} \\
\mathcal{I}_{39} & \equiv & L_j = 39 \Rightarrow r = R_j \wedge \bar{R}_j^j = \mathsf{nil} \\
\mathcal{I}_{46} & \equiv & L_i \in [46,54] \Rightarrow i \neq j \wedge r \leq R_j \\
\mathcal{I}_{49} & \equiv & L_i = 49 \Rightarrow \bar{I}_j^i = \mathsf{nil} \\
\mathcal{I}_{51} & \equiv & L_i \in [51,54] \Rightarrow r < R_j \\
\mathcal{I}_{53} & \equiv & L_i \in [53,54] \Rightarrow O_i^i \neq \mathsf{RECEIVED} \wedge (\forall j \neq i.\ \bar{O}_i^j = \mathsf{nil}) \\
\mathcal{I}_{57} & \equiv & L_i \in [57,62] \Rightarrow O_i^i = \mathsf{RECEIVED} \wedge (\forall j \neq i.\ \bar{O}_i^j = \mathsf{nil}) \\
\mathcal{I}_{58} & \equiv & L_i \in [58,63] \Rightarrow i \neq j \\
\mathcal{I}_{60} & \equiv & L_i \in [60,63] \Rightarrow \bar{I}_j^i = \mathsf{nil} \\
\mathcal{I}_{62} & \equiv & L_i = 62 \Rightarrow r \leq R_j \\
\mathcal{I}_{63} & \equiv & L_i = 63 \Rightarrow O_i^i = \mathcal{S}(j,r) \wedge (\forall j \neq i.\ \bar{O}_i^j = \mathsf{nil})
\end{array}
$$

Stability We need to show that the invariant is stable in the sense that whenever a writes is pulled out from a write buffer and applied to main memory then the invariant is stable. Note that when an invariant states that a buffer is empty, then the invariant is immediately stable. The non-trivial stability proofs are the following.

- $\mathcal{I}_R$: By $\mathcal{I}_N$, a new value of R_j can only come from $\bar{R}_j^j$. Therefore, the sequence $\vec{R}_j^j$ remains decreasing.
- All assertions of the form $r \leq R_j$ are stable because by $\mathcal{I}_R$, the content of R_j can only increase.
- $\mathcal{I}_I$ and $\mathcal{I}_P$. If the new value for I_j comes from the buffer of j, then by $\mathcal{I}_W$ it is $\mathsf{WAITING}$. This satisfies $\mathcal{I}_I$ and $\mathcal{I}_P$. Otherwise, it comes from the buffer of a processor $i \neq j$. By $\mathcal{I}_P$, the new value is of the form $\mathcal{S}(i,r)$ with r being less than R_j and less than the previous round number in the same buffer. Therefore, $\mathcal{I}_I$ and $\mathcal{I}_P$ are both satisfied.
- $\mathcal{I}_D$, with respect to action on $\bar{R}_j^j$. Assume that $D_{i,j}$ holds, otherwise there is nothing to prove. $\bar{O}_i^j$ contains a write that occurs before the write of $(r+1)$ in $\bar{R}_j^j$. So, this latter write cannot make it to main memory otherwise we would have a contradiction on the content on $\bar{O}_i^j$.

– $\mathcal{I}_T$, with respect to writes to T_i. By $\mathcal{I}_N$, a new value in T_i can only come from $\bar{T}_i^i$. Therefore, T_i^i is stable.

– $\mathcal{I}_W$. The assertion of this invariant is clearly stable.

– $B_{i,j,r}$ remains true if it is true, because it forces the buffers to be empty. So, $\mathcal{I}_{34}$ and $\mathcal{I}_{37}$ are stable.

– $\mathcal{I}_A$ with respect to writes to $I_{j'}$. Assume $A_{i,j,r}$ and $L_j \neq$, otherwise $\mathcal{I}_A$ trivially holds after pushing a write. If $j' \neq j$, then the conclusion of $\mathcal{I}_A$ remains unchanged. Otherwise, if the write comes from the buffer of j, then we have $\bar{I}_j^i \neq \mathsf{nil}$. If the buffer contains more than one element, the invariant remains true. Otherwise, if the last write of the buffer makes it to main memory, then it must be the case that $\bar{O}_i^i$ and $\bar{T}_i^i$ are already empty. The invariant remains satisfied by taking $t = 0$.

– $\mathcal{I}_O$ and $\mathcal{I}_D$ and $\mathcal{I}_A$ and $\mathcal{I}_T$ and $\mathcal{I}_{53}$ and $\mathcal{I}_{57}$, with respect to a write to O_i by processor j'. This is the most interesting part. First, we consider the case of a write to O_i by processor i itself. In this case, the value of O_i^i is unchanged. The only invariant mentio ning $\bar{O}_i^i$ explicitly is $\mathcal{I}_D$. There are two cases. If $D_{i,j}$ is false, we are done. Otherwise, we have $\bar{O}_i^i = \mathsf{nil}$, so there cannot be any write from this buffer. This concludes our first case.

  Second, we consider the case of a write to O_i by a processor $j \neq i$. If such a write occurs, it must be that $\bar{O}_i^j \neq \mathsf{nil}$. Therefore, just before this point, $D_{i,j}$ holds. By $\mathcal{I}_D$, we have $\bar{O}_i^j = \mathsf{RECEIVED} :: \mathsf{nil}$ and $\bar{O}_i^i = \mathsf{nil}$ and $O_i = \mathcal{S}(j, r)$ and $r = R_j$. We can show at this point at V_i is false. Indeed, $\bar{O}_i^j \neq \mathsf{nil}$ with $j \neq i$. After the write, V_i is still false. Indeed, O_i^i is then equal to $\mathsf{RECEIVED}$. This shows the stability of $\mathcal{I}_T$. Moreover, after the write, $\mathcal{I}_O$ holds because of O_i^i being now equal to $\mathsf{RECEIVED}$. $\mathcal{I}_D$ holds because $D_{i,j}$ has become false due to $\bar{O}_i^j$ becoming empty. $\mathcal{I}_A$ holds because $A_{i,j,r}$ has become false due to O_i^i becoming $\mathsf{RECEIVED}$. $\mathcal{I}_{53}$ and $\mathcal{I}_{57}$ are preserved because they ensure that $\bar{O}_i^j = \mathsf{nil}$, contradiction our hypothesis of a write being issued from this buffer.

Verification of the acquire function

– Line 32. The write of $\mathsf{WAITING}$ in $\bar{I}_j^j$ preserves $\mathcal{I}_I$ and $\mathcal{I}_W$.

– Line 34. The value of inc is equal to I_j^j. Because we just exited the loop, $I_j^j \neq \mathsf{WAITING}$. By $\mathcal{I}_W$, $\bar{I}_j^j$ can only contain values equal to $\mathsf{WAITING}$. So, we we must have $\bar{I}_j^j = \mathsf{nil}$ and $I_j \neq \mathsf{WAITING}$. By $\mathcal{I}_I$, we know thus that $I_j = \mathcal{S}(i, r)$ with $i \neq j$, where r is the value of `o_rnd (o)` and i is the value of `o_id (o)`. We need to prove $\mathcal{I}_{34}$. So, we assume $r = R_j^j$ and we establish $B_{i,j,r}$.

  First, we prove $r = R_j = R_j^j$. By $\mathcal{I}_R$, we know that $R_j^j \geq R_j$. So, $r \geq R_j$. By $\mathcal{I}_P$, because we have $I_j = \mathcal{S}(i, r)$ and $r \geq R_j$, it must be the case that $r = R_j$, with $\bar{I}_j^i = \mathsf{nil}$.

  Before we can derive the other components of $B_{i,j,r}$, we first need to establish that $A_{i,j,r}$ holds. Indeed, we have $i \neq j$ and $r = R_j^j$ and $\mathcal{S}(i, r) \in \vec{I}_j^i$. From $A_{i,j,r}$ and $L_j \neq 38$, we can exploit the conclusions of $\mathcal{I}_A$. As shown earlier on, we have $\bar{I}_j^i = \mathsf{nil}$. So from $\mathcal{I}_A$ we can derive $O_i^i = \mathcal{S}(j, r)$ and $\bar{O}_i^i = \mathsf{nil}$ and $\bar{T}_i^i = \mathsf{nil}$.

  We are now ready to establish the other components of $B_{i,j,r}$. We have already derived $i \neq j$. From $O_i^i = \mathcal{S}(j, r)$ and $\bar{O}_i^i = \mathsf{nil}$, we derive $O_i = \mathcal{S}(j, r)$. To prove $\forall j'.\ \bar{O}_i^{j'} = \mathsf{nil}$ when $j' \neq i$, we assume the contrary holds. We would have $D_{i,j'}$ true. By $\mathcal{I}_D$, this would entail $O_i = \mathcal{S}(j', r)$. Because $O_i^i = \mathcal{S}(j, r)$, we would therefore have $j' = j$. Moreover, by $\mathcal{I}_D$ and using the $r = R_j^j$, we get $L_j = \mathsf{Acq}_{39}$, which contradicts the fact that we are currently on line 7. Finally, to prove $\forall j' \neq j.\ \neg C_{i,j}$, we assume the contrary holds. By $\mathcal{I}_{37}$, we have $O_i^i = \mathcal{S}(j', r)$. Because $O_i^i = \mathcal{S}(j', r)$, this would give $j' = j$, which contradicts $j' \neq j$.

– Line 34. If the test on line 34 fails, we're done. Otherwise, we have $r = R_j^j$. Therefore we can exploit $\mathcal{I}_{34}$, which gives us immediately the conclusion of $\mathcal{I}_{37}$.

– Line 37. At this line, a task pointer is moved from processor i to j. In order to preserve $\mathcal{I}_T$, we have to prove that V_i evolves from true to false. Indeed, V_i becomes false because $C_{i,j}$ becomes true, as we end up with $L_j = 38$ with the source variable equal to i. It thus remains to show that V_i was true. For that, we use the hypotheses from $B_{i,j,r}$, which is the conclusion of $\mathcal{I}_{37}$. First, $O_i^i \neq \mathsf{RECEIVED}$ holds because $O_i^i = \mathcal{S}(j, r)$. Second, $L_i \neq 54$ because otherwise by $\mathcal{I}_{51}$ we would have $r < R_j$. Third, ($\forall j' \neq i.\ \bar{O}_i^{j'} = \mathsf{nil}$) follows from $B_{i,j,r}$. The fact $\bar{O}_i^{j'} = \mathsf{nil}$ follows from $B_{i,j,r}$ in the case $j' \neq j$. In the case $j' = j$ it follows from the fact that just before executing line 37, we have $L_j = 37$ so $L_j \neq 38$, hence $C_{i,j}$ is false.

– Line 38. We are writing in $\bar{O}_i^j$. At this point, $D_{i,j}$ becomes true. We thus need to prove the conclusion of $\mathcal{I}_D$ hold. For that, we use the fact from $B_{i,j,r}$, which is the conclusion of $\mathcal{I}_{37}$. First, we still have $O_i^i = \mathcal{S}(j,r)$, because the write that we are doing on this line has not yet made it to O_i. Second, $\bar{O}_i^i = \mathsf{nil}$ follows from $B_{i,j,r}$. Third, $\bar{O}_i^j$ indeed contains a single write of the value RECEIVED, because according to $B_{i,j,r}$, we had $\bar{O}_i^{j'} = \mathsf{nil}$ for any j' just before line 38. Fourth, we have $r = R_j$ and $\bar{R}_j^j = \mathsf{nil}$ follow from $B_{i,j,r}$. Fifth, we have $L_j = 39$ with source i because the next line to execute is line 39. Besides, $\mathcal{I}_{39}$ holds because it directly follows from $B_{i,j,r}$.

– Line 39. This line actually contains two atomic instructions: a read and a write. Because only processor j can write in the variable `round[j]`, we know that the instruction `round[j]++` will have the effect of incrementing the value R_j^j by one, no matter what the other processor do between the read and the write. Before the write to the round number, by $\mathcal{I}_{39}$ we have $r = R_j$ and $\bar{R}_j^j = \mathsf{nil}$. So, we are writing in $\bar{R}_j^j$ the value $R_j + 1$. Thus, $\mathcal{I}_R$ is satisfied. $\mathcal{I}_N$ is clearly preserved. For $\mathcal{I}_D$, there are two cases. If $D_{i,j}$ is false (meaning that the write from line 38 has already made it to the shared memory), then there is nothing to prove. Otherwise, if $D_{i,j}$ is true, then we can assume the conclusions of $\mathcal{I}_D$. To prove that these conclusions still hold, it suffices to observe that the write of $r + 1$ being pushed to $\bar{R}_j^j$ is being requested after the write of RECEIVED into $\bar{O}_i^j$.

Verification of the communicate function

– Line 44. If the test succeeds, then $O_i^i \neq$ RECEIVED. In this case, we need to prove the conclusion of $\mathcal{I}_{46}$, which is $i \neq j \wedge r \leq R_j$. This follows immediately from $\mathcal{I}_P$. Otherwise, if the test fails, then $O_i^i =$ RECEIVED. In this case, we need to prove the conclusion of $\mathcal{I}_{57}$. This amounts to proving $\forall j \neq i.\ \bar{O}_i^j = \mathsf{nil}$. Assume the contrary, that is, $\bar{O}_i^j \neq \mathsf{nil}$ for some $j \neq i$. Then $D_{i,j}$ would be true. By $\mathcal{I}_D$, we would have $O_i = \mathcal{S}(j,r)$ and $\bar{O}_i^i = \mathsf{nil}$, hence $O_i^i = \mathcal{S}(j,r)$ which contradicts the test.

– Line 46. By $\mathcal{I}_{46}$, we can assume $i \neq j$ and $r \leq R_j$. The test of line 46 is comparing r with the value R_j^i. By $\mathcal{I}_R$, we have $\bar{R}_j^i = \mathsf{nil}$, therefore $R_j^i = R_j$. So, if the test succeeds, then $r = R_j$. This fact is not used by the correctness proof but only for establishing progress. Otherwise, if the test fails, then $r \neq R_j$. This entails the conclusion of $\mathcal{I}_{51}$, that is, $r < R_j$.

– Line 48. If the test of line 48 succeeds, then $I_j^i =$ WAITING. In this case we need to show $\mathcal{I}_{49}$, which is $\bar{I}_j^i = \mathsf{nil}$. This follows from $\mathcal{I}_P$, which states that $\bar{I}_j^i$ can only contain stamps. Otherwise, if the test fails, then the function returns.

– Line 49. By $\mathcal{I}_{48}$ and $\mathcal{I}_{49}$, we can assume $r \leq R_j$ and $\bar{I}_j^i = \mathsf{nil}$. On line 49, we write $\mathcal{S}(j,r)$ into $\bar{I}_j^i$. Invariant $\mathcal{I}_P$ is preserved because $\bar{I}_j^i$ is now a singleton list made of $\mathcal{S}(j,r)$ with $r \leq R_j$. For $\mathcal{I}_A$, there are two cases. If $A_{i,j,r}$ is false, there is nothing to prove. Otherwise, since $\bar{I}_j^i = \mathsf{nil}$, we can assume $\bar{O}_i^j = \mathcal{S}(j,r)$. The conclusions of $\mathcal{I}_A$ are then satisfied.

– Line 51. By $\mathcal{I}_{51}$, we can assume $r < R_j$. If the test of line 51 failw, then we are done. Otherwise, if it succeeds, then we learn $O_i^i \neq$ RECEIVED. In this case, we need to establish the conclusion of $\mathcal{I}_{53}$, which are $O_i^i \neq$ RECEIVED and $(\forall j \neq i.\ \bar{O}_i^j = \mathsf{nil})$. For the later, assume the contrary. Then there would exists some $j \neq i$ such that $\bar{O}_i^j \neq \mathsf{nil}$, and we would have $D_{i,j}$ true. The last conclusion of $\mathcal{I}_D$ would contradict $r < R_j$. Indeed, either $r = R_j$, which is impossible; or $\bar{R}_j^j$ contains a write of $r + 1$, which would imply by $\mathcal{I}_R$ that $R_j < r + 1$, and this is also impossible.

– Line 53. We are transferring a task back from T_i^i into the deque Q_i. For that, we need to show that V_i evolves from true to false. First, we prove that V_i was true before executing line 53. By $\mathcal{I}_{51}$ and $\mathcal{I}_{53}$, we have $r < R_j$ and $O_i^i \neq$ RECEIVED and $(\forall j \neq i.\ \bar{O}_i^j = \mathsf{nil})$. Moreover, we have $L_i \neq 54$. To prove $\neg C_{i,j}$, assume the contrary. By $\mathcal{I}_{37}$, we would have $B_{i,j,r}$ for some r, which implies that $r = R_j$, contradicting our hypothesis. This ends our proof that V_i was true. As the next line is 54, we'll have $L_i = 54$ and $L_i \notin \{61, 62\}$, therefore V_i becomes false after the execution of line 53. It follows that $\mathcal{I}_T$ is preserved.

– Line 54. By $\mathcal{I}_{51}$ and $\mathcal{I}_{53}$, we have $r < R_j$ and $O_i^i \neq$ RECEIVED and $(\forall j \neq i.\ \bar{O}_i^j = \mathsf{nil})$. We are writing RECEIVED into $\bar{O}_i^i$. Before this write, V_i was false, due to $L_i = 54$ and $L_i \notin 61, 62$. After this write, we can show that V_i remains false. Indeed, O_i^i becomes equal to RECEIVED. So $\mathcal{I}_T$ is preserved. Invariant $\mathcal{I}_O$ holds due

to $O_i^i = \mathsf{RECEIVED}$. Invariant $\mathcal{I}_A$ holds because $A_{i,j,r}$ is false due to $r < R_j$. Invariant $\mathcal{I}_D$ holds due to $D_{i,j}$ being false by assumption.

- Line 58. This ensures $j \neq i$, satisfying $\mathcal{I}_{58}$.
- Line 59. The failure of the test ensures $I_j^i = \mathsf{WAITING}$, which by $\mathcal{I}_P$ implies $\bar{I}_j^i = \mathsf{nil}$. Therefore, $\mathcal{I}_{60}$ holds.
- Line 60. By $\mathcal{I}_{57}$ and $\mathcal{I}_{58}$ and $\mathcal{I}_{60}$, we can assume $O_i^i = \mathsf{RECEIVED}$ and $(\forall j \neq i.\ \bar{O}_i^j = \mathsf{nil})$ and $j \neq i$ and $\bar{I}_j^i = \mathsf{nil}$. We need to show that V_i goes from false to true. It was previously false because $O_i^i = \mathsf{RECEIVED}$ and L_i was not in $\{61, 62\}$. It now becomes true because L_i becomes 61. Therefore, $\mathcal{I}_T$ is preserved.
- Line 61. $r = R_j^i$. By $\mathcal{I}_R$, $\bar{R}_j^i$ is empty so $R_j^i = R_j$. This establishes $\mathcal{I}_{62}$.
- Line 62. By $\mathcal{I}_{57}$ and $\mathcal{I}_{58}$ and $\mathcal{I}_{60}$ and $\mathcal{I}_{62}$, we can assume $O_i^i = \mathsf{RECEIVED}$ and $(\forall j \neq i.\ \bar{O}_i^j = \mathsf{nil})$ and $j \neq i$ and $\bar{I}_j^i = \mathsf{nil}$ and $r \leq R_j$. We are writing $\mathcal{S}(j, r)$ into $\bar{O}_i^i$, so now $O_i^i = \mathcal{S}(j, r)$. $\mathcal{I}_O$ is satisfied. $\mathcal{I}_D$ holds because $D_{i,j}$ is false. For $\mathcal{I}_A$, assume $A_{i,j',r'}$ is true after the write $O_i^i = \mathcal{S}(j, r)$. There are two cases. If $A_{i,j',r'}$ was already true before this write, then we had $O_i^i = \mathcal{S}(j', r')$ which contradicts the assumption $O_i^i = \mathsf{RECEIVED}$. Otherwise, if $A_{i,j',r'}$ was false before, then it means that $\mathcal{S}(i, r) \notin \bar{I}_j^i$. So, after the execution of line 62, $A_{i,j',r'}$ is still false, due to $L_i = 63$. We can check the conclusions of $\mathcal{I}_{63}$, which are $O_i^i = \mathcal{S}(j, r)$ and $(\forall j \neq i.\ \bar{O}_i^j = \mathsf{nil})$.
- Line 63. By $\mathcal{I}_{58}$ and $\mathcal{I}_{60}$ and $\mathcal{I}_{63}$, we can assume $j \neq i$ and $\bar{I}_j^i = \mathsf{nil}$ and $r \leq R_j$ and $O_i^i = \mathcal{S}(j, r)$ and $(\forall j \neq i.\ \bar{O}_i^j = \mathsf{nil})$. We are writing $\mathcal{S}(i, r)$ into $\bar{I}_j^i$. We can check that $\mathcal{I}_P$ is satisfied, because $\mathcal{S}(i, r)$ is the only item in $\bar{I}_j^i$ and $r \leq R_j$. It remains to show $\mathcal{I}_A$. Assume $A_{i,j',r'}$ holds and $L_j \neq 39$. Due to $O_i^i = \mathcal{S}(j, r)$, we must have $j' = j$ and $r' = r$. We can establish the conclusions of $\mathcal{I}_A$. $O_i^i = \mathcal{S}(j, r)$ holds and $\bar{I}_j^i \neq \mathsf{nil}$ after line 63 executes. Moreover, the last writes in $\bar{O}_i^i$ and $\bar{T}_i^i$ occured before.

A.3 Proof of efficiency

P	number of processors
T	execution time with P processors
T_1	execution time with 1 processors
T_∞	execution time with ∞ processors (critical path)
H	maximal number of forks in any path of the computation DAG
δ	delay between two steal/deal phases
ϵ	upper bound on the delay required for a write to reach main memory
χ	bound on the execution time of the communicate function
I	total number of tokens in the idle bucket
J	total number of tokens in the transition bucket
K	total number of tokens in the communication bucket
S	total number of task migrations
s	a configuration of the algorithm
s_0	initial configuration of the algorithm
$\alpha(s)$	number of idle processors in configuration s
$\rho(s, s')$	probability to make a transition from s to s'
u	a node from the transformed computation DAG
$g(u)$	the length of the critical path starting from u
$f(u)$	the maximal number of forks in any path from u
$h(u, Q)$	1 if u is at the bottom of deque Q, 0 otherwise
Q	a deque, including the currently-running task at the bottom
$G(Q)$	maximum value of $g(u)$ for any u in the deque Q
$F(Q)$	maximum value of $f(u) + x(u)$ for any u in the deque Q
$\Phi(Q)$	potential of a deque Q
$Q_i(s)$	deque of processor i in configuration s
$\Phi(s)$	total potential of configuration s
$I(s)$	expected number of idle tokens generated from configuration s
κ	the constant that minimizes $\frac{\kappa}{1-2e^{-\kappa}}$; we have $\kappa \approx 1.68$
λ	a shorthand for $(1 - 2e^{-\kappa})^{-1}$; this gives $\lambda \approx 1.59$
c	a shorthand for $(1 - \frac{2\epsilon}{\delta})^{-1}$, which is greater than 1
ν	a shorthand for $\frac{1/\delta}{1-\mathrm{e}^{-1/\delta}}$, which is just slightly greater than 1
μ	a shorthand for $\frac{1}{\delta c \lambda}$
r	a shorthand for $\frac{\nu(P-1)}{\mu}$, which is equal to $\nu\delta c\lambda(P-1)$

Figure 7: Variables used in the proof of efficiency

Assumptions for the complexity proof We assume that there exists some upper bound ϵ on the delay required for a write to reach main memory. We moreover assume that $\delta > 2\epsilon$. We assume that the cost of calling the communicate function does not exceed a constant χ. We moreover assume that $\delta > \chi$. We assume that a buzy processor calls the communicate function with probability $\frac{1}{\delta}$, where δ is expressed in time units.

Discretization and tokens We encode a weighted computation DAG into a unit-cost DAG, so that, at each time step, each processor is executing zero or one node of the transformed computation DAG. We analyse the execution time using tokens: at each time step, each processor places one token in a bucket. A busy processors contributes to the *work bucket*. The content of this bucket at the end of the execution is equal to T_1. A processor whose in-offer field has the value WAITING in shared memory contributes to the *idle bucket*. We call I the final content of this bucket. A processor that just got idle and whose in-offer field does not yet contain WAITING in the shared memory contributes to the *transition bucket*, whose final content is called J. A processor that is in the process of calling the communicate function to make offer contributes to the *communication bucket*, whose final content is called K. Since

the cummulated execution time is PT, we have:

$$T \;=\; \frac{T_1 + I + J + K}{P} \tag{1}$$

Configuration DAG The key challenge in the proof consists in bounding the exected total number of tokens in the idle bucket, that is, $\mathbb{E}[I]$. We are going to introduce a potential function Φ that decreases during the execution of the program. Let s denote a particular configuration of the work stealing algorithm, that is, a description of the content of the deque of each processor. In the initial configuration s_0, the initial task is placed on the deque of one processor, while all the other processors have empty deques. Let $I(s)$ be a random variable denoting the number of tokens placed in the idle bucket starting from a configuration s. We are going to prove, by induction on the set of all possible configurations that $\mathbb{E}[I(s)] \leq r \ln \Phi(s)$, for some appropriate constant r. By applying this result to the initial configuration, we will derive a bound on $\mathbb{E}[I]$.

At each time step, the work stealing algorithm makes a transition from a configuration to another. The *configuration DAG* is the graph whose nodes are the configurations and whose edges are the transitions between configurations. There is indeed no cycle in this graph because the total amount of work decreases strictly at each time step. Moreover, let $\rho(s, s')$ denote the probability that the algorithm reaches the configuration s' at the next time step, knowing that it is currently in the configuration s. For any configuration s (except the terminal one), the probabilities add up to one, i.e., $\sum_{s'} \rho(s, s') = 1$.

Let $\alpha(s)$ denote the number of idle processors in the configuration s, that is, the number of processors that have an empty deque and the value WAITING visible in shared memory for their in-offer field. The number $\alpha(s)$ thus gives the number of tokens contributing to the idle bucket at the time step associated with the execution of the configuration s. Observe that $\alpha(s) \leq P - 1$, because at any time step there is at least one procesor working (otherwise the program has already terminated). The expected total number of tokens placed in the idle bucket starting from the configuration s, namely $I(s)$, is equal to $\alpha(s)$ plus the expected number of tokens produced after this step, that is, the average the values $\mathbb{E}[I(s')]$ for all s', weighted by the probability of having a transition from s to s'. Formally, this property is expressed as follows.

$$\mathbb{E}[I(s)] \;=\; \alpha(s) \,+\, \sum_{s'} \rho(s, s') \cdot \mathbb{E}[I(s')] \tag{2}$$

Potential function The definition of the potential function involves two auxiliary definitions. The *depth potential* of a node u, written $g(u)$, is defined as the total depth, called T_∞, minus the minimal length of a path that reaches the node u from the root. The *fork potential* of a node u, written $f(u)$, is defined as the maximal number of forks in a path of the entire computation DAG, called H, minus the minimal number of fork nodes in a path that reaches the node u from the root. In a binary fork-join program, a fork node is a node with out-degree 2. Let Q be a deque, that is, the content of all the ready tasks plus the task that is currently executing on the processor considered, if any. If the deque Q is empty, we define its potential $\Phi(Q)$ to be zero. Otherwise, we define it as:

$$\Phi(Q) \;\equiv\; \mathrm{e}^{\mu G(Q) + \kappa F(Q)} \quad \text{with} \quad G(Q) \equiv \max_{u \in Q} \; g(u) \quad \text{and} \quad F(Q) \equiv \max_{u \in Q} \; f(u) + h(u, Q)$$

where $h(u, Q)$ is equal to 0 if u is at the bottom of Q (that is, if u is running or about to run) and to 1 otherwise. We take $\kappa = 1.68$. We will explain later on the choice of this particular value. We define $\mu = \frac{1}{\delta c \lambda}$, where λ and c are constants that will be defined later on. We introduce the constant ν in order to get a little bit of slack required for a crucial inequality to hold. We then define the potential of a configuration as the sum of the potential of each of the deques of this configuration. More precisely, we let $Q_i(s)$ denote the deque of processor i in the configuration s and we define the potential as follows.

$$\Phi(s) \;\equiv\; \sum_i \Phi(Q_i(s))$$

The performance of the work stealing algorithm critically relies on the fact that the task with bigger potential sits at the top of the deque. If u and u' are two nodes contained in a same deque Q, and if u is located above u', then $g(u') \leq g(u) - 1$ and $f(u') \leq f(u) - 1$. This invariant is maintained because processors work on the bottom of their deque. A detailed proof of the deque invariant can be found in Blumofe and Leiserson's original paper [].

Probability of a successful deal Consider a given busy processor at a given time step. The probability of this processor executing the communicate function at this time step is by assumption $\frac{1}{\delta}$. Moreover, thanks to the assumption $\delta > 2\epsilon$, if the processor has made an offer at the previous call to the communicate function, then it got a response for it already. Indeed, it takes at most a time ϵ for the offer to reach the target and at most ϵ for the explicit accept or the implicit reject to come back. Therefore, the processor is able to initiate a new deal attempt. Because there are $\alpha(s)$ processors declared idle, the probability for the busy processor to find one of them at random is equal to $\frac{\alpha(s)}{P-1}$. So, the probability to try to make an offer is $\frac{\alpha(s)}{(P-1)\delta}$.

We now need to bound the probability for an offer to be accepted. A processor makes an offer only if it does not see an existing offer for its target. An offer fails only if a race occurs. There are less than $(P-1)$ other processors to race with. Consider a particular other processor. A race with it can occur only if the distance in time between the writes of their offer is less than ϵ. This requires the other processor to enter its communicate function within plus or minus ϵ time, which happens with probability $\frac{2\epsilon}{\delta}$. Moreover, a race requires the other processor to pick the same target, which happens with probability $\frac{1}{P-1}$. Multiplying the (independent) probabilities together, the probability for a race to happen is less than $(P-1) \cdot \frac{2\epsilon}{\delta} \cdot \frac{1}{P-1} = \frac{2\epsilon}{\delta}$.

In summary, the probability for a given processor to try to make an offer is $\frac{1}{\delta} \cdot \frac{\alpha(s)}{P-1}$ and the probability for this offer to be accepted is more than $1 - \frac{2\epsilon}{\delta}$. Overall, the probability for a task migration thus exceeds $\frac{\alpha(s)}{(P-1)\delta c}$, where c is a shorthand for $(1 - \frac{2\epsilon}{\delta})^{-1}$.

Expected local decrease in potential Consider a particular configuration s of the work stealing algorithm. Consider a particular busy processor, and let Q denote its deque. Let us investigate the variation in potential of this deque when the algorithm runs for one time step and makes a transition into some configuration s'. We call Q' the new configuration of the deque. If the processor considered sends a task to another processor, then we call Δ the potential of the singleton deque Q'' obtained by the target processor. Otherwise, we define Δ to be zero. Our goal is here to show that the relative decrease in potential, $\frac{\Phi(Q)-\Phi(Q')-\Delta}{\Phi(Q)}$ is expected to exceed $\frac{\alpha(s)}{r}$. We distinguish two cases, like in earlier earlier proofs of proof stealing, but with the difference that our potential function is carefully designed so as to avoid making too large over-approximations.

First, assume that the deque Q contains exactly one task u. The processor is working on this task and no deal can happen so $\Delta = 0$. The execution of u produces zero, one or two new tasks, which are placed into the deque, thus lying at the bottom of Q'. In the particular case where the processor has produced no tasks and ends up with an empty deque, we have $\Phi(Q') = 0$, so $\frac{\Phi(Q)-\Phi(Q')}{\Phi(Q)} \geq 1$. Otherwise Q' is not empty. All the tasks created by u have a depth potential less than that of u. So, the depth potential of Q' is less than that of Q, i.e. $G(Q') \leq G(Q) - 1$. Moreover, because the fork potential never increases, we have $F(Q') \leq F(Q)$. By definition of the potential, we have $\Phi(Q) = \mathrm{e}^{\mu G(Q) + \kappa F(Q)}$ and $\Phi(Q') = \mathrm{e}^{\mu G(Q') + \kappa F(Q')}$. Therefore, we can derive the inequality $\Phi(Q') \leq \mathrm{e}^{-\mu}\Phi(Q)$, which implies $\frac{\Phi(Q)-\Phi(Q')-\Delta}{\Phi(Q)} \geq 1 - \mathrm{e}^{-\mu}$ since $\Delta = 0$.

In order to prove the relative decrease to exceed $\frac{\alpha(s)}{r}$, we need to establish the inequality $1-\mathrm{e}^{-\mu} \geq \frac{\alpha(s)}{r}$. We define $r = \frac{(P-1)\nu}{\mu}$, where ν is a value close to 1 whose value will be specified soon. So, we have to prove $1-\mathrm{e}^{-\mu} \geq \frac{\alpha(s)}{P-1} \cdot \frac{\mu}{\nu}$. Because $\frac{\alpha(s)}{P-1} \leq 1$, it suffices to prove $1 - \mathrm{e}^{-\mu} \geq \frac{\mu}{\nu}$, which is equivalent to $\nu \geq \frac{\mu}{1-\mathrm{e}^{-\mu}}$. Mathematical analysis shows that the function $\frac{x}{1-\mathrm{e}^{-x}}$ increases with x (and tends to 1 when x tends to zero). Since μ is defined as $\frac{1}{\delta c \lambda}$ and since both c and λ are greater than 1, we have $\frac{1}{\delta} \geq \mu$. It follows that $\frac{1/\delta}{1-\mathrm{e}^{-1/\delta}} \geq \frac{\mu}{1-\mathrm{e}^{-\mu}}$. We define ν as $\frac{1/\delta}{1-\mathrm{e}^{-1/\delta}}$, and this suffices to ensure the desired inequality $\nu \geq \frac{\mu}{1-\mathrm{e}^{-\mu}}$. Note that ν is very close to 1: for example, $\nu \leq 1.06$ when δ exceeds 20, and $\nu \leq 1.0005$ when δ exceeds 1000. In conclusion, the inequality $1 - \mathrm{e}^{-\mu} \geq \frac{\alpha(s)}{r}$ holds, ensuring that the relative decrease in potential exceeds $\frac{\alpha(s)}{r}$.

The second case corresponds to the case where the deque Q contains more than one task. This means that the task at the top of deque Q may be migrated towards an empty deque. Let u be the task transfered. This task was at the top of the deque Q. Due to the deque invariant, all the other tasks from Q have a fork potential less than $f(u)$. So, $F(Q') \leq F(Q) - 1$. Because the depth potential does not increase, $G(Q') \leq G(Q)$. As a result, $\Phi(Q') \leq \mathrm{e}^{-\kappa}\Phi(Q)$, The processor that receives the task had an empty deque. It now has a deque Q'' made of the task u alone. Note that u was not at the bottom of Q is at the bottom of Q'', so we have $h(u, Q) = 1$ and $h(u, Q'') = 0$. By definition of F, we thus have $F(Q) = f(u) + 1$ and $F(Q'') = f(u) + 0$, which entail $F(Q'') = F(Q) - 1$. Furthermore, we have $G(Q'') \leq G(Q)$. So, $\Phi(Q'') \leq \mathrm{e}^{-\kappa}\Phi(Q)$. Adding this inequality with the earlier one shows

$\Phi(Q') + \Phi(Q'') \leq 2\mathrm{e}^{-\kappa}\Phi(Q)$, which, since $\Delta = \Phi(Q'')$, entails $\frac{\Phi(Q)-\Phi(Q')-\Delta}{\Phi(Q)} \geq 1 - 2\mathrm{e}^{-\kappa}$.

To summarize this second case: either no task is being transfered, in which case we have $\Phi(Q') \leq \Phi(Q)$, which is equivalent to $\frac{\Phi(Q)-\Phi(Q')-\Delta}{\Phi(Q)} \geq 0$ because $\Delta = 0$; or there is, with probability $\frac{\alpha(s)}{\delta(P-1)c}$, a successful deal which leads to a decrease in potential greater than $1-2\mathrm{e}^{-\kappa}$. Combining these two results shows that the expected decrease in potential is: $\frac{\Phi(Q)-\Phi(Q')-\Delta}{\Phi(Q)} \geq \frac{\alpha(s)}{\delta(P-1)c} \cdot (1 - 2\mathrm{e}^{-\kappa})$. By definition $r = \frac{(P-1)\nu}{\mu} = \nu\delta c\lambda(P-1)$, where $\lambda = (1-2e^{-\kappa})^{-1}$. We have $\frac{\alpha(s)}{\delta(P-1)c} \cdot (1 - 2\mathrm{e}^{-\kappa}) = \frac{\alpha(s)}{\delta c\lambda(P-1)} = \frac{\alpha(s)\nu}{r} \geq \frac{\alpha(s)}{r}$, where we used the inequality $\nu \geq 1$. This shows that the expected relative decrease in potential exceeds $\frac{\alpha(s)}{r}$.

Expected global decrease in potential In the previous section we have established that the potential of every non-empty deque decreases in expectation by at least a factor $\frac{\alpha(s)}{r}$. In this section, we show that it follows that the total potential (i.e., the sum of the potential of all the deques) decreases in expectation by at least a factor $\frac{\alpha(s)}{r}$. Consider a configuration s. Recall that $Q_i(s)$ denotes the deque of processor i in this configuration. Let $Q'_i(s')$ denote the state of the deque of i in a subsequent configuration s'. Moreover, let $\Delta_i(s,s')$ denote the potential of the deque of the processor that received a task from i when evolving from s to s', or zero if no such task was migrated. Let $\mathcal{I}$ denote the set of busy processors in configuration s, that is $\{i \,|\, Q_i(s) \neq \emptyset\}$. We can reformulate the result of the previous section as:

$$\forall i \in \mathcal{I}, \quad \Phi(Q_i(s)) - \Phi(Q_i(s')) - \Phi(\Delta_i(s,s')) \;\geq\; \frac{\alpha(s)}{r} \cdot \Phi(Q_i(s))$$

We now sum this family of inequalities, moreover summing on all possible successor configuration s'. By definition of the potential, we have $\Phi(s) = \sum_{i\in I} \Phi(Q_i(s))$. A processor can have a non-empty deque in state s' only if it had a non-empty deque in state s or if it received a task. Because a processor can receive at most one task in a time step, we have $\Phi(s') = \sum_{i\in I} \Phi(Q_i(s')) + \Delta_i(s,s')$. We can bound the expected decrease in total potential as follows.

$$\begin{aligned}
\sum_{i\in I}\sum_{s'} \rho(s,s')\left(\Phi(Q_i(s)) - \Phi(Q_i(s')) - \Delta_i(s,s')\right) &\geq \sum_{i\in I} \frac{\alpha(s)}{r} \cdot \Phi(Q_i(s)) \\
\sum_{s'} \rho(s,s') \left(\sum_{i\in I} \Phi(Q_i(s)) - \sum_{i\in I} \left(\Phi(Q_i(s')) + \Delta_i(s,s')\right)\right) &\geq \frac{\alpha(s)}{r} \cdot \Phi(s) \\
\sum_{s'} \rho(s,s') \frac{\Phi(s) - \Phi(s')}{\Phi(s)} &\geq \frac{\alpha(s)}{r}
\end{aligned} \tag{3}$$

Bound on the local number of idle tockens The result above states that the expected relative decrease is greater than a fraction $\frac{1}{r}$ of the number of idle processors $\alpha(s)$. In what follows, we show that this result can be used to bound the expected decrease in the logarithm of the potential.

$$\begin{aligned}
\sum_{s'} \rho(s,s') \frac{\Phi(s) - \Phi(s')}{\Phi(s)} &\geq \frac{\alpha(s)}{r} \\
\sum_{s'} \rho(s,s') \frac{\Phi(s')}{\Phi(s)} &\leq 1 - \frac{\alpha(s)}{r} \\
\ln\left(\sum_{s'} \rho(s,s') \frac{\Phi(s')}{\Phi(s)}\right) &\leq \ln\left(1 - \frac{\alpha(s)}{r}\right)
\end{aligned}$$

Remark: above, to take the logarithm we need to check $1 - \frac{\alpha(s)}{r} > 0$; this inequality holds because $\frac{\alpha(s)}{r} \leq \frac{P-1}{r} = \frac{1}{\delta c\lambda\nu} < 1$. At this point, for the left-hand side, we use the concavity of the logarithmic function to prove that the expectation of the logarithm is smaller than the logarithm of the expectation. For the right-hand side, we expoit the mathematical inequality $\ln(1-x) \leq -x$, instantiated with $x = \frac{\alpha(s)}{r}$. We thus get $\sum_{s'} \rho(s,s') \ln \frac{\Phi(s')}{\Phi(s)} \leq -\frac{\alpha(s)}{r}$, which entails:

$$\alpha(s) \;\leq\; r \cdot \sum_{s'} \rho(s,s') \ln \frac{\Phi(s)}{\Phi(s')} \tag{4}$$

Bound on the number of idle tockens generated from a configuration We are now ready to bound the expected number of idle tockens produced from any given non-terminal configuration s. We prove the following.

$$\mathbb{E}[I(s)] \;\leq\; r \cdot \ln \Phi(s) \tag{5}$$

The proof is by induction on the configuration DAG. If s is a configuration of the program immediately before termination, then there is a single time step remaining. At most $P-1$ processors are idle during this time step, thus $\mathbb{E}[I(s)] \leq P-1$. Moreover, there must be at least one task executing. So, at least one deque has a potential exceeding e^{μ}. It follows that $\Phi(s) \geq \mathrm{e}^{\mu}$ holds. This ensures $\mathbb{E}[I(s)] \leq P-1 \leq P' = r\mu \leq r \cdot \ln \Phi(s)$ and thereby completes the base case.

In the general case, and $\alpha(s)$ idle tokens are issued at configuration s and then the algorithm takes a step towards some configuration s'. We start from (2). On the next line, we use (4) as well as the induction hypothesis.

$$\begin{array}{rcl} \mathbb{E}[I(s)] & \leq & \alpha(s) + \sum_{s'} \rho(s,s')\, \mathbb{E}[I(s')] \\ & \leq & r \cdot \sum_{s'} \rho(s,s')\, \ln \frac{\Phi(s)}{\Phi(s')} + \sum_{s'} \rho(s,s')\, r \ln \Phi(s') \\ & = & r \cdot \ln \Phi(s) \cdot \sum_{s'} \rho(s,s') \\ & = & r \cdot \ln \Phi(s) \end{array}$$

Bound on the total number of idle tockens By applying (5) to the initial state and recalling that $I = I(s_0)$, we deduce a bound on the expected total number of idle tokens, namely I.

$$\mathbb{E}[I] \;\leq\; r \cdot \ln \Phi(s_0) \;=\; r \cdot (\mu T_\infty + \kappa H) \;=\; r \cdot \frac{(P-1)\nu}{r} T_\infty + (P-1)\nu\delta c\lambda \cdot \kappa H \;=\; (P-1)\nu \cdot (T_\infty + c\kappa\lambda \cdot \delta H)$$

We are free to chose κ. We want to minimize the product $\kappa\lambda$, which, by definition of λ, is equal to $\frac{\kappa}{(1-2e^{-\kappa})}$. Numerical analysis shows that the minimum of this expression is attained when $\kappa = 1.68$, in which case $\kappa\lambda \leq 2.68$. Thus,

$$\mathbb{E}[I] \;\leq\; (P-1)\nu \cdot \left(T_\infty \;+\; \frac{2.68}{1 - \frac{2\epsilon}{\delta}} \cdot \delta H \right) \tag{6}$$

Bound on the total number of steals We can bound the expected number of task migrations, $\mathbb{E}[s]$, in terms of the number of tokens in the idle bucket. Indeed, when a processor goes idle it takes in expectation a time $\frac{\delta}{2}$ before it receives a task, because there are at most $P-1$ processors than run the communicate function, and they do so every δ time and pick a particular processor with probability $\frac{1}{P-1}$. So, the expected delay before a processor receives a task is at least $\frac{\delta}{2}$. During a period of $\frac{\delta}{2}$, a processor contributes $\frac{\delta}{2}$ tokens to the idle bucket I. So, $\mathbb{E}[I] \geq \frac{\delta}{2}\mathbb{E}[S]$. Combining this result with (6) shows:

$$\mathbb{E}[S] \;\leq\; \frac{2}{\delta}\mathbb{E}[I] \;\leq\; 2(P-1)\nu \cdot \left(\frac{T_\infty}{\delta} \;+\; \frac{2.68}{1 - \frac{2\epsilon}{\delta}} \cdot H \right) \tag{7}$$

Bound on the number of transition tokens We now need to bound the number of tokens in the transition bucket, J. A processor is contributing to the transition bucket if it just ran out of work. At this point, it needs to complete its current offer, which takes at most 2ϵ time, and it needs to write WAITING in its in-offer field, which takes at most ϵ time. Overall, every time a processor enters an idle period, it contribues for at most 3ϵ tokens to the transition bucket. A processor enters an idle period as many times as it ends an idle period, and it exits an idle period as many times as there is a task migration. We thus have $J \leq 3\epsilon S$. Combining this with the bound $\mathbb{E}[I] \geq \frac{\delta}{2}\mathbb{E}[S]$ obtained at the previous paragraph shows $\mathbb{E}[J] \leq 3\epsilon \cdot \frac{2}{\delta} \cdot \mathbb{E}[I]$, which we can reformulate as follows.

$$\mathbb{E}[J] \quad\leq\quad \frac{6\epsilon}{\delta} \cdot \mathbb{E}[I] \tag{8}$$

Bound on the number of communication tokens We can bound the number of tokens placed in the communication bucket K in terms of the number of tokens placed in the other buckets. Recall that, by assumption, each call to the communicate function takes less than χ units of time. Moreover, a given processor makes a call with probability

$\frac{1}{\delta}$ at each time step. If T is the execution time, then the expected number of calls to the communicate made by a processor is T/δ. So, each processor contributes at most $\frac{T}{\delta}\chi$ tokens to the communication bucket. Because there are P processors, we deduce the following bound.

$$\mathbb{E}[K] \leq P\frac{T}{\delta}\chi \tag{9}$$

Conclusion We are ready to derive our final result about the expected execution time of a parallel program under the work stealing scheduler. By (1), we have $T = \frac{T_1+I+J}{P} + \frac{K}{P}$. Combining this with (9) gives $\mathbb{E}[T] = \frac{T_1+\mathbb{E}[I]+\mathbb{E}[J]}{P} + \frac{\chi}{\delta}\mathbb{E}[T]$. We deduce

$$\mathbb{E}[T] \leq \frac{1}{1-\frac{\chi}{\delta}} \cdot \frac{T_1 + \mathbb{E}[I] + \mathbb{E}[J]}{P}$$

We now use the bound on the number transition tokens (8) and that on the number of idle tokens (6). Recall that the bound above holds under the assumption that there exists an upper bound, called χ, on the cost of calling the communicate function. Moreover, we have assumed that $\delta > \max(\chi, 2\epsilon)$, to ensure that the cost of calling the communicate function is smaller than the expected delay between two calls, and to be able to bound the probability for concurrent offers to happen.

$$\begin{aligned}
\mathbb{E}[T] &\leq \frac{1}{1-\frac{\chi}{\delta}} \cdot \left(\frac{T_1}{P} + \frac{1}{P}(\mathbb{E}[I] + \mathbb{E}[J])\right) \\
\mathbb{E}[T] &\leq \frac{1}{1-\frac{\chi}{\delta}} \cdot \left(\frac{T_1}{P} + \frac{1}{P}\left(1+\frac{6\epsilon}{\delta}\right)\mathbb{E}[I]\right) \\
\mathbb{E}[T] &\leq \frac{1}{1-\frac{\chi}{\delta}} \cdot \left(\frac{T_1}{P} + \frac{P-1}{P} \cdot \frac{1/\delta}{1-\mathrm{e}^{-1/\delta}} \cdot \left(1+\frac{6\epsilon}{\delta}\right) \cdot \left(T_\infty + \frac{2.68}{1-\frac{2\epsilon}{\delta}} \cdot \delta H\right)\right)
\end{aligned} \tag{10}$$

In the bound (10), the factor $\frac{1}{1-\frac{\chi}{\delta}}$ corresponds to the overhead of calling the function communicate. The factor $\frac{1/\delta}{1-\mathrm{e}^{-1/\delta}}$ is virtually equal to one for all practical purpose. The factor $1 + \frac{6\epsilon}{\delta}$ corresponds to the overhead due to the delays associated with the offers traversing the store buffers. The constant 2.68 is the minimal possible value for the expression $\frac{\kappa}{(1-2e^{-\kappa})}$, which arises in the computation of the expected decrease in potential. Finally, the term $\frac{1}{1-\frac{2\epsilon}{\delta}}$ reflects the probability for a given offer to be accepted by the targeted processor.

We also established, in (7), the following bound on the expected number of steals:

$$\mathbb{E}[S] \leq 2(P-1) \cdot \frac{1/\delta}{1-\mathrm{e}^{-1/\delta}} \cdot \left(\frac{T_\infty}{\delta} + \frac{2.68}{1-\frac{2\epsilon}{\delta}} \cdot H\right) \tag{11}$$

If we make reasonable assumptions about the value of δ being large enough relatively to χ and ϵ, we can greatly simplify the statement of the bounds.

$$\delta > \max(101\chi, 31\epsilon) \quad \Rightarrow \quad \begin{cases} \mathbb{E}[T] \leq 1.01\,\frac{T_1}{P} + 1.20\,T_\infty + 3.44\,\delta H \\ \mathbb{E}[S] \leq 2P \cdot \left(\frac{T_\infty}{\delta} + 2.88\,H\right) \end{cases}$$

For example, consider a 2 Ghz machine, using a relatively-small delay of $\delta = 200$ microseconds (400k cycles), assuming the bound on the of the communicate function to be $\chi = 0.5$ microseconds (1000 cycles), and assuming a conservative upper bound on delay for the writes of $\epsilon = 5$ microseconds (10k cycles, which is enough to perform 64 writes taking 150 cycles each). Then, we have $\nu \approx 1 + 1.25 \cdot 10^{-6}$ and $\delta = 400\,\chi$ and $\delta = 40\,\epsilon$. Our theorem then specializes as follows.

In this example: $\mathbb{E}[T] \leq 1.0025\,\frac{T_1}{P} + 1.15\,T_\infty + 3.25\,\delta H \qquad \mathbb{E}[S] \leq 2P \cdot \left(\frac{T_\infty}{\delta} + 2.83\,H\right)$

A.4 Empirical results

	input	serial-cutoff size	serial time (s)	30-core time (s)
`bellman_ford`	2500 vertices, 6.25M edges	10K edges	40.15	3.41
`cilksort`	800M 32-bit ints	10K ints	118.01	5.45
`matmul`	5Kx5K	32x32 block	178.35	8.39
`dfs`	cage15	128 nodes	1.98	0.17

Table 1: Summary of benchmarks.