HAL
open science

Scheduling Parallel Programs by Work Stealing with Private Deques

Umut A. Acar, Arthur Charguéraud, Mike Rainey

To cite this version:

Umut A. Acar, Arthur Charguéraud, Mike Rainey. Scheduling Parallel Programs by Work Stealing with Private Deques. PPOPP - 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, Feb 2013, Shenzhen, China. pp.219-228. hal-00863028

HAL Id: hal-00863028

https://inria.hal.science/hal-00863028

Submitted on 18 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Scheduling Parallel Programs by Work Stealing with Private Deques

Umut A. Acar
Department of Computer Science
Carnegie Mellon University
umut@cs.cmu.edu

Arthur Charguéraud
Inria Saclay – Île-de-France
& LRI, Université Paris Sud, CNRS
arthur.chargueraud@inria.fr

Mike Rainey
Max Planck Institute
for Software Systems
mrainey@mpi-sws.org

Abstract

Work stealing has proven to be an effective method for scheduling fine-grained parallel programs on multicore computers. To achieve high performance, work stealing distributes tasks between concurrent queues, called deques, assigned to each processor. Each processor operates on its deque locally except when performing load balancing via steals. Unfortunately, concurrent deques suffer from two limitations: 1) local deque operations require expensive memory fences in modern weak-memory architectures, 2) they can be very difficult to extend to support various optimizations and flexible forms of task distribution strategies needed many applications, e.g., those that do not fit nicely into the divide-and-conquer, nested data parallel paradigm.

For these reasons, there has been a lot recent interest in implementations of work stealing with non-concurrent deques, where deques remain entirely private to each processor and load balancing is performed via message passing. Private deques eliminate the need for memory fences from local operations and enable the design and implementation of efficient techniques for reducing task-creation overheads and improving task distribution. These advantages, however, come at the cost of communication. It is not known whether work stealing with private deques enjoys the theoretical guarantees of concurrent deques and whether they can be effective in practice.

In this paper, we propose two work-stealing algorithms with private deques and prove that the algorithms guarantee similar theoretical bounds as work stealing with concurrent deques. For the analysis, we use a probabilistic model and consider a new parameter, the branching depth of the computation. We present an implementation of the algorithm as a C++ library and show that it compares well to Cilk on a range of benchmarks. Since our approach relies on private deques, it enables implementing flexible task creation and distribution strategies. As a specific example, we show how to implement task coalescing and steal-half strategies, which can be important in fine-grain, non-divide-and-conquer algorithms such as graph algorithms, and apply them to the depth-first-search problem.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Run-time environments

Keywords work stealing, nested parallelism, dynamic load balancing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'13, February 23–27, 2013, Shenzhen, China.
Copyright © 2013 ACM 978-1-4503-1922/13/02…$10.00

1. Introduction

As multicore computers (computers with chip-multiprocessors) become mainstream, techniques for writing and executing parallel programs have become increasingly important. By allowing parallel programs to be written in a style similar to sequential programs, and by generating a plethora of parallel tasks, fine-grained parallelism, as elegantly exemplified by languages such as Cilk [21], NESL [5], parallel Haskell [30], and parallel ML [20] has emerged as a promising technique for parallel programming [31].

Executing fine-grained parallel programs with high-performance requires overcoming a key challenge: efficient scheduling. Scheduling costs include the cost creating a potentially very large number of parallel tasks each of which can contain a tiny amount of actual work (e.g., thousand cycles), and distributing such parallel tasks among the available processors to minimize the total run time. In the course of the last decade, the randomized work-stealing algorithm as popularized by Cilk [7, 21] has emerged as an effective scheduler for fine-grained parallel programs. (The idea of work stealing goes back to 80's [9, 23].)

In work stealing each processor maintains a *deque* (doubly-ended queue) of ready tasks to execute. By operating at the "bottom" end of its own deque, each processor treats its deque as a stack, mimicking sequential execution as it works locally. When a processor finds its deque empty, it acts globally by stealing the task at the top end of the deques of a *victim*, a randomly chosen processor. In theory, work stealing delivers close to optimal performance for a reasonably broad range of computations [8]. Furthermore, these theoretical results can be matched in practice by carefully designing scheduling algorithms and data structures. Key to achieving practical efficiency is a non-blocking deque data structures that prevents contention during concurrent operations. Arora et al [2] proposed the first such data structure for fixed-sized deques. Hendler et al [24] generalized that data structure to support unbounded deques; however, the algorithm could result in memory leaks. Chase and Lev [10] used circular buffers to obtain deques whose size can grow without memory leaks. Most current parallel programming systems that support this form of work stealing critically utilize these data structures.

While the aforementioned style of randomized work stealing has been shown to be effective in many applications, previous research has also identified both algorithmic and practical limitations with it. A number of studies show that in scheduling fine-grained applications, it is important to be flexible in choosing the task(s) to transfer during a steal. These studies show both experimentally [12, 15, 25, 36, 37] and theoretically [3, 34, 35], that, for certain important irregular graph computations, it is advantageous to transfer not just a single task at every steal, but multiple tasks. For instance, previous research found the *steal-half* strategy [25]

where a steal transfers half of tasks from the victim's deque, can be more effective [12, 15, 37] compared to the "steal-one" approach. Another important practical limitation concerns the interaction between concurrent deque data structures used in the implementation of work stealing and modern memory models, which provide increasingly weaker consistency guarantees. On weak memory models, concurrent deques require expensive memory-fences, which can degrade performance significantly [17, 21, 32]; for example, Frigo et al found that Cilk's work-stealing protocol spends half of its time executing the memory fence [21]. The final limitation concerns flexibility and generality. Due to their inherent complexity, the non-blocking deques are difficult to extend to support sophisticated algorithms for creating and scheduling parallel tasks. For example, Hiraishi et al [27] and Umatani et al [42] used non-concurrent, private deques to implement their techniques for reducing task-creation overheads; Hendler et al's non-blocking, concurrent deques for steal-half work stealing require asymptotically non-constant atomic operations, and works only for bounded, fixed-size deques [25]; Cong et al found that a batching technique can reduce task-creation overheads but were not able to combine it with the flexibility of steal-half strategy [12] using private deques.

Due to these limitations, there has been a lot of interest in work-stealing algorithms where non-concurrent, *private* deques replace the concurrent, *shared* deques, and processors explicitly communicate to balance load. In such algorithms, each processor keeps its deque private, operating on its bottom end as usual. When a processor finds its deque to be empty, instead of manipulating a remote deque concurrently, it sends a message to a randomly chosen victim processor and awaits for a response, which either includes one or more tasks or indicates that victim's deque is empty. In order to respond to messages, each processor periodically polls, often driven by interrupts, its message queue.

This message-passing approach to work stealing has been receiving significant attention in multicore computers, especially more recently. In early work, Feeley [17] investigates the use of work stealing with private deques to accellerate task creation. Hendler et al. [26] use a private deques to implement a load distribution strategy for improved locality. Hiraishi et al [27] and Umatani et al [42] use private deques to reduce task-creation overheads. The Manticore system for Parallel ML uses private deques, because they simplify the parallel-garbage-collection problem by minimizing pointers between the memory of different processors [20]. Using simulation studies, Sanchez et al [37] show that minimal hardware support for message passing and interrupts can further improve the performance of work stealing, even if the private deques and the work stealing algorithm itself is implemented in software.

While previous work highlights the benefits of work stealing with private deques in terms of enabling key optimizations, algorithms, and flexible distribution strategies, relatively little is known about whether the approach can in general perform as well as work stealing with concurrent deques. Theoretically, it is not known whether private deques can yield similar theoretical guarantees as the work-stealing algorithm with concurrent deques. Practically, it is not known whether private deques can yield as good performance as state of the art systems such as Cilk that use concurrent deques. To the best of our knowledge no thorough comparison between the two approaches exist.

In this paper, we study the theoretical and the practical effectiveness of work stealing with private deques for fine-grained parallel programs on modern shared memory multicore computers. Our algorithms are specifically designed for fine-grained parallel computations, where the size of the tasks are small, often in the thousands of clock cycles. We propose two algorithms, a sender- and a receiver-initiated algorithm for work stealing. Using private deques, our algorithms eliminate memory fences from the common scheduling path.

To balance load, our algorithms rely primarily on explicit communication between processors. To provide efficient communication, our algorithms exploit a key property of fine-grained parallel programs: that calls to the scheduler are frequent because the tasks are naturally small. Thus, most of polling needed for communication can be performed by the scheduler without (hardware or software) interrupts, which, depending on the platform, may not be able to deliver interrupts frequently and cheaply enough. Although the treatment of parallel programs with coarse granularity is out of the scope of this paper, we report on some preliminary investigations on the use of interrupts to make our algorithms robust in the face of potentially-large tasks.

We give a proof of efficiency for both the sender- and receiver-initiated work-stealing algorithms using private deques. For our analysis, we consider a probabilistic model, which takes into account the delays due to the interval between polling operations. We present a bound in terms of the work and depth (traditional parameters used in the analysis of work stealing), and a new parameter, the *branching depth*, which measures the maximal number of branching nodes in the computation DAG. The branching depth is similar to traditional notion of depth but is often significantly smaller because it counts only the number of fork nodes along the path, ignoring sequential work performed in between. The branching depth parameter enables us to bound tightly the effect of the polling delays on performance, showing that the algorithm performs close to a greedy scheduler, even when the communication delay is quite large. Due to space restrictions we could not provide all the details of the proof, which can be found in the accompanying technical report accessible online [1].

We present an implementation of our algorithm as a C++ library. To evaluate the effectiveness of our implementation, we consider a number of parallel benchmarks, including standard Cilk benchmarks as well as more recently proposed graph benchmarks from the PBBS [4] benchmark suite. Using these benchmarks, we compare our algorithms with Cilk (more precisely, Cilk Plus [28]). Furthermore, in order to isolate the differences due to the use of private deques from the differences due to the representation of tasks and other implementation details, we compare our algorithms against an implementation of the standard Chase Lev work stealing algorithm [10] in our framework. Our experiments show that our algorithms are competitive with both Cilk and to our own implementation of work stealing with concurrent deques.

A key benefit of the proposed approach with private deques is that it eliminates all concurrency operations from local deque operations. More precisely, our algorithms require only a simple, non-concurrent deque data structure, because all other load balancing actions are performed via explicit communication. This approach allows implementing sophisticated task-creation and scheduling algorithms as may be needed by the application at hand, e.g., those that do not fit into the divide-and-conquer or nested data parallel paradigm. As an important example, we show how to coalesce small tasks into larger tasks while also supporting the steal-half policy for load balancing. These techniques enable us to consider applications beyond divide-and-conquer (nested data parallel applications), where granularity control is not possible by sequentialization of small tasks as in the divide-and-conquer computations. As a specific example, we consider a parallel depth-first-search (DFS) algorithm.

2. Algorithms

Our sender-initiated and receiver-initiated algorithms both follow the same skeleton but differ in how they perform the load balancing actions. Figure 1 shows the parts that are shared by both algorithms.

```
deque<task*> q[P] = {EMPTY, ..} // deques

// entry point for the workers
void main(int i) // i = ID of the worker
  repeat
    if (empty(q[i]))
      acquire(i)
    else
      task* t = pop_bottom(q[i])
      update_status(i)
      communicate(i)
      execute(t)

// called for scheduling a ready task t
void add_task(int i, task* t)
  push_bottom(q[i], t)
  update_status(i)
```

Figure 1. Scheduler code for work stealing with private deques

```
bool a[P] = {false, .. } // status flag
int NO_REQU = -1
int r[P] = {NO_REQU, .. } // requests cells
task* NO_RESP = 1 // any non-null pointer
task* t[P] = {NO_RESP, .. } // tranfer cells

// update the status flag
void update_status(int i)
  bool b = (size(q[i]) > 0)
  if a[i] != b then a[i] = b

// called by workers when running out of work
void acquire(int i)
  while true
    t[i] = NO_RESP
    int k = random in {0, .., P-1}\{i}
    if a[k] && compare_and_swap(&r[k], NO_REQU, i)
      while (t[i] == NO_RESP)
        communicate(i)
      if (t[i] != null)
        add_task(i, t[i])
        r[i] = NO_REQU
        return
    communicate(i)

// check for incoming steal requests
void communicate(int i)
  int j = r[i]
  if j == NO_REQU then return
  if empty(q[i])
    t[j] = null
  else
    t[j] = pop_top(q[i])
  r[i] = NO_REQU
```

Figure 2. Receiver-initiated algorithm

Each of the P processors owns a deque (doubly-ended queue of tasks). The deque is accessible by its owner only. The function `main` implements main scheduling loop. The loop starts by checking if the deque is empty. If so, it calls the function `acquire`, which obtains a task to execute. Otherwise, it pops the bottom task from the deque and executes it. When executed, a task can create new subtasks, which are then pushed at the bottom of the deque with function `add_task`. Between the execution of every two tasks, the function `communicate` is called, for the purpose of load balancing, to communicate with other processors. Observe that the call to `communicate` takes place after the pop operation, ensuring that a processor never sends away the last task that it owns. The receiver-initiated algorithm and the sender-initiated

```
task* DUMMY_TASK = 1 // any non-null pointer
task* INCOMING = 2 // another non-null pointer
task* s[P] = {DUMMY_TASK, .. } // communication cells
double d[P] = { 0, ...} // date of next deal attempt

// called by workers when running out of work
void acquire(int i)
  s[i] = null
  while (s[i] == null)
    noop
  add_task(i, s[i])

// attempt to deal a task to an idle processor
void deal_attempt(int i)
  if empty(q[i]) then return
  int j = random in {0, .., P-1}\{i}
  if s[j] != null then return
  bool r = compare_and_swap(&s[j], null, INCOMING)
  if r then s[j] = pop_top(q[i])

// call try_send if it is time to do so
void communicate(int i)
  if now() > d[i]
    deal_attempt(i)
    d[i] = now() - delta * ln (rand(0,1))
```

Figure 3. Sender-initiated algorithm

algorithm differ only in the design of the function `acquire` and `communicate`. The auxiliary function `update_status`, which appears in the the function `main`, is used by the receiver-initiated algorithm only.

Receiver-initiated algorithm Figure 2 shows the pseudo-code for the receiver-initiated algorithm. Processors communicate via two kinds of cells: *request cells*, stored in the array `r`, and *transfer cells*, stored in the array `t`. Each processor has its own request and transfer cell. In addition, each processor uses the array `a` to indicate that its deque contains more than one task (i.e., that the processor has work to offer). The function `update_status` updates the value stored in this cell.

In the receiver-initiated algorithm, when an idle processor calls the function `acquire`, it picks a random target "victim" processor. The idle processor then reads the status cell of the victim processor to determine whether the victim processor has some work to offer. If not, it starts over with another random target. If, however, the victim processor has some work to offer, then the idle processor makes a steal request. To that end, it writes atomically (with a compare-and-swap operation) its id in the request cell of its victim processor. The atomic write guarantees a processor receives at most one steal request at once. If the atomic write fails, the processor starts over; if it succeeds, then the idle processor simply waits for an answer from its victim, by repeatedly reading its transfer cell.

Whenever a busy processor calls the function `communicate`, it checks whether its request cell contains the processor id of a thief. If so, then it responds to the thief by writing the top task in its deque to the transfer cell of the thief. Otherwise, if the processor has no more than one task then it declines the request by sending the null pointer. Since write operations can take some time to become visible to all processors, a processor may receive steal requests while it is already idle and running the function `acquire`. There are two ways to ensure that the steal request receives a response in such a case. One possibility, which we follow in this paper, is to have the idle processor call `communicate` regularly while looping in the function `acquire`. Another possibility, which we implement and which is described in the long version of the paper [1], is to have the idle processor atomically write its own id in its request cell, thereby blocking incoming requests.

Sender-initiated algorithm Figure 3 shows the pseudo code for the sender-initiated algorithm. Each processor uses a *communication cell*, both to indicate its status and to receive tasks. These cells are stored in the array `s`. Each processor additionally keeps track of the next date at which it should make a deal attempt, using the array `d`. We will explain later why these dates are needed.

In the sender-initiated algorithm, when an idle processor calls the function `acquire`, it simply declares itself as idle by writing the value null in its communication cell. It then waits until a busy processor delivers work in this cell. A busy processor uses the function `deal_attempt` to attempt to deal a task to an idle processor. To make a deal attempt, the busy processor first checks whether its deque is empty. If so, the busy processor returns immediately because it cannot send a task. Otherwise, the busy processor picks a random target, and checks whether this target is idle, by testing whether the communication cell of the target contains the value null. If the target is not idle, then the busy processor gives up, that is, it does not try to find another target. If the target is idle, then the busy processor tries to atomically update the communication cell of the target by writing the constant `INCOMING` into it, so as to prevent other processors from concurrently delivering a task. If the atomic operation succeeds, the busy processor pops the task from the top of its deque, and writes the corresponding pointer into the communication cell of the target. If the atomic operation fails, indicating that the busy processor has been out-raced by another busy processor, the busy processor simply aborts.

In the particular case of the steal-one policy, which is being described here, we can save the intermediate write of the constant `INCOMING` and instead directly send a task pointer. This optimization can be obtained by replacing in Figure 3 the last two lines of the function `deal_attempt` with the following code.

```
task* t = peek_top(q[i])
bool r = compare_and_swap(&s[j], null, t)
if r then pop_top(q[i])
```

The two-step process described in Figure 3 is, however, required to support policies such as steal-half, as discussed in Section 5.

Consider the execution of a processor i that is working on a collection of small tasks. If the tasks owned by i are smaller on average than those owned by other processors, then i would have more chances of dealing tasks than other processors. Because the tasks that i deals are small, many more task migrations would be needed than in a fair situation, where processors owning big pieces of the computation have similar chances of dealing them.

To ensure fairness, we could impose that busy processors make deal attempts only at regular intervals. We have observed in practice slightly better and much more regular results when we introduce randomness in the intervals between deal attempts. There are many possible ways of introducing randomness. Our approach, which follows the assumption that we make in the proof of efficiency, consists is making the delay between two deal attemps follow a Poisson distribution with parameter δ, for some δ larger than the typical duration of a task. With this approach, deal attempts take place on average slightly more than every δ, because a processor needs to complete a task before it is able to check whether the time has come to make a deal attempt.

Once a deal attempt is made, to determine the time for the next deal attempt according to the Poisson distribution, we use Knuth's formula $-\delta \ln(x)$, where x is a random variable uniformly picked in the range $[0, 1]$. As an optimization, we do not reset the date `d[i]` to the value `now()` at the end of the function `acquire`, meaning that we typically allow a processor that receives a task in `acquire` to make a deal attempt immediately after it has executed this task. This optimization significantly helps in distributing the work quickly in the initial phase of a parallel algorithm.

3. Analysis

When using concurrent deques, idle processors are able to almost immediately acquire some work by stealing it from the deque of one of the busy processors. On the contrary, when using private deques, idle processors need to wait for a busy processor to communicate with them. A central aspect of work stealing algorithms based on private deques is therefore to quantify the amount of additional idle time induced by the communication delays. In this section, we prove a bound showing that the amount of idle time is bounded by $O(\delta F)$, where δ denotes the average communication delay and where F denotes the branching depth, that is, the maximal number of branching nodes in a path from the computation DAG.

To model the communication pattern in the proof, we use a probabilistic model. For the sender-initiated algorithm, we assume that deal attemps follow a Poisson distribution with parameter δ. This model is faithful to the behavior of the actual algorithm whenever δ is larger than the duration of a few tasks. Note that the larger is δ compared with the typical size of sequential tasks, the more faithful is the model.

For the receiver-initiated algorithm, we assign a different interpretation to the variable δ: we assume that the interval between two polling operations made by a given processor follow a Poisson distribution with parameter δ. The parameter δ here corresponds to the average duration of a sequential task. In the actual algorithms, some polling operations actually happen more frequently, because of the "fork tasks" and the "join tasks" which perform only a tiny amount of work. In the receiver-initiated algorithm, these additional polling operations can only help the algorithm by accelerating the distribution of tasks. The direct cost of these additional polling operations, which consists simply in reading a local variable, is negligible in front of the costs associated with the creation and the manipulation of tasks. Note that the receiver-initiated algorithm has no issue with fairness like that of the sender-initiated one, because, in the receiver-initiated algorithm, all the random decisions are made by the idle processors.

Our proof establishes a bound on the execution time for both the receiver- and the sender-initiated algorithms. Before stating our bound, we briefly recall the bounds from the literature for work stealing with concurrent deques. The proof given by Blumofe and Leiserson [6], later simplified and generalized by Arora, Blumofe, Plaxton in [2], is $\mathbb{E}[T_P] \leq \frac{T_1}{P} + 32T_\infty$, where T_P denotes the execution time with P processors, T_1 denotes the sequential execution time, and T_∞ denotes the length of the critical path (which corresponds to the minimal execution time with infinitely-many processors). This bound is established using an potential analysis based on *phases*: at each phase, the relative decrease in potential exceeds $\frac{1}{4}$ with probability greater than $\frac{1}{4}$. Tchiboukdjian et al [40] tightened this bound to $\mathbb{E}[T_P] \leq \frac{T_1}{P} + 3.65T_\infty$, using an analysis based on a bound of the expected decrease in potential at each time step. This bound shows that work stealing is not far from matching Brent's bound $\frac{T_1}{P} + \frac{P-1}{P}T_\infty$, which applies to all greedy schedulers.

Our proof is also based on the expected decrease in potential, however it uses a different potential function, which depends on the value of δ and which distinguishes the contribution of T_∞ from that of the branching depth. In first approximation, the bound that we establish for both receiver-initiated and sender-initiated work stealing with private deques is:

$$\mathbb{E}[T_P] \leq\approx \left(1 + \frac{1}{\delta - 1}\right) \cdot \left(\frac{T_1}{P} + T_\infty + O(\delta F)\right).$$

The bound above includes a factor $1 + \frac{1}{\delta-1}$, which corresponds to the overhead associated with polling. The constant 1 that appears in the denominator should be interpreted as the round-trip time for a message to go back and forth between two processors. The bound

	Concur. deques (speedup)	Concur. deques (sec)	Recv.-init. (%)	Sender-init. (%)	Cilk Plus (%)
matmul	21.7	2.61	-18	-18	-3
cilksort(exptintseq)	18.6	1.32	-2	-0	-7
cilksort(randintseq)	21.7	1.51	-2	+0	-7
fibonnacci	26.2	4.11	-2	+1	-3
matching(eggrid2d)	19.6	0.44	+9	+12	+9
matching(egrlg)	20.0	0.72	-1	+2	+5
matching(egrmat)	20.1	0.90	+0	+4	+6
MIS(grid2d)	17.5	0.19	+2	-0	+5
MIS(rlg)	17.9	0.21	-4	-2	+7
MIS(rmat)	18.5	0.16	+1	+4	+7
hull(plummer2d)	18.0	0.27	+6	+4	-5
hull(uniform2d)	19.1	0.55	+2	+2	-3
sort(exptseq)	23.2	1.90	-4	-4	+29
sort(randdblseq)	23.5	2.84	-7	-6	+25

Figure 4. Comparison of the schedulers.

also includes the term $O(\delta F)$, which corresponds to the idle time associated with task migrations. The formal lower bound that we prove on $\mathbb{E}[T_P]$ involves a constant c, defined as 1.0 in the sender-initiated algorithm and $\frac{1}{1-1/e} \approx 1.58$ in the receiver-initiated algorithm. It also involves a factor $\frac{\mu}{1-e^{-\mu}}$, where μ is defined as $\frac{0.63}{c\delta}$. Since $\frac{\mu}{1-e^{-\mu}} \approx 1 + \frac{\mu}{2} \approx 1 + \frac{0.31}{c\delta}$, the factor $\frac{\mu}{1-e^{-\mu}}$ can be approximated as 1.0 for any practical purpose. The formal bound is:

$$\left(1 + \frac{1}{\delta - 1}\right) \cdot \left(\frac{T_1}{P} + \frac{P-1}{P} \cdot \frac{\mu}{1-e^{-\mu}} \cdot (T_\infty + 2.68 \cdot c\delta F)\right).$$

The fact that c is larger in the receiver-initiated algorithm corresponds to the fact that idle processors may need some time to find a busy target. Note that this difference does not imply that the receiver-initiated algorithm is slower than the sender-initiated one, because the former algorithm is associated with a smaller value for δ. The main arguments of the proof can be found in the appendix of this paper. The complete proof can be found in the online appendix [1].

4. Evaluation

We implemented a C++ library to provide a framework for evaluating our algorithms. The library creates one POSIX thread (i.e., one *pthread*) for each core available. We implemented the receiver-initiated and the sender-initiated algorithms with private deques, as well as the standard Chase-Lev algorithm based on concurrent deques [10]. We also compare against Cilk Plus, an extension of GCC, that is the result of many years of careful engineering. Our goal is to evaluate whether private deques can be competitive with our own implementation of concurrent deques, and whether this baseline is competitive with the state of the art technology.

Comparison. We evaluated the schedulers on several programs. First, we ported three classic Cilk programs: *cilksort*, which is based on a parallel version of merge-sort; *matmul*, which multiplies two dense matrices in place using a cache-efficient, divide-and-conquer algorithm [21]; and *fibonacci*, which computes Fibonacci number using the exponential algorithm. This last benchmark is useful to perform analyses without observing interference from the memory. We also ported four benchmarks from the recent Blelloch et al's problem-based benchmark suit (PBBS) [4], which consists of internally-deterministic parallel programs targeting Cilk. We ported: *matching*, which computes the maximal matching of an undirected graph; *hull*, which computes a 2-dimensional convex hull; and *sample-sort*, which is a low-depth, cache-efficient version of the classic sample sort algorithm.

In order to reuse some parts of Cilk Plus, and in order to ease the comparison, we use the same heap allocator (*miser* [39]), the same random number generator, and the same compiler as Cilk. We left the benchmarks programs exactly as they were implemented originally, only increasing slightly the sequential cutoff value in the three Cilk benchmarks programs to adapt to the speed of our test machine. One difference, though, concerns the implementation of Cilk's parallel for-loops, which is used by the three PBBS benchmarks. The strategy of Cilk consists in statically partitionning loops in $8P$ subtasks. This approach results in the creation of large sequential tasks, which is problematic for schedulers based on private deques. Instead, we use a divide-and-conquer approach to scheduling parallel loops, simply cutting off at a number of iterations that roughly corresponds to 10 microseconds worth of work. The difference in the number of subtasks generated explains the significant difference in execution time observed on some benchmarks between Cilk Plus and our implementation of concurrent deques.

Our test machine hosts four eight-core Intel Xeon X7550 [29] chips with each core running at 2.0GHz. Each core has 32Kb each of L1 instruction and data cache and 256 Kb of L2 cache. Each chip has an 18Mb L3 cache that is shared by all eight cores. The system has 1Tb of RAM and runs Debian Linux (kernel version 3.2.21.1.amd64-smp). We consider just 30 out of the 32 total cores in order to reduce interference with the operating system. All of our code is compiled by the Cilk Plus GCC (v4.8.0 20120625) with the `-O2` option. For the sender-initiated algorithm, we set the delay parameter δ to 30 microseconds, which we have found to yield good performance on our machine. The input sizes are as follows: *cilksort*: random and exponentially-distributed, 240m integers, *matmul*: square matrix of size 3500, *fibonacci*: n = 48, *matching*: 3-d grid with 40m nodes, random graph with 40m nodes and 200m edges, and rMat graph with 40m nodes and 200m edges, *hull*: uniform and plummer with 100m points, *sample-sort*: random and exponentially-distributed, 240m doubles. To tame the variance observed in the measures when running with 30 cores (there is usually between 5% and 10% difference between a fast and a slow run), we averaged the measures over 20 runs.

Figure 4 gives the speedup and the absolute execution time for our baseline (Chase-Lev concurrent-deques algorithm), and gives the relative value of the execution time of the other schedulers: our receiver-initiated, our sender-initiated algorithms, and Cilk Plus. Several interesting conclusions can be drawn from this figure. First, the receiver-initiated algorithm and the sender-initiated algorithm perform almost exactly the same (usually within 2% of each other). This similarity confirms the intuition that, at a high-level, these algorithms are dual of one another. Second, we observe that, on many benchmarks, private deques are performing close to concurrent deques, sometimes a little worse and sometimes a little better. In one particular benchmark, such as maximal matching on a grid, private deques shows poorer performance than concurrent deques. This benchmark involves some phases where parallelism is so scarce that the communication delay becomes visible. In some other benchmarks, such as *matmul* and *sample-sort*, private deques seem to perform significantly better. We believe that, in these memory-intensive benchmarks, saving the cost of the memory fence brings a significant improvement. Third, we observe that our baseline is competitive with Cilk Plus. Our library is never more than 7% slower, and it is often about 7% faster. Moreover, due to our different treatment of for-loops, we are able to outperform Cilk by over 25% on the sample sort benchmark. From these results, we conclude that the private-deque approach to work stealing is competitive with state of the art, concurrent-deques algorithms.

Although having competitive performance in practice is crucial, the true benefits of the private-deque approach relate to flexibility and generality. In Section 1, we discussed the benefits, but here, we

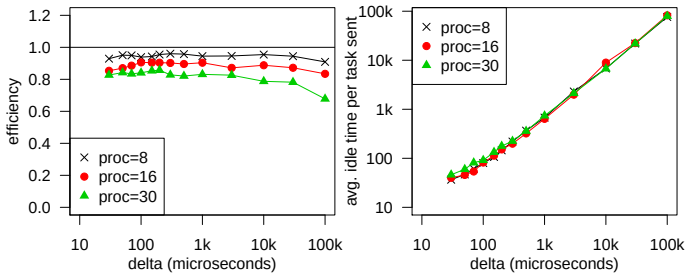

Figure 5. Impact of δ in the sender-initiated algorithm (Fibonacci).

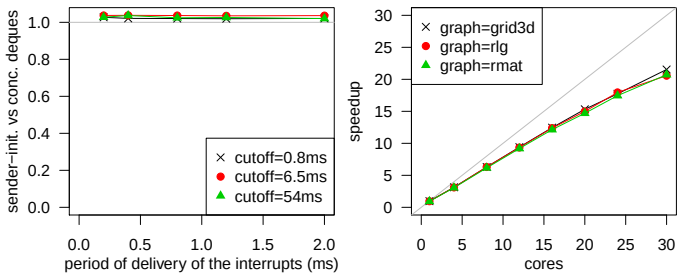

Figure 6. (a) Handling of large sequential tasks using interrupts, and (b) Speedup curves for pseudo-DFS on three graphs.

provide further evidence to back these claims. The development of state-of-the-art concurrent-deque algorithms dates back to the non-blocking algorithm of Arora et al. [2], which went through a few revisions due to concurrency bugs. Several years later the nonblocking algorithm was extended by Chase and Lev to support dynamic resizing [10]. Their first and, to our knowledge, only proof of correctness of a nonblocking work-stealing algorithm is not trivial: it spans over thirty pages [11]. Moreover, there is, in the literature, no nonblocking algorithm which combines resizeability with other extensions, such as steal half, possibly owing to the complexity involved in extending the proof of correctness. Although we had to omit the proof of correctness for our sender- and receiver-initiated algorithms due to space limitations, the proofs are trivial because in both cases the concurrency is limited to accesses on a single shared cell. The private-deque algorithms support steal half as well as other extensions that are not yet supported by concurrent deques.

Analysis of the impact of δ in the sender-initiated algorithm. In the evaluation, we have been setting δ to 30 microseconds. In benchmarks where the branching depth is large, typically in algorithms that have an outer sequential loop and an inner parallel loop, the value of δ needs to remain relatively small in order to efficiently distribute tasks. However, in benchmark where the branching depth is small, the value of δ can be safely increased without noticeably affecting on the execution time. According to our theorem, it is perfectly fine to use any δ such that $2.68 \cdot \delta F \ll \frac{T_1}{P}$. For example, for *fibonacci* with $n = 48$ and sequential cutoff at $n = 18$, the fork depth is 30. Given that $T_1 = 56$ seconds and $P = 30$, $\frac{T_1}{2.68 \cdot FP} = 23$ milliseconds. Therefore, up to $\delta = 1$ millisecond, we do not expect to see any effect on the execution time. This theoretical prediction is confirmed by the first chart shown in Figure 5.

To better understand the impact of δ on the idle time involved in an execution, we measure the ratio between the total amount of idle time and the number of tasks being migrated between processors. At high load, when a processor runs out of work, there are $P - 1$ processors that may send work to it; each of these busy processors performs a deal attempt on average every δ, and find the idle processor with probability $\frac{1}{P-1}$. As a result, the expected time before a processor receives some work is exactly δ. The second chart in Figure 5 confirms that the average idle time per task migration is indeed extremely close to δ. Because the number of task migrations is typically small when the deque discipline of work stealing is followed, the total amount of idle time is relatively small and grows only linearly with δ.

Handling of large sequential tasks If we cannot assume that the tasks have a bounded execution time, then we need to adapt the algorithms so as to ensure that the control is handed back to the scheduler regularly enough. Several approaches are possible. One possibility, in the receiver-initiated algorithm, is for idle processors to send interrupts to their targets. However, this approach does not apply to the sender-initiated algorithm. We could resort to compiler-assisted software polling, where the compiler inserts into the program operations to check for incoming messages [18, 19], or to the use of periodic interrupts. In order to make our C++ library more generally applicable, we do not use software polling. Ideally, we would use interrupts triggered by some form of hardware down-counter. For now, we have instead been using a more basic approach that consists in running an additional pthread that issues interrupts at regular intervals. With this approach, we are able to get interrupts delivered as frequently as every 200 microseconds.

When interrupting tasks during their execution, we need to prevent races between the interrupt handler and the action of the running task. In particular, we need to prevent a race from corrupting the deque. Such races are much simpler to handle than that involved in concurrent deques, because interrupts happen on the same core, and therefore with a consistent view of the memory. Futhermore, their execution is not arbitrarily interleaved with that of the running task. If suffices for the actions on the deque to be protected by a local lock, which can be implemented without atomic operation. If an interrupt is raised during a critical section, then it can be ignored because the scheduler has the control during these critical sections, so it is able to execute a polling operation anyway.

As explained earlier on, for all programs with limited branching depth, δ can be set to 200 microseconds or even more without noticeable effect on the execution time. To evaluate the overheads associated with interrupts, we considered the dense matrix multiplication benchmark (on a parallel run of 3.7s), and varied the sequential cutoff so as to generate sequential tasks of size either 0.8ms, or 6.5ms, or 54ms. We measured the efficiency of our sender-initiated algorithm equipped with periodically-delivered interrupts relative to our implementation of concurrent deques. The results appear in the the first plot in Figure 6. For all periods tested between 200 microseconds and 2 milliseconds, and for any of the three sequential cutoff, we observe that the execution time is only 3.7 % slower than the baseline. On other benchmarks, we also observed overheads of the same order of magnitude. These results suggest that even the sender-initiated algorithm extended with interrupts can achieve competitive performance in the context of corse-grained parallelism.

5. Going beyond divide-and-conquer parallelism

In this section, we investigate the benefits of using private deques for programs falling outside of the fork-join model. We explain why concurrent deques are limited when it comes to supporting the steal-half policy and task coalescing. We show that, on the contrary, private deques can easily accomodate these two features. We demonstrate this ability by implementing a pseudo-DFS algorithm, which computes reachability in a graph from a given source.

Steal half The steal-half policy consists in transferring, during one steal operation, half of the tasks in the deque, instead of only one task. A number of researchers have argued for the benefit of the steal-half policy, in particular in the context of irregular

graph applications [3, 12, 15, 34, 35, 37]. Intuitively, the weakness of the steal-one policy in a non fork-join computation is that, when a processor receives a single task, it is likely to run out of work soon afterwards. Hendler et al. have developed a concurrent deque able to support the steal-half strategy, but only at the cost of logarithmically many atomic operations in the total number of deque accesses [25]. Furthermore, Hendler et. al.'s data structure supports only fixed-sized deques, and it is not known if the data structure can be generalized to support resizable deques. When using private deques, however, implementing the steal-half policy requires only a trivial change to the algorithm. For example, in the sender-initiated pseudo-code, it suffices to change the type of the communication cells to `vector<task*>*` and to update the code from Figure 3 so that the busy processor sends a vector of tasks carved out of its own deque.

```
// change the last line of deal_attempt to:
int half_size = (size(q[i]) + 1) / 2
if r then s[j] = q[i].extract_items(half_size)
// change the last line of acquire to:
delete(q[i]) // q[i] is empty here
q[i] = s[i]  // use the incoming vector
```

Remark: when vectors are used to represent deques, the splitting operation has a linear cost. This cost is usually well amortized, because the average number of times that a task is transfered from a processor to another is usually tiny. Furthermore, if needed, the cost of splitting can be made logarithmic instead of linear by using more advanced data structures, such as binomial trees, which achieve logarithmic-time splitting and amortized constant-time push and pop operations.

More generally, work stealing with private deques can easily accomodate a wide range of transfer policies and accomodate efficient data structures to implement these policies, without requiring the development, for each policy, of a specific concurrent data structure.

Task coalescing To achieve good speedups, the task-creation costs need to be well amortized. In divide-and-conquer programs, this type of amortization is obtained by sequentializing the execution of subtasks smaller than some threshold. However, this technique does not apply to less structured applications such as irregular graph algorithms. To make matters worse, in this type of applications, the amount of work associated with each individual task is usually tiny. For example, in pseudo-DFS, if one task corresponds to the treatment of a single node from the graph, then the task-creation overheads are overwhelming. We conducted experiments showing that these overheads typically slow down the program by a factor 3 or more.

Task coalescing is a classic approach to reducing the overheads. It consists in grouping similar tasks into one, in order to reduce the work associated with task creation. In the case of pseudo-DFS, one coalesced task describes not just one node but a batch of nodes to visit. Let us explain why task coalescing is incompatible with the use of concurrent deques. If the size of the batches (number of nodes contained in each task) is constant, then it can happen that never more than one task is created, resulting in a purely sequential run. (Consider the case where the graph is a complete binary tree: the size of the stack of nodes to visit never contains more than a logarithmic number of nodes at once.) To overcome this problem, Cong et al [12] suggest the following policy: if the number of nodes in the batch of the currently-running task is about to exceed $\min(2^Q, S)$, where Q is the size of the local deque and S is a constant large-enough to amortize scheduling costs, then the current batch is packed into a new task and pushed into the deque. While this approach can be effective for relatively regular graphs, it suffers from prohibitive overheads on all the graphs where the size of the stack of nodes to visit remains small (typically, in balanced trees and graphs with hierarchical clusters), because in this case one task typically contains a small number of the nodes. Moreover, Cong et al's cannot be combined with steal half.

By contrast, when using private deques, task coalescing is straightforward to implement and can be combined with steal half. For pseudo-DFS, each processor can use a single task, which contains a vector of nodes to visit. When executed, the task processes no more than a constant number of S nodes, where S is large enough to amortize the scheduling overhead. Before it continues, the task hands the control back to the scheduler. If the scheduler needs to transfer some work to another processor, then it may call on the task a splitting function that returns a new task containing half of the nodes.

We have implemented pseudo-DFS with task coalescing and the steal-half policy, using simple vectors to represent set of nodes. We implement efficient termination detection by having each processor keep a local count of the difference between the number of tasks received and the number of tasks sent. Termination occurs when the sum of the per-processor counts equals zero. One processor, assigned arbitrarily, performs the check when idle. We benchmarked our program on the three kinds of graphsthat Blelloch et. al. [4] used to benchmark their BFS program: *3d-grid* (40m nodes, $T_1 = 11.5$s), *rlg* (40m nodes, 150m edges, $T_1 = 12.8$s), and *rmat* (40m nodes, 90 edges, $T_1 = 9.1$s). The second plot in Figure 6 shows that, on each of the three graphs, we achieve over 20x speedup with 30 cores.

In summary, the use of private deques offer a lot of flexibility. They allow for simple implementations of various scheduling techniques, without having to worry about the performance and the correctness of ad-hoc concurrent data structures.

6. Related work

We have discussed closely related work in relevant sections, in particular in Sections 1 and 5; here, we briefly review other more remotely related work, specifically the work on distributed systems. In distributed systems (without shared memory), scheduling algorithms usually rely on explicit communication between processors rather than concurrent data structures. Our algorithms therefore share some properties of distributed scheduling algorithms. Our algorithms also differ from distributed ones, because we perform communication via hardware shared memory and use atomic operations to maintain certain critical invariants.

Using the logp model, Sanders [38] analyzes a receiver-initiated load-balancing algorithm for a subclass of tree-shaped computations, presenting strong bounds that show the approach to be theoretically efficient. In contrast to the literature on hardware shared-memory systems where there is relatively little discussion of the sender-initiated approach, many studies on distributed scheduling compare the receiver- and sender-initiated approaches. Eager et al. [16] compare sender-initiated policies under different job scheduling policies, finding that performance depends on the system load as well as cost of certain operations, such as task transfers. Followup work refines these comparisons by considering the delays in the system [33], and different job scheduling policies [13]. More recently Dinan et al [14] compare work stealing (receiver initiated) and work sharing (sender initiated) when implemented on top of the MPI interface for message passing by using the unbalanced tree-search benchmark. These papers find that the algorithms both perform quite well—there are no clear winners—and the specifics such as the delays, the system load, and the job scheduling and preemption policies can make one preferable over the other.

Our empirical results also show that the two algorithms perform similarly on shared memory architectures. That said, workload characteristics and future advances in hardware and can make one more effective than the other. For example, receiver-initiated

algorithms may be not as well suited to multiprogrammed environments because an idle processor takes exclusive access to a specific victim, which can delay execution if the sender is swapped out. Also, in receiver-initiated systems, processors can spin while looking for work, thereby making it difficult for the job scheduler to identify idle processors [22]. In contrast, in the sender-initiated approach, any sender can send work to an idle processor, and idle processors do not spin to look for work.

Tzannes proposes an algorithm in which each processor keeps all tasks in a private deque, except for the topmost one, which is exposed in a shared cell [41]. Although slightly simpler than our receiver-initiated algorithm, Tzannes' algorithm could show worst-case behavior when given certain computation graphs that lead the algorithm to access the topmost task at a high frequency, because the scheduler would have to repeatedly push and pop using compare and swap on the shared cell. (The Chase-Lev algorithm has a similar problem.)

7. Conclusion

Two factors involving hardware and software are closing the gap between hardware shared-memory systems and distributed systems. On the hardware side, as multicore processors grow larger, they resemble distributed systems. For example, in hierarchical and non-uniform memory systems, accesses to a local cache can be orders of magnitude faster than non-local memory, and, due to the weak memory-consistency model, operations on concurrent data structures can require expensive synchronization, even when executing local operations. On the software side, the emergence of fine-grained parallelism as a paradigm for programming multicore computers has made it important to control the costs of communication, i.e., memory reads/writes, and synchronization operations due to cache and memory consistency.

The work-stealing paradigm has emerged, in its many instances, as an important technique for scheduling fine-grained parallel computations on multicore computers. But many of the existing algorithms for work stealing rely on concurrent data structures that can impose large synchronization costs and can be difficult to adapt for different applications, e.g., irregular problems, such as the depth-first-search algorithm, that do not fit into the divide-and-conquer model, where controlling costs is simpler.

In this paper, we design, analyze, and empirically evaluate two work-stealing algorithms for executing fine-grained parallel programs on modern multicores. Our analysis shows the algorithms to be competitive with the optimal bounds. Our implementation and experiments show that they are competitive with Cilk Plus, a state-of-the-art, highly optimized software system. We show that, thanks to eliminating concurrency from local deque operations, our approach enables designing and implementing sophisticated task coalescing and scheduling techniques that accellerate irregular problems. As a challenge benchmark, we consider depth first search and obtain encouraging results. Our results suggests that as we tackle the parallelism challenge, we may benefit from insights resulting from previous work from the field of distributed computing.

References

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. Technical report, Max Planck Institute for Software Systems, 2013.

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

[3] Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. The natural work-stealing algorithm is stable. *SIAM J. Comput.*, 32:1260–1279, May 2003.

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

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

[6] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. *Foundations of Computer Science, IEEE Annual Symposium on*, 0:356–368, 1994.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] Guojing Cong, Sreedhar B. Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay A. Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*, pages 536–545, 2008.

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

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

[15] 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.

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

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

[18] 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.

[19] 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.

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

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

[22] David Grove, Olivier Tardieu, David Cunningham, Ben Herta, Igor Peshansky, and Vijay Saraswat. A performance model for x10 applications: what's going on under the hood? In *Proceedings of the 2011*

ACM SIGPLAN X10 Workshop, X10 '11, pages 1:1–1:8, New York, NY, USA, 2011. ACM.

[23] 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.

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

[25] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *PODC*, pages 280–289, 2002.

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

[27] 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.

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

[29] 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)`.

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

[31] 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.

[32] 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.

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

[34] Michael Mitzenmacher. Analyses of load stealing models based on differential equations. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 212–221, New York, NY, USA, 1998. ACM.

[35] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, SPAA '91, pages 237–245, New York, NY, USA, 1991. ACM.

[36] Bratin Saha, Ali-Reza Adl-Tabatabai, Anwar Ghuloum, Mohan Rajagopalan, Richard L. Hudson, Leaf Petersen, Vijay Menon, Brian Murphy, Tatiana Shpeisman, Jesse Fang, Eric Sprangle, Anwar Rohillah, and Doug Carmean. Enabling scalability and performance in a large scale chip multiprocessor environment. *Technical Report. Intel Corp.*, 2006.

[37] 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.

[38] Peter Sanders. Randomized receiver initiated load-balancing algorithms for tree-shaped computations. *Comput. J.*, 45(5):561–573, 2002.

[39] Barry Tannenbaum. Miser - a dynamically loadable memory allocator for multi-threaded applications. *Intel Software Network*, 2009.

[40] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. In *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Proceedings, Part II*, volume 6507 of *Lecture Notes in Computer Science*, pages 291–302. Springer, 2010.

[41] Alexandros Tzannes. *Enhancing Productivity and Performance Portability of General-Purpose Parallel Programming*. PhD thesis, University of Maryland, 2012.

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

A. Proof of efficiency

Figure 7 summarizes the variables introduced in the proof.

P	number of processors
T_1	execution time with 1 processors (work)
T_∞	execution time with ∞ processors (depth)
T_P	execution time with P processors
F	maximal number of forks in a path of the computation DAG
δ	expected delay between two transfer attempts
s	a configuration of the algorithm
s_i	the initial configuration of the algorithm
s_e	the terminal 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'
$W(s)$	number of work tokens generated from s
$I(s)$	random variable: idle tokens generated from s
$K(s)$	random variable: communication tokens generated from s
Q	a deque of tasks
$Q_i(s)$	the deque of processor i in configuration s
u	a node from the computation DAG
$d(u)$	maximal length of a path starting from u
$f(u)$	maximal number of forks contained in a path starting from u
$b(u, Q)$	equal to 0 if u is at the bottom of deque Q, and 1 otherwise
$D(Q)$	maximum value of $d(u)$ for any $u \in Q$
$F(Q)$	maximum value of $f(u) + b(u, Q)$ for any $u \in Q$
$\Phi(Q)$	potential of deque Q
$\Phi(s)$	potential of configuration s
κ	a constant ≈ 1.68, which minimizes $\frac{\kappa}{1-2e^{-\kappa}}$
c	1.0 in sender-initiated, and $\frac{1}{1-1/e} \approx 1.58$ in receiver-initiated
μ	a shorthand for $\frac{1-2e^{-\kappa}}{c\delta}$
r	a shorthand for $\frac{P-1}{(1-1/\delta)\cdot(1-e^{-\mu})}$

Figure 7. Variables used in the proof of efficiency

Assumptions We say that a processor makes a *transfer attempt* when, in the sender-initiated algorithm, it makes a deal attempt, or, in the receiver-initiated algorithm, when it polls on its reception cell and possibly answer a steal request. We assume that a transfer attempt always completes in one time step, and that, at any given time step, a non-idle processor makes a transfer attempt happens with probability $\frac{1}{\delta}$, for some constant $\delta > 1$. For the receiver-initiated algorithm, we assume that, in one time step, an idle processor is able to send a request to a busy processor and receive a response from this busy processor in case it is polling for requests in the same time step. For the sender-initiated algorithm, we assume that, in one time step, a busy processor is able to query the state of the idle processor and to deliver a task to this processor.

Consider a unit computation DAG describing a binary fork-join computation. In this DAG, each node is uniquely identified and corresponds to a task that takes one time step to complete. Each edge indicates a dependency between two tasks. Let T_1 be the total work (number of nodes) and T_∞ be the total depth (maximal length of a path in the DAG). We call a node a *fork node* if its out-degree is two. We define the *branching depth*, written F, as the maximal number of fork nodes in a path contained in the DAG.

Configurations Consider an execution of the computation DAG using P processors. A *configuration*, written s, represents the state

of the processors during this execution. Concretely, it maps each processor to the list of nodes contained in its deque, in the deque order. In the initial configuration, called s_i, the initial task is in the deque of the processor 0, while all the other processors have empty deques. In the terminal configuration, called s_e, all the deques are empty. At each time step, the algorithm makes a transition from a configuration to another configuration (possibly the same). Let $\rho(s, s')$ denote the probability that, from configuration s, the algorithm makes a transition to the configuration s'. For any non-terminal configuration, the probabilities of the outgoing transitions add up to one, i.e.,

$$\forall s \neq s_e. \qquad \textstyle\sum_{s'} \rho(s, s') = 1. \tag{1}$$

Consider a particular computation DAG and let s_i denote the initial configuration of its execution. The *configuration graph* is a finite graph whose nodes correspond to the set of possible configurations reachable from s_i, and whose edges correspond to the possible transitions. More precisely, the configuration graph contains an edge from a configuration s to a configuration s' if s' is reachable in one time step from s, that is, if $\rho(s, s') > 0$. The configuration graph has s_i for source and s_e for sink. Each path in the configuration graph joining s_i to s_e describes one possible parallel execution of the computation DAG considered.

Tokens We analyse the execution time using tokens. At each time step, the algorithm makes a transition from a configuration to another, following an edge of the configuration graph. In a time step, P tokens are created: one *work token* is produced by each processor executing a task, one *communication token* is produced by each processor making a transfer attempt, and one *idle token* is produced by each idle processor. We introduce four variables to analyse the number of tokens produced during an execution path joining a given configuration s to the terminal configuration s_e.

- $W(s)$: the number of work tokens issued starting at configuration s. (Note that the amount of work does does not depend on the execution path followed.)
- $K(s)$: a random variable that denotes the number of communication tokens generated from s,
- $I(s)$: a random variable that denotes the number of idle tokens generated from s.
- $T_P(s)$: a random variable that denotes the length (in number of time steps) of the execution path taken from s.

Let $W(s)$ be the number of work tokens issued starting at configuration s. (Note that the amount of work does does not depend on the execution path followed.) Let $K(s)$ be a random variable that denotes the number of communication tokens generated from s, Let $I(s)$ be a random variable that denotes the number of idle tokens generated from s. Let $T_P(s)$ be a random variable that denotes the length (in number of time steps) of the execution path taken from s. Note that, once the terminal configuration has been reached, no more tokens are produced, therefore $W(s_e) = K(s_e) = I(s_e) = T_P(s_e) = 0$. Note also that, since the total amount of work is T_1, we have $W(s_i) = T_1$. We let T_P be a shorthand for $T_P(s_i)$, which is a random variable that denotes the parallel execution time.

Throughout the proof, we write $\alpha(s)$ to denote the number of processors with an empty deque at configuration s. The value $\alpha(s)$ corresponds to the number of token produced when making a transition starting from configuration s. Note that, for any non-terminal configuration s, we have $\alpha(s) \leq P - 1$ because at least one processor has a non-empty deque.

Analysis We bound the expected execution time with P processors, $\mathbb{E}[T_P] = \mathbb{E}[T_P(s_i)]$, in terms of the total work T_1, the expected number of communication tokens $\mathbb{E}[K(s_i)]$, and the expected number of idle tokens $\mathbb{E}[I(s_i)]$. Using the fact that communication is only performed by busy processors, we then show that $\mathbb{E}[K(s_i)]$ does not exceed a fraction of T_1. To bound $\mathbb{E}[I(s_i)]$, we introduce a potential function Φ, which maps each configuration s to a natural number $\Phi(s)$. The potential function is defined in such a way that the potential decreases along any execution path. Moreover, the potential decreases significantly either when a processor has a single task to work on or when a processor succeeds in dealing a task. We conduct an inductive proof to establish that the expected number of idle tokens issued from a configuration s does not exceed $r \cdot \ln \Phi(s)$, for some constant r. A key lemma used in this proof establishes that the potential decreases by a factor at least $\frac{\alpha(s)}{r}$ during a time step that starts in a configuration s where $\alpha(s)$ processors are idle. Once the inductive proof is completed, we are able to deduce a bound on $\mathbb{E}[I(s_i)]$. Combining all these results yields a bound on the expected execution time $\mathbb{E}[T_P]$.

LEMMA A.1.

$$\mathbb{E}[T_P] \;=\; \tfrac{T_1}{P} + \tfrac{1}{P}\,\mathbb{E}[K(s_i)] + \tfrac{1}{P}\,\mathbb{E}[I(s_i)]$$

Proof. Because P tokens are produced at each time step, the total number of tokens issued on a given execution path is equal to P times the length of this path. The number of tokens issued is also equal to the number of issued tokens of each of the three kinds. This reasoning applies to any given path. Therefore, for any configuration s, we have $\mathbb{E}[P \cdot T_P(s)] = \mathbb{E}[W(s) + K(s) + I(s)]$. In particular, for the initial configuration s_i, for which $W(s_i) = T_1$, we have $\mathbb{E}[P \cdot T_P(s_i)] = \mathbb{E}[T_1 + K(s_i) + I(s_i)]$. By linearity of expectations, we obtain the desired equality. □

LEMMA A.2.

$$\mathbb{E}[K(s_i)] \;=\; \tfrac{T_1}{\delta - 1}$$

Proof. Consider a configuration s. Recall that $K(s)$ is a random variable that denotes the number of communication tokens issued during an execution starting at s, that is, the number transfer attempts performed during an execution starting at s. The number of time steps executed by non-idle processors during an execution starting at s is described by the random variable $W(s) + K(s)$. During a time step, a busy processor makes a transfer attempt with probability $\frac{1}{\delta}$. Therefore, the random variable $K(s)$ follows a binomial distribution with parameters $n = (W(s) + K(s))$ and $p = \frac{1}{\delta}$. The expected value of $K(s)$ is therefore equal to $(W(s)+K(s)) \cdot \frac{1}{\delta}$. It follows that $\mathbb{E}[K(s)] = \mathbb{E}\left[(W(s) + K(s)) \cdot \frac{1}{\delta}\right]$. By using the fact that $W(s)$ is a constant and by rearranging the terms, we deduce $(\delta - 1) \cdot \mathbb{E}[K(s)] = W(s)$. We conclude by instantiating s as s_i and using $W(s_i) = T_1$. □

LEMMA A.3. *For any non-terminal configuration s,*

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

Proof. Consider an execution path starting at s. The expected number of idle tokens produced along this path is equal to the number $\alpha(s)$ of idle tokens produced on the first edge of this path, plus the expected number of tokens produced in the rest of the path, this number being computed as sum weighted by the appropriate probabilities as shown in the statement of the lemma. □

DEFINITION A.1 (Potential function).

The definition of the potential function involves a few auxiliary definitions. The *depth potential* of a node u, written $d(u)$, is defined as T_∞ (the total depth) minus the minimal length of a path that reaches the node u from the root node (in the unit-cost DAG).

The *fork potential* of a node u, written $f(u)$, is defined as F (the total branching depth) minus the minimal number of fork nodes in a path that reaches the node u from the root node. Given a task u and a deque s, we let $b(u, Q)$ be equal to 0 if u is at the bottom of Q and to 1 otherwise. We let c be 1.0 in sender-initiated algorithm and $\frac{1}{1-1/e} \approx 1.58$ in receiver-initiated algorithm. We define $\mu = \frac{1-2e^{-\kappa}}{c\delta}$ and κ to be any constant such that $\kappa > \ln 2$, which ensures $\mu > 0$. We define the potential $\Phi(Q)$ of a deque Q as follows.

$$\begin{array}{rrcl} & \Phi(Q) & \equiv & \begin{cases} 0 & \text{if } Q \text{ is empty} \\ \mathrm{e}^{\mu D(Q) + \kappa F(Q)} & \text{otherwise} \end{cases} \\ \text{where} & D(Q) & \equiv & \max_{u \in Q} \; d(u) \\ & F(Q) & \equiv & \max_{u \in Q} \; f(u) + b(u, Q) \end{array}$$

We then define the potential $\Phi(s)$ of a configuration s as the sum of the potential of all the deques, i.e. $\Phi(s) = \sum_{i \in I} \Phi(Q_i(s))$, where I denotes the set of all processors and where $Q_i(s)$ denotes the deque of processor i in the configuration s. Note that a potential is always greater or equal to zero and that the terminal configuration s_e is the only configuration whose potential is equal to zero.

Our algorithm follows the same *deque discipline* as in work stealing: processors push and pop tasks at the bottom of their deque when working locally, and migrate tasks taken from the top of their deque.

LEMMA A.4 (Deque discipline). *If u and u' are two nodes contained in a same deque Q in such a way that u is located above u', then $d(u') \leq d(u) - 1$ and $f(u') \leq f(u) - 1$.*

Proof. This is a standard property of work stealing deques. Intuitively, at any given time, the tasks stored in the deque correspond to the right branches of a list of consecutive fork nodes taken from one path of the computation DAG. A detailed proof can be found in [2]. □

LEMMA A.5 (Decrease in potential). *The total potential never increases: if $\rho(s, s') > 0$ then $\Phi(s') \leq \Phi(s)$.*

Proof. We establish this result by showing (1) that the potential of the deque of a processor never increases when the processor work on his deque, and (2) that, after a task migration, the potential of the deque of the sender plus the potential of the deque of the receiver is less than the initial potential of the deque of the sender. To prove (1), we observe that the depth and the branching depth of a task are always smaller than that of its parent task, the potential of a deque never increases, and that the value of $b(u, Q)$ can only change from 1 to 0, when the task u reaches the bottom of the deque.

To prove (2), we consider a task being migrated from the top of a deque Q towards an empty deque. Let u denote this task, let Q' denote the state of deque Q after u has been removed, and let Q'' denotes the deque made of the task u alone, i.e., $Q'' = \{u\}$. Our goal is to show that $\Phi(Q') + \Phi(Q'')$ is less than $\Phi(Q)$. Since u is not at the bottom of Q (because a processor never sends away its last task), we have $b(u, Q) = 1$. By Lemma A.4, all the other tasks from Q have a fork potential strictly less than that of u. If follows that $F(Q) = f(u) + b(u, Q) = f(u) + 1$ and $F(Q') \leq F(Q) - 1$. Besides, because Q' is a subset of Q, we have $D(Q') \leq D(Q)$. From these last two results, we derive $\Phi(Q') \leq \mathrm{e}^{-\kappa}\Phi(Q)$, The idle processor that receives the task u had an empty deque. It now has a deque, call it Q'', made of the task u alone. By definition, $F(Q'') = f(u) + b(u, Q'')$. Because u sits at the bottom of the singleton deque Q'', we have $b(u, Q'') = 0$. Combining these results with the equality $F(Q) = f(u) + 1$ which we derived earlier on, we deduce $F(Q'') = F(Q) - 1$. Besides, because Q'' is a subset of Q, we have $D(Q'') \leq D(Q)$. From these last two results, we obtain $\Phi(Q'') \leq \mathrm{e}^{-\kappa}\Phi(Q)$. Combining $\Phi(Q') \leq \mathrm{e}^{-\kappa}\Phi(Q)$ and $\Phi(Q'') \leq \mathrm{e}^{-\kappa}\Phi(Q)$ gives $\Phi(Q') + \Phi(Q'') \leq 2\mathrm{e}^{-\kappa}\Phi(Q)$. Under the assumption $\kappa \geq \ln 2$, we have $2\mathrm{e}^{-\kappa} \leq 1$, therefore $\Phi(Q') + \Phi(Q'')$ is less than $\Phi(Q)$. □

LEMMA A.6. *Consider a configuration s, where $\alpha(s)$ processors are idle. Consider a busy processor with more than one task in its deque in this configuration. If this processor makes a transfer attempt, then a task migration happens with probability at least $\frac{\alpha(s)}{P-1} \cdot \frac{1}{c} \cdot (1 - \frac{1}{\delta})$.*

Proof. Case 1: sender-initiated algorithm. In this case $\frac{1}{c} = 1$. Since there are $\alpha(s)$ idle processors, the busy processor targets an idle processor with probability $\frac{\alpha(s)}{P-1}$. The deal attempt to this target succeeds unless another busy processor targets the same target. Let us prove that the probability of such a conflict occurs with probability no more than $\frac{1}{\delta}$. Consider another busy processor. It initiates a deal attempt with probability $\frac{1}{\delta}$ and picks the same target with probability $\frac{1}{P-1}$. Therefore, the probability of conflicting with this particular other processor is no more than $\frac{1}{\delta(P-1)}$. Because there are at most $P - 1$ other busy processors, the probability of conflicting with any other processor is no more than $(P - 1) \cdot \frac{1}{\delta(P-1)}$, which is equal to $\frac{1}{\delta}$. Therefore, the probability no conflict occuring is $1 - \frac{1}{\delta}$. This shows that the probability of a successful deal is at least $\frac{\alpha(s)}{P-1} \cdot (1 - \frac{1}{\delta})$.

Case 2: receiver-initiated algorithm. In this case $\frac{1}{c} = 1 - \frac{1}{e}$. A task migration takes place if and only if the busy processor considered has received a steal request. To prove the lemma, since $(1 - \frac{1}{\delta}) < 1$, it suffices to show that the probability for a busy processor to have received a request is at least $\frac{\alpha(s)}{P-1} \cdot (1 - \frac{1}{e})$. Each of the $\alpha(s)$ processors have made a steal request, either at this time step or at an earlier time step. Consider one particular idle processor. Since it has chosen its target at random, the probability that it has aimed at the busy processor that we consider is $\frac{1}{P-1}$. So, the probability that it does not aim at it is $1 - 1/(P - 1)$. Since all the idle processors act independently, the probability that none of the idle processor picks the busy processor considered is $(1 - 1/(P - 1))^{\alpha(s)}$. It follows that the probability for the busy processor to have received at least one steal request from an idle processor is $1 - (1 - 1/(P - 1))^{\alpha(s)}$. To conclude the proof, it therefore suffices to show $1 - (1 - 1/(P - 1))^{\alpha(s)} \geq (1 - \frac{1}{e}) \cdot \frac{\alpha(s)}{P-1}$. This inequality follows from a purely-mathematical analysis, which can be found in the appendix. □

DEFINITION A.2 (Transfered potential).

Let s and s' be two configurations such that $\rho(s, s') > 0$. Let i be a busy processor in s. We define:

$$\Delta_i(s, s') \equiv \begin{cases} 0 & \text{if } i \text{ sends no task during the transition from } s \text{ to } s' \\ \Phi(\{u\}) & \text{if } u \text{ is the unique task sent by } i \text{ in this transition.} \end{cases}$$

LEMMA A.7. *Let r be equal to $\frac{P-1}{(1-1/\delta)\cdot(1-e^{-\mu})}$. Let s be a non-terminal configuration, and let i be the index of a busy processor in s. Then,*

$$\sum_{s'} \rho(s, s') \cdot \frac{\Phi(Q_i(s)) - \Phi(Q_i(s')) - \Delta_i(s, s')}{\Phi(Q_i(s))} \geq \frac{\alpha(s)}{r}.$$

Proof. For simplicity, we write Q as a shorthand for $Q_i(s)$, we write Q' for $Q_i(s')$, we write Δ for $\Delta_i(s, s')$, and we write Q'' for the singleton deque $\{u\}$. Our goal is to bound the expected value of the relative decrease in potential $\frac{\Phi(Q)-\Phi(Q')-\Delta}{\Phi(Q)}$. We distinguish two cases, depending on whether the busy processor has exactly

one task or more than one task. (Additional details can be found in the full version.)

Case 1: processor with exactly one task. In this case, we assume that Q contains a single task u. We want to establish a lower bound for the expected decrease in potential.

Sub-case 1a: transfer attempt. With probability $\frac{1}{\delta}$, the processor makes a transfer attempt. Because the processor has only one task, it does not send it away. Therefore, $Q' = Q$ and $\Delta = 0$. In this case, the relative decrease in potential is thus equal to zero.

Sub-case 1b: no transfer attempt. With probability $1 - \frac{1}{\delta}$, the task u will be executed. The execution of u produces zero, one or two new tasks, which constitute the content of the new deque Q'. All the tasks created by u have a depth potential less than that of u. So, $D(Q') \leq D(Q) - 1$. Moreover, because the fork potential never increases along a path from the computation DAG, we have $F(Q') \leq F(Q)$. By definition of the potential, we have $\Phi(Q) = \mathrm{e}^{\mu D(Q) + \kappa F(Q)}$ and $\Phi(Q') = \mathrm{e}^{\mu D(Q') + \kappa F(Q')}$. We derive the inequality $\Phi(Q') \leq \mathrm{e}^{-\mu}\Phi(Q)$, which implies $\frac{\Phi(Q)-\Phi(Q')}{\Phi(Q)} \geq 1 - \mathrm{e}^{-\mu}$.

Summary of case 1: The relative decreases in potential exceeds $1-\mathrm{e}^{-\mu}$ with probability $1-\frac{1}{\delta}$, and is zero otherwise. Therefore, the expected decrease in potential for a busy processor whose deque contains a single task exceeds $(1 - \frac{1}{\delta}) \cdot (1 - \mathrm{e}^{-\mu})$. Because $\alpha(s) \leq P - 1$, the value $\frac{\alpha(s)}{P-1} \cdot (1 - \frac{1}{\delta}) \cdot (1 - \mathrm{e}^{-\mu})$ is also a valid lower bound. This value can be rewritten in the form $\frac{\alpha(s)}{r}$. Thus, for a busy processor whose deque contains a single task, the expected relative decrease in potential after one time step exceeds $\frac{\alpha(s)}{r}$.

Case 2: processor with more than one task.

There are three cases.

Sub-case 2a: successful task transfer. With probability $\frac{1}{\delta}$, the busy processor considered is making a transfer attempt. By Lemma A.6, the transfer attempt results in a task migration with probability more than $\frac{\alpha(s)}{P-1} \cdot \frac{1}{c} \cdot (1 - \frac{1}{\delta})$. Therefore, the probability for the busy processor considered to make a transfer attempt and successfully migrate a task is $\frac{1}{\delta} \cdot \frac{\alpha(s)}{P-1} \cdot \frac{1}{c} \cdot (1 - \frac{1}{\delta})$. Let us now bound the expected decrease in potential associated with such a successful deal. As shown in the proof of Lemma A.5, if a task u is migrated from the top of a deque Q, leaving a deque Q', and if Q'' denotes the deque made of the task u alone, then we have: $\Phi(Q') + \Phi(Q'') \leq 2\mathrm{e}^{-\kappa}\Phi(Q)$. Since the amount of potential transfered, Δ, is equal to $\Phi(Q'')$, we have $\frac{\Phi(Q)-\Phi(Q')-\Delta}{\Phi(Q)} \geq 1 - 2\mathrm{e}^{-\kappa}$.

Sub-case 2b: unsuccessful task transfer. In this case, the potential of the deque of the processor considered does not change. In other words, the relative decrease in potential is zero.

Sub-case 2c: no transfer attempt. In this case, the processor works locally on its deque and the potential of the deque can only decrease, due to the structural properties of the deque. In other words, the relative decrease in potential is at least zero.

Summary of case 2: With probability $\frac{1}{\delta} \cdot \frac{\alpha(s)}{P-1} \cdot \frac{1}{c} \cdot (1 - \frac{1}{\delta})$, the relative decrease in potential is at least $1 - 2\mathrm{e}^{-\kappa}$, and in all other cases the relative decrease in potential is at least zero. Therefore, the expected decrease in potential exceeds $\frac{1}{\delta} \cdot \frac{\alpha(s)}{P-1} \cdot \frac{1}{c} \cdot (1 - \frac{1}{\delta}) \cdot (1 - 2\mathrm{e}^{-\kappa})$. We define $\mu = \frac{1-2\mathrm{e}^{-\kappa}}{c\delta}$. Recall that $r = \frac{P-1}{(1-\frac{1}{\delta})\cdot(1-e^{-\mu})}$. The bound $\frac{1}{\delta} \cdot \frac{\alpha(s)}{P-1} \cdot \frac{1}{c} \cdot (1 - \frac{1}{\delta}) \cdot (1 - 2\mathrm{e}^{-\kappa})$ can be rewriten as $\frac{\mu}{1-e^{-\mu}} \cdot \frac{\alpha(s)}{r}$. We conclude using $\frac{\mu}{1-e^{-\mu}} \geq 1$. Considering the mathematical inequality $e^x \geq 1 + x$ and instantiating x with $-\mu$ shows $\frac{\mu}{1-e^{-\mu}} \geq 1$. Therefore, we conclude that, for a busy processor whose deque contains more than one task, the expected relative decrease in potential $\frac{\Phi(Q)-\Phi(Q')-\Delta}{\Phi(Q)}$ exceeds $\frac{\alpha(s)}{r}$. □

LEMMA A.8. *For any non-terminal configuration s,*

$$\sum_{s'} \rho(s, s') \cdot \frac{\Phi(s) - \Phi(s')}{\Phi(s)} \;\geq\; \frac{\alpha(s)}{r}.$$

Proof. Let B denote the set of busy processors in s, that is $B = \{i \in I \,|\, Q_i(s) \neq \emptyset\}$. By Lemma A.7, for any $i \in B$, we have

$$\textstyle\sum_{s'} \rho(s, s') \cdot (\Phi(Q_i(s)) - \Phi(Q_i(s')) - \Delta_i(s, s')) \geq \frac{\alpha(s)}{r} \cdot \Phi(Q_i(s))$$

Summing up over all $i \in B$ gives:

$$\begin{aligned} &\textstyle\sum_{i\in B} \cdot \sum_{s'} \rho(s, s') \cdot (\Phi(Q_i(s)) - \Phi(Q_i(s')) - \Delta_i(s, s')) \\ \geq\; &\textstyle\sum_{i\in B} \cdot \frac{\alpha(s)}{r} \cdot \Phi(Q_i(s)) \end{aligned}$$

By rewriting, we obtain:

$$\begin{aligned} &\textstyle\sum_{s'} \rho(s, s') \cdot \left(\sum_{i\in B} \Phi(Q_i(s)) - \sum_{i\in B} (\Phi(Q_i(s')) - \Delta_i(s, s'))\right) \\ \geq\; &\textstyle\frac{\alpha(s)}{r} \cdot \sum_{i\in B} \Phi(Q_i(s)). \end{aligned} \tag{2}$$

On the one hand, we have $\Phi(s) = \sum_{i\in I} \Phi(Q_i(s)) = \sum_{i\in B} \Phi(Q_i(s))$, because idle processor have an empty deque whose potential is equal to zero. On the other hand, we have $\Phi(s') = \sum_{i\in I} \Phi(Q_i(s')) = \sum_{i\in B} \Phi(Q_i(s')) + \sum_{i\notin B} \Phi(Q_i(s'))$. Because the set of tasks received by idle processors is equal to the set of tasks sent by busy processors, and because an idle processor can receive at most one task during one time step, we have $\sum_{i\notin B} \Phi(Q_i(s')) = \sum_{i\in B} \Delta_i(s, s')$. We deduce $\Phi(s') = \sum_{i\in B} (\Phi(Q_i(s')) + \Delta_i(s, s'))$. Thanks to all these observations, the inequation (2) can be simplified as follows.

$$\textstyle\sum_{s'} \rho(s, s') \cdot (\Phi(s) - \Phi(s')) \;\geq\; \frac{\alpha(s)}{r} \cdot \Phi(s)$$

The result of the lemma then follows immediately. □

LEMMA A.9. *For any non-terminal configuration s,*

$$\alpha(s) \;\leq\; r \cdot \sum_{s' \neq s_e} \rho(s, s') \ln \frac{\Phi(s)}{\Phi(s')}.$$

Proof. Consider a non-terminal configuration s. By Lemma A.8, we have $\sum_{s'} \rho(s, s') \cdot (1 - \frac{\Phi(s')}{\Phi(s)}) \geq \frac{\alpha(s)}{r}$. Thus, $\sum_{s'} \rho(s, s') - \sum_{s'} \rho(s, s') \frac{\Phi(s')}{\Phi(s)} \geq \frac{\alpha(s)}{r}$. By equality (1), $\sum_{s'} \rho(s, s')$ is equal to 1. Moreover, $\Phi(s_e) = 0$. Therefore,

$$\textstyle\sum_{s' \neq s_e} \rho(s, s') \frac{\Phi(s')}{\Phi(s)} \;\leq\; 1 - \frac{\alpha(s)}{r}. \tag{3}$$

Before we can apply the logarithm function to both sides of this inequality, we need to justify that the values are positive. For the right-hand side, we have $\frac{\alpha(s)}{r} < 1$, because $\frac{\alpha(s)}{r}$ is equal to $\frac{\alpha(s)}{P-1} \cdot (1 - \frac{1}{\delta}) \cdot (1 - e^{-\mu})$, where $\frac{\alpha(s)}{P-1} \leq 1$ and $1 - \frac{1}{\delta} < 1$ and $1 - e^{-\mu} < 1$. For the left-hand side, consider a non-terminal configuration. There is at least one busy processor. With some probability, one of the busy processor makes a transfer attempt and therefore its deque is not empty a the next configuration. Therefore, $\sum_{s' \neq s_e} \rho(s, s') \frac{\Phi(s')}{\Phi(s)} > 0$. We are now able to apply the logarithm function to both sides of inequation (3).

$$\ln\left(\textstyle\sum_{s' \neq s_e} \rho(s, s') \frac{\Phi(s')}{\Phi(s)}\right) \;\leq\; \ln\left(1 - \frac{\alpha(s)}{r}\right) \tag{4}$$

On the left-hand side, we invoke the concavity of the logarithmic function to justify that the expectation of the logarithm is smaller than the logarithm of the expectation:

$$\textstyle\sum_{s' \neq s_e} \rho(s, s') \ln \frac{\Phi(s')}{\Phi(s)} \;\leq\; \ln\left(\sum_{s' \neq s_e} \rho(s, s') \frac{\Phi(s')}{\Phi(s)}\right). \tag{5}$$

On the right-hand side, we use the inequality

$$\ln\left(1 - \frac{\alpha(s)}{r}\right) \;\leq\; -\frac{\alpha(s)}{r} \tag{6}$$

which is an instance of $\ln(1-x) \leq -x$, which holds for any $x < 1$. Combining (4) with (5) and (6) shows $\sum_{s' \neq s_e} \rho(s,s') \ \ln \frac{\Phi(s')}{\Phi(s)} \leq -\frac{\alpha(s)}{r}$, which can be easily rearranged in the desired form. □

We have just bounded the expected relative decrease in potential. This result will help us prove the inequality $\mathbb{E}[I(s)] \leq r \cdot \ln \Phi(s)$. We conduct this proof by induction on a well-founded relation, written $\prec$. The relation $\prec$ is defined below, as a lexicographical order that first compares the amount of remaining work and then compares the number of idle processors.

DEFINITION A.3. *Partial order on configurations*

$$s' \prec s \ \equiv \ W(s') < W(s) \ \vee \ \big(W(s') = W(s) \ \wedge \ \alpha(s') < \alpha(s)\big)$$

LEMMA A.10. *For any configurations s and s',*

$$\rho(s,s') > 0 \quad \Rightarrow \quad s' \prec s \ \vee \ s' = s.$$

Proof. Consider a time step during which the algorithm makes a transition from a configuration s to a configuration s'. Let us prove that either $s' \prec s$ or $s' = s$. If, during this time step, the amount of work decreases strictly, then $W(s') < W(s)$ and therefore $s' \prec s$. Otherwise, if the amount of work does not decrease, i.e., $W(s') = W(s)$, then it means that all the busy processors have been performing deal attemps. If at least one of these attempts succeeds, then the number of idle processors decreases, so $\alpha(s') < \alpha(s)$ and therefore $s' \prec s$. Otherwise, if all the deal attempts are unsuccessful, then all the deques remain the same, in which case $s' = s$. □

LEMMA A.11. *For any non-terminal configuration s,*

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

Proof. We are going prove the inequality by induction on $\prec$. Consider a non-terminal configuration s. The induction hypothesis states that $\mathbb{E}[I(s')] \leq r \cdot \ln \Phi(s')$ holds for any $s' \prec s$. Our goal is to show $\mathbb{E}[I(s)] \leq r \cdot \ln \Phi(s)$. By Lemma A.3, $\mathbb{E}[I(s)] = \alpha(s) + \sum_{s' \neq s_e} \rho(s,s') \cdot \mathbb{E}[I(s')]$. Let us bound each of the two terms of this sum. On the one hand, by Lemma A.9, $\alpha(s) \leq r \cdot \sum_{s' \neq s_e} \rho(s,s') \cdot \ln \frac{\Phi(s)}{\Phi(s')}$. We can exclude the case $s' = s$ from this sum because $\ln \frac{\Phi(s)}{\Phi(s)} = 0$. Therefore, we have $\alpha(s) \leq r \cdot \sum_{s' \neq s, s_e} \rho(s,s') \cdot (\ln \Phi(s) - \ln \Phi(s'))$. On the other hand, the term $\sum_{s' \neq s_e} \rho(s,s') \cdot \mathbb{E}[I(s')]$ is equal to $\rho(s,s) \cdot \mathbb{E}[I(s)] + \sum_{s' \neq s, s_e} \rho(s,s') \cdot \mathbb{E}[I(s')]$. We have $\sum_{s' \neq s, s_e} \rho(s,s') \cdot \mathbb{E}[I(s')] = \sum_{s' \mid s' \neq s, s_e \wedge \rho(s,s') > 0} \rho(s,s') \cdot \mathbb{E}[I(s')]$, because the configuration s' such that $\rho(s,s') = 0$ do not contribute to the sum. Consider a configuration s' such that $s' \neq s, s_e$ and $\rho(s,s') > 0$. By Lemma A.10, we have $s' \prec s$. Using the induction hypothesis, we derive $\sum_{s' \neq s, s_e} \rho(s,s') \cdot \mathbb{E}[I(s')] \leq \sum_{s' \neq s, s_e} \rho(s,s') \cdot r \cdot \ln \Phi(s')$. We exploit the bounds on $\alpha(s)$ and on $\sum_{s' \neq s_e} \rho(s,s') \cdot \mathbb{E}[I(s')]$ to continue the proof as follows.

$$\begin{aligned}
\mathbb{E}[I(s)] \ &= \ \alpha(s) + \textstyle\sum_{s' \neq s_e} \rho(s,s') \cdot \mathbb{E}[I(s')] \\
&\leq \ r \cdot \textstyle\sum_{s' \neq s, s_e} \rho(s,s') \cdot (\ln \Phi(s) - \ln \Phi(s')) \\
&\quad + \rho(s,s) \cdot \mathbb{E}[I(s)] + \textstyle\sum_{s' \neq s, s_e} \rho(s,s') \cdot r \cdot \ln \Phi(s') \\
&\leq \ r \cdot \ln \Phi(s) \cdot \Big(\textstyle\sum_{s' \neq s, s_e} \rho(s,s')\Big) \ + \ \rho(s,s) \cdot \mathbb{E}[I(s)] \\
&\leq \ r \cdot \ln \Phi(s) \cdot (1 - \rho(s,s)) \ + \ \rho(s,s) \cdot \mathbb{E}[I(s)]
\end{aligned}$$

On the last line, we have used equality (1) to show $\sum_{s' \neq s, s_e} \rho(s,s') \leq \sum_{s' \neq s} \rho(s,s') = 1 - \rho(s,s)$. Rewriting the bound on $\mathbb{E}[I(s)]$ gives: $(1-\rho(s,s)) \cdot \mathbb{E}[I(s)] \leq (1-\rho(s,s)) \cdot r \cdot \ln \Phi(s)$. Because $\rho(s,s)$ cannot be equal to 1 (since $\frac{1}{\delta} < 1$), we have $1 - \rho(s,s) \neq 0$. Therefore, we can divide both sides by $1 - \rho(s,s)$ and conclude the proof. □

The proof is by induction on $(\prec)$. By Lemma A.3, $\mathbb{E}[I(s)] = \alpha(s) + \sum_{s' \neq s_e} \rho(s,s') \cdot \mathbb{E}[I(s')]$. Using Lemma A.9, we can show $\alpha(s) \leq r \cdot \sum_{s' \neq s, s_e} \rho(s,s') \cdot (\ln \Phi(s) - \ln \Phi(s'))$. Using Lemma A.10 and the induction hypothesis, we can derive $\sum_{s' \neq s, s_e} \rho(s,s') \cdot \mathbb{E}[I(s')] \leq \sum_{s' \neq s, s_e} \rho(s,s') \cdot r \cdot \ln \Phi(s')$. We are then able to prove $\mathbb{E}[I(s)] \leq r \cdot \ln \Phi(s) \cdot \Big(\sum_{s' \neq s, s_e} \rho(s,s')\Big) + \rho(s,s) \cdot \mathbb{E}[I(s)]$, from which we derive $(1 - \rho(s,s)) \cdot \mathbb{E}[I(s)] \leq (1 - \rho(s,s)) \cdot r \cdot \ln \Phi(s)$ and then conclude.

LEMMA A.12.

$$\mathbb{E}[I(s_i)] \ \leq \ \left(1 + \frac{1}{\delta - 1}\right) \cdot (P-1) \cdot \frac{\mu}{1 - e^{-\mu}} \cdot \left(T_\infty + \frac{\kappa}{\mu} F\right)$$

Proof. We apply Lemma A.11 to the initial configuration s_i. In this configuration, there is a single task u_0 placed in the deque of processor with index 0. Its potential is $\Phi(s_i) = \Phi(Q_0(s_i)) = \mathrm{e}^{\mu d(u_0) + \kappa f(u_0)} = \mathrm{e}^{\mu T_\infty + \kappa F}$. So, we have: $\mathbb{E}[I(s_i)] \leq r \cdot (\mu T_\infty + \kappa F)$. Unfolding the definition $r = \frac{P-1}{(1-\frac{1}{\delta}) \cdot (1 - e^{-\mu})}$ gives $\mathbb{E}[I(s_i)] \leq \frac{P-1}{1-\frac{1}{\delta}} \cdot \frac{\mu}{1-e^{-\mu}} \cdot \Big(T_\infty + \frac{\kappa}{\mu} F\Big)$. The conclusion follows from the observation that $\frac{1}{1-\frac{1}{\delta}} = \frac{\delta}{\delta - 1} = 1 + \frac{1}{\delta - 1}$. □

LEMMA A.13. *The expected parallel execution time, $\mathbb{E}[T_P]$, is less than*

$$\left(1 + \frac{1}{\delta - 1}\right) \cdot \left(\frac{T_1}{P} \ + \ \frac{P-1}{P} \cdot \frac{\mu}{1 - e^{-\mu}} \cdot (T_\infty + 2.68 \cdot c\delta F)\right).$$

Proof. The proof is as follows.

$$\begin{aligned}
\mathbb{E}[T_P] &= \tfrac{1}{P}\left(T_1 + \mathbb{E}[K(s_i)] + \mathbb{E}[I(s_i)]\right) \\
&\quad \text{by Lemma A.1} \\
&\leq \tfrac{1}{P}\left(T_1 + \tfrac{1}{\delta-1} T_1 \ + \ \left(1 + \tfrac{1}{\delta-1}\right) \tfrac{P-1}{1} \tfrac{\mu}{1-e^{-\mu}} \left(T_\infty + \tfrac{\kappa}{\mu} F\right)\right) \\
&\quad \text{by Lemma A.2 and Lemma A.12} \\
&\leq \left(1 + \tfrac{1}{\delta-1}\right) \cdot \left(\tfrac{T_1}{P} \ + \ \tfrac{\mu}{1-e^{-\mu}} \tfrac{P-1}{P} \left(T_\infty + \tfrac{\kappa}{1-2e^{-\kappa}} c\delta F\right)\right) \\
&\quad \text{rearranging terms and unfolding the definition of } \mu
\end{aligned}$$

At this point, we are still completely free to instantiate κ with any value such that $\kappa > \ln(2) \approx 0.69$. We choose the value of κ that minimizes the value of $\frac{\kappa}{1-2e^{-\kappa}}$. Numerical analysis shows that $\kappa \approx 1.67835$ is the optimal choice. For this value of κ, we have $1 - 2e^{-\kappa} \approx 0.626637$ and $\frac{\kappa}{1-2e^{-\kappa}} \approx 2.67835 < 2.68$. □

B. Mathematical lemma

LEMMA B.1. *For any q and n such that $1 \leq n \leq q$,*

$$1 - \left(1 - \frac{1}{q}\right)^n \geq \left(1 - \frac{1}{e}\right) \cdot \frac{n}{q}.$$

Proof. Consider the function $f(x) = 1 - \left(1 - \frac{1}{q}\right)^x - (1 - \frac{1}{e}) \cdot \frac{x}{q}$. Proving our goal is equivalent to showing $f(x) \geq 0$ on the domain $x \in [1, q]$. We proof separately the inequality for the cases $q = 1$ and $q = 2$. When $q = 1$, x can only be 1. We check $f(1) = 1 - (1 - \frac{1}{e}) = \frac{1}{e} \geq 0$. When $q = 2$, x can be either 1 or 2. We check $f(1) = 1 - \frac{1}{2} - (1 - \frac{1}{e}) \cdot \frac{1}{2} = \frac{1}{e} \geq 0$. We check $f(2) = 1 - (\frac{1}{2})^2 - (1 - \frac{1}{e}) = \frac{1}{e} - \frac{1}{4} \geq 0.11 \geq 0$.

Otherwise, we have $q \geq 3$ and our strategy is as follows: we show $f(1) \geq 0$ and $f(q) \geq 0$, we show $f'(1) \geq 0$ (meaning that f is increasing at $x = 1$) and $f''(x) < 0$ for any $x \in [1, q]$ (meaning that f is a concave function). From these analyses of the variations

```
// turn down any incoming steal request
void reject(int i)
  int j = r[i]
  if j == NO_REQU
    if not compare_and_swap(&r[i], NO_REQU,
                                   REQ_BLOCKED)
      reject(i) // recurse at most once
  else
    t[j] = null
    r[i] = REQ_BLOCKED

// called by workers when running out of work
void acquire(int i)
  reject(i)
  while true
    t[i] = NO_RESP
    int k = random in {0, .., P-1}\{i}
    if a[k] && compare_and_swap(&r[k], NO_REQU, i)
      while (t[i] == NO_RESP)
        noop
      if (t[i] != null)
        add_task(i, t[i])
        r[i] = NO_REQU
        return
```

Figure 8. Alternative acquire function for the receiver-initiated algorithm

of f, we can conclude that the continuous function f remains above zero on the range $[1, q]$.

We have $f(1) = 1 - (1 - \frac{1}{q}) - (1 - \frac{1}{e})\frac{1}{q} = \frac{1}{e} \geq 0$. We have $f(q) = 1 - (1 - \frac{1}{q})^q - (1 - \frac{1}{e})$. Using the standard mathematical inequality $(1 - \frac{1}{q})^q \leq \frac{1}{e}$, we have $f(q) \geq 1 - \frac{1}{e} - (1 - \frac{1}{e}) = 0$. We have $f'(x) = -\ln(1 - \frac{1}{q}) \left(1 - \frac{1}{q}\right)^x - (1 - \frac{1}{e}) \cdot \frac{1}{q}$. In particular, $f'(1) = -\ln(1 - \frac{1}{q}) \cdot (1 - \frac{1}{q}) - (1 - \frac{1}{e}) \cdot \frac{1}{q}$. Using the mathematical inequality $\ln(1 - a) \leq -a$, we have $f'(1) \geq \frac{1}{q} \cdot (1 - \frac{1}{q}) - (1 - \frac{1}{e}) \cdot \frac{1}{q} = \frac{1}{q} \cdot (\frac{1}{e} - \frac{1}{q})$. Since $q \geq 3 \geq e$, we can conclude $f'(1) \geq 0$. Finally, we have $f''(x) = -(\ln(1 - \frac{1}{q}))^2 \left(1 - \frac{1}{q}\right)^x < 0$. We have therefore established sufficient conditions for showing $f(x) \geq 0$ on the domain $x \in [1, q]$. □

C. Alternative acquire function for the receiver-initiated algorithm

Figure 8 shows an alternative implementation of `acquire` in the receiver-initiated algorithm. This version does not require calling `communicate` while idling.