Abstract

Work stealing has proven to be an effective method for scheduling parallel programs on multicore computers. To achieve high performance, work stealing distributes tasks between concurrent queues, called deques, which are 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

1. Introduction

As multicore computers (i.e., 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 implicit parallelism, as elegantly exemplified by languages such as Cilk [23], NESL [7], parallel Haskell [32], and parallel ML [22] has emerged as a promising technique for parallel programming [33]. In such languages, the programmer expresses all opportunities for parallelism and leaves it to the language and its run-time system to create and manage the parallel tasks needed to take advantage of the opportunities for parallel execution.

Since implicit parallel programs expose all opportunities for parallelism, they can create an overabundance of tasks, including tiny ones. Executing such fine-grained parallel programs with high-performance therefore 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., several thousands of 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 [9, 23] has emerged as an effective scheduler for implicitly parallel programs. (The idea of work stealing goes back to 80’s [11, 25].)

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 [10]. 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 [3] proposed the first such data structure for fixed-sized deques. Hendler et al [26] generalized that data structure to support unbounded deques; however, the algorithm could result in memory leaks. Chase and Lev [12] 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 randomized work stealing with concurrent queues 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, it can be important to be flexible in choosing the task(s) to transfer during a steal.
These studies show experimentally [14, 17, 27, 38, 39] and theoretically [4, 36, 37], 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 [27], where a steals transfers half of tasks from the victim’s deque, can be more effective [14, 17, 39] 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 [19, 23, 34]; for example, Frigo et al. found that Cilk’s work-stealing protocol spends half of its time executing the memory fence [23]. 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. [29] and Umatani et al. [44] 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 [27]; 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 using private deques [14].

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 in a concurrent fashion, the processor 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 polls its message queue on a regular basis.

This message-passing approach to work stealing has been receiving significant attention in multicore computers. In early work, Feeley [19] investigates the use of work stealing with private deques to accelerate task creation. Hendler et al. [28] use a private deque to implement a load distribution strategy for improved locality. Hiraishi et al. [29] and Umatani et al. [44] 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 [22]. Using simulation studies, Sanchez et al. [39] 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. 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. In implicitly parallel programs, such communication is naturally easy to support, because the scheduler is invoked frequently due to small task sizes. 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 we do not consider coarse-grain parallel programs in this paper, we report on some preliminary investigations on the use of interrupts to make our algorithms robust even with 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 notions 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 [2].

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 [?] benchmark suite. Using these benchmarks, we compare our algorithms with Cilk (more precisely, Cilk Plus [30]). 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 [12] 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. We show, in particular, how to apply our approach to solve a challenging graph-reachability problem and demonstrate good scaling with our implementation.

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. 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 calls the bottom
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)

// 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)
    communicate(i)
  if (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 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, ..} // transfer 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)
    communicate(i)
  if (t[i] != NO_RESP)
    communicate(i)
  if (t[i] != null)
    add_task(i, t[i])
  r[i] = NO_REQU
  return
communicate(i)

// attempt to deal a task to an idle processor
void deal_attempt(int i)
if now() > d[i]
deal_attempt(i)
if r then s[j] = pop_top(q[i])
if s[j] != null then return
b = (size(q[i]) > 0)
if b 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

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 [2], 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 \( \delta \). We will explain later why these dates are needed.

In the sender-initiated algorithm, when an idle processor calls the function \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{deal_attempt} with the following code:

```c
    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 in making the delay between two deal attempts follow a Poisson distribution with parameter \( \delta \), for some \( \delta \) larger than the typical duration of a task. With this approach, deal attempts take place on average slightly more than every \( \delta \), 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 \( \text{now}() \) at the end of the function \texttt{acquire}, meaning that we typically allow a processor that receives a task in \texttt{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 \( \delta \) denotes the average communication delay and \( F \) 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 attempts follow a Poisson distribution with parameter \( \delta \). This model is faithful to the behavior of the actual algorithm whenever \( \delta \) is larger than the duration of a few tasks. Note that the larger is \( \delta \) 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 \( \delta \): we assume that the interval between two polling operations made by a given processor follow a Poisson distribution with parameter \( \delta \). The parameter \( \delta \) 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 [8], later simplified and generalized by Arora, Blumofe, Plaxton in [3], is \( E[TP] \leq \frac{2}{\delta_{F}} + 32T_{\infty} \), where \( TP \) denotes the execution time with \( P \) processors, \( T_{F} \) denotes the sequential execution time, and \( T_{\infty} \) 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 \( \frac{1}{T_{F}} \) with probability greater than \( \frac{1}{2} \). Tchiboukdjian et al [42] tightened this bound to \( E[TP] \leq \frac{2}{\delta_{F}} + 3.65T_{\infty} \), 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 \( \frac{2}{\delta_{F}} + \frac{2}{\delta_{F}}T_{\infty} \), 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 \( \delta \) and which distinguishes the contribution of \( T_{\infty} \) 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:

\[
E[TP] \leq \frac{1}{\delta_{F}} \left( 1 + \frac{1}{\delta} \right) \left( T_{F} + T_{\infty} + O(\delta F) \right).
\]

The bound above includes a factor \( 1 + \frac{1}{\delta_{F}} \), 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 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 \( E[TP] \) involves a constant \( c \), defined as 1.0 in the sender-initiated algorithm and \( 1/\sqrt{1/2} \approx 1.58 \) in the receiver-initiated algorithm.
The Concur. desques (speedup) is 21.77 for the cilksort (exptintseq) algorithm. It also involves a factor $\frac{\mu}{1 - e^{-\mu}}$, where $\mu$ is defined as $0.62$. Since $\frac{\mu}{1 - e^{-\mu}} \approx 1 + \frac{\mu}{4} \approx 1 + \frac{0.62}{4}$, the factor $\frac{\mu}{1 - e^{-\mu}}$ can be approximated as 1.0 for any practical purpose. The formal bound is:

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

The fact that $\delta$ 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 associated with a smaller value for $\delta$. 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 [2].

### 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 [12]. 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 [23]; 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 Billebo et al.’s problem-based benchmark suite (PBBS) [6], 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 [41]), 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 partitioning 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 [31] 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 $\delta$ 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 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. [3], which went through a few revisions due to concurrency bugs. Several years later, the nonblock-
An efficiency algorithm was extended by Chase and Lev to support dynamic resizing [12]. Their first and, to our knowledge, only proof of correctness of a nonblocking work-stealing algorithm is not trivial: it spans over thirty pages [13]. Moreover, there is, in the literature, no nonblocking algorithm which combines resizeability with other extensions, such as steal half, possibly owing to the complication 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 $\delta$ in the sender-initiated algorithm.

In the evaluation, we have been setting $\delta$ 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 $\delta$ needs to remain relatively small in order to efficiently distribute tasks. However, in benchmark where the branching depth is small, the value of $\delta$ can be safely increased without noticeably affecting on the execution time. According to our theorem, it is perfectly fine to use any $\delta$ such that $2.68 \cdot \delta F < T_1$. 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 $P = 30$, $\frac{T_1}{\delta F/P} = 23$ 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 $\delta$ 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 $\delta$, and find the idle processor with probability $\frac{1}{P}$. As a result, the expected time before a processor receives some work is exactly $\delta$. The second chart in Figure 5 confirms that the average idle time per task migration is indeed extremely close to $\delta$. 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 $\delta$.

Handling of large sequential tasks

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 [20, 21], 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 counter. For now, we have instead been using a more basic approach that consists in running an additional thread 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. Furthermore, 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, $\delta$ 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 coarse-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 accommodate 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 [4, 14, 17, 36, 37, 39]. 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 [27]. 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.

```c
// 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 transferred 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. To see why, 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 [14] suggest the following policy: if the number of nodes in the batch of the currently-running task is about to exceed \( \min(Q, S) \), 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 approach 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 graphs that Blelloch et. al. [6] used to benchmark their BFS program: 3d-grid (40m nodes, \( T_1 = 11.5s \)), mg (40m nodes, 150m edges, \( T_1 = 12.8s \)), and rmat (40m nodes, 90 edges, \( T_1 = 9.1s \)). 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 [40] analyzes a receiver-initiated load-balancing algorithm for a subclass of tree-shaped computations, presenting 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. [18] 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 [35], and different job scheduling policies [15]. More recently Dinan et al [16] 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 pre-emption 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 [24]. 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 [43]. 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.)

The work presented in this paper was partly motivated by a desire to design scheduling algorithms for improved data locality (e.g., [1, 5]), which we expect to be a promising research direction.

7. Conclusion

In this paper, we design, analyze, and empirically evaluate two work-stealing algorithms for executing implicitly parallel programs on modern multicore architectures. Both algorithms use private, non-concurrent dequeues to store parallel tasks and rely on explicit communication for load balancing. Our analysis shows the algorithms to be competitive with 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 accelerate irregular problems. As a challenge benchmark, we consider a challenge benchmark, we consider depth first search and obtain an efficient multithreaded runtime system. In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 207–216, 1995.

Acknowledgments

We would like to thank the anonymous referees, editor, and Alexander Tzannes for their helpful comments and suggestions.

7. Conclusion

In this paper, we design, analyze, and empirically evaluate two work-stealing algorithms for executing implicitly parallel programs on modern multicore architectures. Both algorithms use private, non-concurrent dequeues to store parallel tasks and rely on explicit communication for load balancing. Our analysis shows the algorithms to be competitive with 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 accelerate irregular problems. As a challenge benchmark, we consider depth first search and obtain an encouraging results.

Acknowledgments

We would like to thank the anonymous referees, editor, and Alexander Tzannes for their helpful comments and suggestions.

7. Conclusion

In this paper, we design, analyze, and empirically evaluate two work-stealing algorithms for executing implicitly parallel programs on modern multicore architectures. Both algorithms use private, non-concurrent dequeues to store parallel tasks and rely on explicit communication for load balancing. Our analysis shows the algorithms to be competitive with 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 accelerate irregular problems. As a challenge benchmark, we consider depth first search and obtain encouraging results.


A. Proof of efficiency

Figure 7 summarizes the variables introduced in the proof.

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

**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 \( \frac{\delta}{2} \), 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 \( T_{\infty} \) 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.,

\[ \forall s \neq s_e, \quad \sum_{s'} \rho(s,s') = 1. \]

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 \( s \). 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 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 not depend on the execution path followed.) Let \( K(s) \) be 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 \( s \). 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 tokens 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 \( \Phi \), 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 \( \alpha(s) \) 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 \( E[I(s)] \). Combining all these results yields a bound on the expected execution time \( E[T_P] \).

**Lemma A.1.**

\[
E[T_P] = \frac{\tau}{p} + \frac{1}{p} E[K(s_i)] + \frac{1}{p} E[I(s_i)]
\]

**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 \( E[P \cdot T_P(s)] = E[W(s) + K(s) + I(s)] \). In particular, for the initial configuration \( s_i \), for which \( W(s_i) = T_i \), we have \( E[P \cdot T_P(s_i)] = E[T_i + K(s_i) + I(s_i)] \). By linearity of expectations, we obtain the desired equality. \( \square \)

**Lemma A.2.**

\[
E[K(s_i)] = \frac{\tau}{p-1}
\]

**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 \( \frac{1}{p} \). Therefore, the random variable \( K(s) \) follows a binomial distribution with parameters \( n = (W(s) + K(s)) \) and \( p = \frac{1}{2} \). The expected value of \( K(s) \) is therefore equal to \( \frac{1}{2} \cdot (W(s) + K(s)) \). It follows that \( E[K(s_i)] = E[(W(s) + K(s)) \cdot \frac{1}{2}] \). By using the fact that \( W(s) \) is a constant and by rearranging the terms, we deduce \( (\delta - 1) \cdot E[K(s_i)] = W(s) \). We conclude by instantiating \( s \) as \( s_i \) and using \( W(s_i) = T_i \). \( \square \)

**Lemma A.3.** For any non-terminal configuration \( s \),

\[
E[I(s)] = \alpha(s) + \sum_{s'} \rho(s, s') \cdot E[I(s')]\]

**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. \( \square \)

**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 \( T_{\infty} \) (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 \( \frac{1}{1+c} \approx 1.58 \) in receiver-initiated algorithm. We define \( \mu = \frac{1-2e^{-c}}{2c} \) and \( \kappa \) 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.

\[
\Phi(Q) = \begin{cases} 
0 & \text{if } Q \text{ is empty} \\
 e^{-\mu D(Q)} + \frac{\kappa}{e} F(Q) & \text{otherwise} 
\end{cases}
\]

where

\[
D(Q) = \max_{u \in Q} d(u) \\
F(Q) = \max_{u \in Q} f(u) + b(u, Q) 
\]

We then define the potential \( \Phi(s) \) of a configuration \( s \) as the sum of the potential of all the deque, i.e., \( \Phi(s) = \sum_{Q \subseteq I} \Phi(Q(s)) \), 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 \( \alpha(s) \) 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 [3].

**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 works 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 \( \Phi(Q') \leq e^{-\mu} \Phi(Q) \). 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 \( \Phi(Q') \leq e^{-\mu} \Phi(Q) \). Combining \( \Phi(Q') \leq e^{-\mu} \Phi(Q) \) and \( \Phi(Q'') \leq e^{-\mu} \Phi(Q) \) gives \( \Phi(Q') + \Phi(Q'') \leq 2e^{-\mu} \Phi(Q) \). Under the assumption \( \kappa \geq \ln 2 \), we have \( 2e^{-\mu} \leq 1 \), 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 \( \frac{\alpha(s)}{r} \).

**Definition A.2** (Transferred potential).

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

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

We define \( \Delta_i(s, s') \) to be equal to: 0 if \( i \) sends no task during the transition from \( s \) to \( s' \), and to \( \Phi(u) \) if \( u \) is the unique task sent by \( i \) in this transition.

**Lemma A.7.** Let \( r \) be equal to \( \frac{P_i}{\rho_i - 1} \). 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.** Let \( B \) denote the set of busy processors in \( s \), that is \( B = \{ i \in I \mid Q_i(s) \neq \emptyset \} \). By Lemma A.7, for any \( i \in B \), we have

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

Summing up over all \( i \in B \) gives:

\[
\sum_{i \in B} \sum_{s'} \rho(s, s') \cdot (\Phi(Q_i(s)) - \Phi(Q_i(s')) - \Delta_i(s, s')) \geq \sum_{i \in B} \frac{\alpha(s)}{r} \cdot \Phi(Q_i(s)).
\]

By rewriting, we obtain:

\[
\sum_{s'} \rho(s, s') \cdot \sum_{i \in B} \Phi(Q_i(s)) - \sum_{i \in B} (\Phi(Q_i(s')) - \Delta_i(s, s')) \geq \frac{\alpha(s)}{r} \cdot \sum_{i \in B} \Phi(Q_i(s)).
\]

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 \( \Phi(s') = \sum_{i \in I} \Phi(Q_i(s')) = \sum_{i \in B} \Phi(Q_i(s')) + \sum_{i \in B} \Phi(Q_i(s)) \). 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 \( \sum_{i \in B} \Phi(Q_i(s')) = \sum_{i \in B} \Delta_i(s, s') \). 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:

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

The result of the lemma then follows immediately. \( \square \)

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

\[
\sum_{s' \neq s_s} \rho(s, s') \Phi(s') \frac{\Phi(s)}{\Phi(s')} \leq 1 - \frac{\alpha(s)}{r}.
\]

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 \( \frac{\alpha(s)}{r} \cdot (1 - \frac{1}{r}) \cdot (1 - e^{-rn}) \), where \( \frac{\alpha(s)}{r} \leq 1 \) and \( 1 - e^{-rn} < 1 \). 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, \( \sum_{s' \neq s_s} \rho(s, s') \Phi(s') \frac{\Phi(s)}{\Phi(s')} > 0 \). We are now able to apply the logarithm function to both sides of inequation (3).

\[
\ln \left( \sum_{s' \neq s_s} \rho(s, s') \Phi(s') \frac{\Phi(s)}{\Phi(s')} \right) \leq \ln \left( 1 - \frac{\alpha(s)}{r} \right)
\]

(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:

\[
\sum_{s' \neq s_s} \rho(s, s') \Phi(s') \frac{\Phi(s)}{\Phi(s')} \leq \ln \left( \sum_{s' \neq s_s} \rho(s, s') \Phi(s') \frac{\Phi(s)}{\Phi(s')} \right).
\]

(5)

On the right-hand side, we use the inequality

\[
\ln \left( 1 - \frac{\alpha(s)}{r} \right) \leq - \frac{\alpha(s)}{r}
\]

(6)

which is an instance of \( \ln(1-x) \leq -x \), which holds for any \( x < 1 \). Combining (4) with (5) and (6) shows

\[
\sum_{s' \neq s_s} \rho(s, s') \Phi(s') \frac{\Phi(s)}{\Phi(s')} \leq - \frac{\alpha(s)}{r}
\]

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

\( \prec \) is defined as

\[
s' \prec s \equiv W(s') < W(s) \lor (W(s') = W(s) \land \alpha(s') < \alpha(s))
\]

**Lemma A.10.** For any configurations \( s \) and \( s' \),

\[
\rho(s, s') > 0 \Rightarrow s' \prec s \lor s' = s.
\]

**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 attempts. 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 \( s' = s \). \( \square \)

**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, \( \mathbb{E}[I(s)] = \alpha(s) + \sum_{s' \neq s_s} \rho(s, s') \cdot \mathbb{E}[I(s')] \). Let us bound each of the two terms of this sum. On the one hand, by Lemma A.9, \( \alpha(s) \leq r \cdot \sum_{s' \neq s_s} \rho(s, s') \cdot \ln \frac{\Phi(s)}{\Phi(s')} \). We can exclude the case \( s' = s \) from this sum because \( \ln \frac{\Phi(s)}{\Phi(s')} = 0 \). Therefore, we have

\[
\alpha(s) \leq r \cdot \sum_{s' \neq s_s} \rho(s, s') \cdot (\ln \Phi(s) - \ln \Phi(s')).
\]

On the other
hand, the term \( \sum_{s' \neq s} \rho(s, s') \cdot E[I(s')] \) is equal to \( \rho(s, s) \cdot E[I(s)] + \sum_{s' \neq s} \rho(s, s') \cdot E[I(s')] \). We have \( \sum_{s' \neq s} \rho(s, s') \cdot E[I(s')] \) because the configuration \( s' \) such that \( \rho(s, s') = 0 \) do not contribute to the sum. Consider a configuration \( s' \) such that \( s' \neq s, s \), and \( \rho(s, s') > 0 \). By Lemma A.10, we have \( s' < s \). Using the induction hypothesis, we derive \( \sum_{s' \neq s} \rho(s, s') \cdot E[I(s')] \leq \sum_{s' \neq s} \rho(s, s') \cdot \ln \Phi(s) \). We exploit the bounds on \( \alpha(s) \) and on \( \sum_{s' \neq s} \rho(s, s') \cdot E[I(s')] \) to continue the proof as follows.

\[
\begin{align*}
E[I(s)] &= \alpha(s) + \sum_{s' \neq s} \rho(s, s') \cdot E[I(s')] \\
&\leq r \cdot \sum_{s' \neq s} \rho(s, s') \cdot (\ln \Phi(s) - \ln \Phi(s')) \\
&\quad + \rho(s, s) \cdot E[I(s)] + \sum_{s' \neq s} \rho(s, s') \cdot r \cdot \ln \Phi(s') \\
&\leq r \cdot \ln \Phi(s) \cdot \left( \sum_{s' \neq s} \rho(s, s') \right) + \rho(s, s) \cdot E[I(s)] \\
&\leq r \cdot \ln \Phi(s) \left( 1 - \rho(s, s) \right) + \rho(s, s) \cdot E[I(s)]
\end{align*}
\]

On the last line, we have used equality (1) to show \( \sum_{s' \neq s} \rho(s, s') = 1 - \rho(s, s) \). Rewriting the bound on \( E[I(s)] \) gives: \( (1 - \rho(s, s)) \cdot E[I(s)] \leq (1 - \rho(s, s)) \cdot r \cdot \ln \Phi(s) \). Because \( \rho(s, s) \) cannot be equal to 1 (since \( \frac{1}{2} < 1 \)), we have \( 1 - \rho(s, s) > 0 \). Therefore, we can divide both sides by \( 1 - \rho(s, s) \) and conclude the proof. \( \square \)

**Lemma A.12.**

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

**Proof.** We apply Lemma A.11 to the initial configuration \( s_0 \). In this configuration, there is a single task \( u_0 \) placed in the deque of processor with index 0. Its potential is \( \Phi(s_0) = \Phi(Q_0 s_0) = e^{\mu T_\infty + \kappa F} \). So, we have: \( E[I(s_0)] \leq r \cdot (\mu T_\infty + \kappa F) \).

Unfolding the definition \( r = \frac{e^{\mu T_\infty + \kappa F}}{(1 - e^{-\mu})} \) gives

\[
E[I(s_0)] \leq \frac{e^{\mu T_\infty + \kappa F}}{(1 - e^{-\mu})} \cdot (T_\infty + \frac{\mu}{\mu} F)
\]

The conclusion follows from the observation that \( \frac{1}{1 - e^{-\mu}} = \frac{\kappa F}{T_\infty} = 1 + \frac{1}{\kappa F} \).

**Lemma A.13.** The expected parallel execution time, \( E[T_P] \), is less than

\[
\left( 1 + \frac{1}{\delta - 1} \right) \cdot \frac{T_1}{P} + \frac{P - 1}{P} \cdot \frac{\mu}{1 - e^{-\mu}} \cdot (T_\infty + 2.68 \cdot c_0 F)
\]

**Proof.** The proof is as follows.

\[
E[T_P] = \frac{1}{P} \left( T_1 + E[K(s_1)] + E[I(s_1)] \right) \leq \frac{1}{P} \left( T_1 + \frac{1}{\delta - 1} T_1 + \left( 1 + \frac{1}{\delta - 1} \right) \cdot \frac{e^{\mu T_\infty + \kappa F}}{(1 - e^{-\mu})} \cdot (T_\infty + \frac{\mu}{\mu} F) \right)
\]

by Lemma A.1 and Lemma A.12.

\[
E[T_P] \leq \left( 1 + \frac{1}{\delta - 1} \right) \cdot \frac{T_1}{P} + \frac{P - 1}{P} \cdot \frac{\mu}{1 - e^{-\mu}} \cdot (T_\infty + \frac{\kappa F}{1 - 2e^{-\kappa}})
\]

rearranging, terms and unfolding the definition of \( \mu \).

At this point, we are still completely free to instantiate \( \kappa \) with any value such that \( \kappa > \ln(2) \approx 0.69 \). We take \( \kappa \approx 1.67835 \), which minimizes the value of \( \frac{\kappa F}{1 - 2e^{-\kappa}} \). We then have \( 1 - 2e^{-\kappa} \approx 0.626637 \) and \( \frac{\kappa F}{1 - 2e^{-\kappa}} \approx 2.67835 < 2.68 \). \( \square \)

**B. Mathematical lemma**

**Lemma B.1.** For any \( q \) and \( n \) such that \( 1 \leq n \leq q \),

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

**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.

![Figure 8](image-url)