Latency-Hiding Work Stealing

Scheduling Interacting Parallel Computations with Work Stealing

Stefan K. Muller
Carnegie Mellon University
Pittsburgh, PA, USA
smuller@cs.cmu.edu

Umut A. Acar
Carnegie Mellon University
Pittsburgh, PA, USA
umut@cs.cmu.edu

With the rise of multicore computers, parallel applications no longer consist solely of computational, batch workloads, but also include applications that may, for example, take input from a user, access secondary storage or the network, or perform remote procedure calls. Such operations can incur substantial latency, requiring the program to wait for a response. In the current state of the art, the theoretical models of parallelism and parallel scheduling algorithms do not account for latency.

In this work, we extend the dag (Directed Acyclic Graph) model for parallelism to account for latency and present a work-stealing algorithm that hides latency to improve performance. This algorithm allows user-level threads to suspend without blocking the underlying worker, usually a system thread. When a user-level thread suspends, the algorithm switches to another thread. Using extensions of existing techniques as well as new technical devices, we bound the running time of our scheduler on a parallel computation. We also briefly present a prototype implementation of the algorithm and some preliminary empirical findings.

1. INTRODUCTION

Recent hardware advances have brought shared-memory parallelism to the mainstream. These advances have motivated significant research and development in programming languages and language extensions for parallelism, including OpenMP, Cilk [15], Fork/Join Java [22], Habanero Java [19], TPL [23], TBB [20], X10 [10], parallel ML [14], and parallel Haskell [21], many of which have had significant impact. In these systems, the programmer expresses parallelism at an abstract level without directly specifying how to create threads and map them onto processors.

In such parallel languages, the runtime system creates the user-level threads, which we simply refer to as threads (they are also known as tasks, strands, sparks, etc.), and relies on a *scheduler* to map the threads onto the processors. The scheduler does not require *a priori* knowledge of the thread structure—it works online as part of the runtime system. The efficiency and performance of a parallel language crucially depends on the scheduler. Many theoretical results have been shown for parallel scheduling, including bounds on runtime and space use [7, 16, 26] as well as locality [1, 6]. Furthermore, the schedulers have been shown to be very effective in practice [15, 2, 30].

The focus of both the theory and experimental analysis of parallel scheduling has been on scheduling computations where threads, once started, are not expected to perform latency-incurring operations, such as waiting for a request from a client or waiting for a response from a remote machine. Popular parallel scheduling algorithms such as work stealing are therefore non-preemptive: they run a thread to completion, never changing to another thread without completing one. This is generally adequate in the traditional applications of parallelism, such as in high-performance computing, where workloads are heavily computational. In such workloads, operations rarely incur latency and thus techniques such as busy-waiting or blocking can be used without significantly harming performance.

Modern workloads of multicore and parallel hardware, however, are not solely computational but also include applications that communicate with external agents such as the user, the file system, a remote client or server, etc. For example, a parallel server may communicate with clients to obtain requests and fulfill them. Since the client may not respond to a request immediately, threads in such an application can incur latency. In such applications, allowing the scheduler to preempt the latency-bound thread and run another thread that can perform useful work could greatly improve performance by *hiding* the latency.

Latency hiding is a classic technique performed by operating system (OS) schedulers. Silbershatz and Galvin's book presents a comprehensive review of these techniques [27]. These techniques, however, are unlikely to carry over to parallel scheduling of user-level threads for several reasons. First, a typical parallel scheduler must carefully control the order in which threads are executed, since it is likely scheduling interdependent threads, while a typical OS scheduler schedules independent jobs which are less sensitive to such ordering. Second, OS schedulers manage a relatively small number of jobs (in the hundreds or perhaps thousands), whereas a typical parallel scheduler will have to manage many more (millions or more) threads. Third, operating system schedulers can use a relatively large scheduling quantum (granularity), whereas parallel schedulers usually schedule fine-grained threads that can perform tiny amounts of work.

In this paper, we propose a model that represents math-

SPAA '16, July 11-13, 2016, Pacific Grove, CA, USA

 ISBN 978-1-4503-4210-0/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2935764.2935793

ematically the behavior of parallel computations that can incur latency and develop a *latency-hiding* scheduling algorithm for such computations. With our scheduling algorithm, a user-level thread that incurs latency can *suspend*, allowing another thread to run in its place.

Our model (Section 2) extends a standard model of parallel computations as Directed Acyclic Graphs, or dags, by allowing edges of the dag to include non-unit weights that represent the *latency* incurred by an instruction. Computational instructions (e.g., integer or floating point arithmetic) incur no latency. This is represented with unit-weight or *light* edges. Operations that communicate, such as I/O operations, remote procedure calls, or messaging primitives can incur latency, which is represented with non-unit weight *(heavy)* edges. We define the *span* or the *depth* of such a weighted computation dag to include the latency, by calculating the longest weighted path in the dag. As in the traditional model, we use the term *work* to refer to the total computational work of the dag, excluding edge weights.

Our scheduling algorithm (Section 3), like traditional work-stealing algorithms, uses deques (double-ended queues) to store the ready threads to be executed. Instead of one deque per worker, however, our algorithm can use many deques per worker to keep track of suspended threads which were switched out in favor of other work. Our algorithm is online and does not require knowledge of the dag or the edge weights (latencies) but requires knowing which edges are heavy and which edges are light.

To analyze the efficiency of the algorithm, in addition to the traditional notions of work and span, we rely on a notion of *suspension width*, which measures the number of heavy edges that cross a source-sink partition of the computation dag. Suspension width is related to the notion of *s-t* cuts in flow algorithms.

For a dag with W work, S span and $U \geq 1$ suspension width, we prove that the latency-hiding work-stealing algorithm yields a P-processor runtime bound of

$$O\left(\frac{W}{P} + SU(1 + \lg U)\right),$$

amortized and in expectation (Section 3). When $U = 0$, all edges are light (with weight 1). In this case, our algorithm behaves identically to standard work stealing, achieving the bound $O\left(\frac{W}{P} + S\right)$. We note that, in our bounds, work W does not include latency, indicating that our scheduler is able to avoid incurring any latency that does not fall on the critical path (S) of the computation.

We have completed a prototype implementation of the algorithm and obtained some preliminary empirical results (Section 6). These results show that, for computations with significant latency, latency-hiding work stealing can improve speedups compared to the standard work-stealing algorithm, which does not hide latency. Our evaluation also suggests that the factors involving U in the runtime bound might not hinder practical speedups and that our algorithm can handle computations with large numbers of suspended threads.

2. WEIGHTED DAG MODEL

We extend the traditional Directed Acyclic Graph (dag) model of parallel computations to account for computations that may suspend. Throughout the paper, we mostly follow the terminology of Arora, Blumofe, and Plaxton [5].

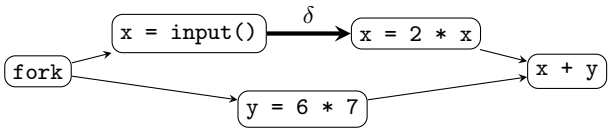


Figure 1: An example weighted dag

We represent a multithreaded/parallel computation with a weighted dag. Vertices of the dag represent single instructions of a thread, each of which is assumed to perform one *step* of work. Operations that require more than one step of work can be represented by a chain of instructions. An edge from u to v indicates a dependence between instructions u and v, requiring u to be executed before v. In a multithreaded or parallel program, an edge from u to v can originate from three conditions: (1) u comes before v within a single program thread; (2) u spawns a new thread whose first instruction is v, (3) the threads containing u and v synchronize, e.g., via a join, requiring u to execute before v. We order the edges in the dag such that when a vertex u spawns a new thread whose first instruction is v, v is the *right child* and the other child, which is the continuation of u in the same thread, is the *left child*.

We label each edge with its latency, a positive integer weight, and write (u, v, δ) for an edge from u to v with weight δ. If $\delta = 1$, then we refer to the edge as a *light* edge. In this case, there is no latency between u and v, and thus v may be scheduled and use the results of u immediately after u executes. If $\delta > 1$, then we refer to the edge as a *heavy* edge. In this case, there is a δ-step latency between u and v, and v may use the results of u only after δ steps. Thus v cannot be scheduled to execute for at least δ steps.

As an example, consider a toy parallel program that creates two threads. The first thread multiplies 6×7 and returns the result. The second thread requests an integer x from the user, doubles x, and returns it. The program then waits for both threads to complete and adds the returned values. Since the user may not respond immediately, the reading of the variable x incurs a latency. Figure 1 shows the dag for an execution of this program where reading the input x takes δ steps. Throughout the paper, the weight of 1 is omitted from light edges, and these are drawn as thin lines while heavy edges are drawn as thick lines. This example demonstrates a common use case of the model in which instructions, like `input()` of the example, perform a unit of work and start an operation (a console input, file read, network operation, etc.) that takes $\delta - 1$ time steps to complete. Such an operation is easily modeled, as in the figure, by a vertex whose outgoing edge(s) has/have weight δ. In the rest of this paper, our examples will be drawn from this simple form of latency-incurring operation, though the model might allow for more general forms of latency.

We make several assumptions regarding dags:

- A dag has exactly one *root*, which has in-degree zero, and exactly one *final vertex*, which has out-degree zero.
- The out-degree (number of outgoing edges) of a vertex is at most two, i.e., an instruction may spawn or synchronize with at most one other thread.
- If vertex v has a heavy incoming edge, i.e., there exists an edge (u, v, δ) with $\delta > 1$, then v has in-degree 1.
- The dag is deterministic, that is, its structure is in-

dependent of the decisions made by the scheduler. In the case of weighted dags, this assumption requires that the latency of an instruction does not depend on when the instruction is executed.

The first two assumptions are common in the literature. The third assumption seems to cause no loss of generality, because a vertex with multiple heavy in-edges can be replaced with multiple vertices, with edges distributed to meet this restriction. The final assumption could be relaxed by considering the set of possible dags and stating time bounds and other properties in terms of the worst case over all possible dags, but we do not consider this here.

We say that a vertex is *enabled* when all of its parents have executed and it is *ready* when it is enabled and all latency requirements expire. Consider a vertex v and let u be the last parent of v to execute. If the edge (u, v, δ) is light, then v is enabled and ready immediately after u is executed. If the edge (u, v, δ) is heavy, then v is enabled immediately after u is executed, but ready only δ steps after u is executed. (Note that by the third assumption above, v has only one heavy in-edge.)

We define the *span* of a weighted dag to be the longest *weighted* path in the dag. If a dag has no heavy edges, then this reduces to the traditional, unweighted notion of span (which simply counts the edges along a path). We define the *work* of a weighted dag as the total number of vertices in the dag. The definition of the work is unchanged from the traditional model—weights do not count toward the work.

To analyze our scheduling algorithm on weighted dags, we introduce a new measure, U, the *suspension width*, of a dag.

DEFINITION 1 (SUSPENSION WIDTH). *Consider a parallel computation represented by a dag $G = (V, E)$ with vertices V and edges $E \subset V \times V \times \mathbb{N}$. Let s be the root vertex and t be the final vertex of G. Let $\mathcal{P}$ be the set of all partitions (S, T) of G (where $S \uplus T = V$ and S and T each induce a connected subdag of G) such that $s \in S$ and $t \in T$. We define the suspension width of G as the maximum number, over all partitions, of heavy edges that cross the partition. More precisely,*

$$U = \max_{(S,T)\in\mathcal{P}} \left|\{(u, v, \delta) : u \in S, v \in T, (u, v, \delta) \in E \mid \delta > 1\}\right|.$$

The suspension width of a dag is the maximum number of heavy edges that cross a source-sink partition. This notion is similar to an *s-t cut* in flow networks, which inspires our notation. The main differences are that s-t cuts generally do not require that the cut induce connected components and they include all source-to-sink edges, rather than certain (heavy) edges as we do here.

The suspension width is relevant to scheduling of weighted dags because it is the maximum number of vertices that can be suspended at any point during the run of the computation. To see this, at the end of step i of the execution, let S_i consist of the instructions that have been executed and T_i consist of the instructions that have not been executed. By construction, $S_i \uplus T_i = V$ and S_i contains the root vertex and T_i contains the final vertex. The suspended vertices at the end of step i are exactly those in the set

$$\{v \mid (u, v, \delta) \in E, u \in S_i, v \in T_i, \delta > 1\}$$

Since each suspended vertex has exactly one in-edge, the number of suspended vertices is equal to the number of heavy edges crossing the partition (S_i, T_i) which, by definition is no greater than U.

Offline scheduling problem. Given a dag and a number P of processors (workers), a *schedule* is an assignment of vertices of the dag to worker-step pairs (p, i), such that each worker executes at most one vertex per step and each vertex is ready when it is executed.

The *offline scheduling problem* requires finding a short schedule for a computation represented by a given weighted or unweighted dag on P workers. For unweighted dags, finding an optimal solution is NP-hard [31], but Brent [8] showed that a "level-by-level" schedule on P workers has length within a factor of two of optimal. Future work [12] proved a similar bound for *greedy schedules*, which are defined by keeping all workers busy on steps when there are at least P ready vertices.

We now generalize these results to show that a greedy schedule gives us a similar bound for weighted dags.

THEOREM 1. *Consider a parallel computation with work W and span S. Any greedy schedule of this computation for P workers has length at most $\frac{W}{P} + S$.*

Our proof, which appears in the companion technical report [25], uses much the same technique as the ABP proof for unweighted dags [5]. The intuition is that, if all workers are busy at a step, then they are "making progress" on the work of the computation, but if any workers are idle on a step, then they are "making progress" on the span by executing all vertices at a level in the dag and/or "counting down" the latencies of heavy edges. Our bound is slightly different from that of ABP, $\frac{W}{P} + S\frac{P-1}{P}$, since in weighted dags, at any step, it is possible for all workers to be idle and waiting for suspended vertices, whereas with unweighted dags, at least one worker must be busy at each step.

Online scheduling problem. The *online scheduling problem* for unweighted or weighted dags requires scheduling a dag as its structure is revealed during the execution. In particular, a scheduler operates in *rounds*. On each round, a scheduler may execute an instruction v and learn what new vertices become enabled and possibly ready as a result. Executing v may enable the children of v but not all children of v become ready immediately because they may suspend due to a heavy incoming edge. Furthermore, in the case of weighted dags, the scheduler does not know the edge weights of heavy edges and thus does not know how long it will be before suspended vertices become ready, though it does know whether outgoing edges are heavy or light.

In the rest of the paper, we present an algorithm that solves the online parallel scheduling problem for weighted dags, and an analysis of this algorithm. Our analysis uses the completed dag, including edge weights, in an *a posteriori* fashion, but the scheduler does not use this information since, in general, such information is usually not available in a realistic computation.

3. ALGORITHM

Our algorithm is a variant of work stealing and, as such, uses deques to store the ready vertices to be executed. Instead of one deque per worker (thread), however, our algorithm can use many deques per worker, only one of which is *active* at a time. To obtain work, the worker pops a vertex

from the bottom of its active deque. If the worker successfully removes a vertex from the bottom of its deque, it executes that vertex in one step. The execution of the vertex may enable zero, one or two new vertices. If a newly enabled vertex is ready, then it is executed or pushed onto the bottom of the deque in the usual way. If however, the vertex suspends (corresponding to a heavy edge in the dag), the algorithm pairs the suspended vertex with the active deque—the suspended vertex *belongs* to the active deque—and goes back to its active deque to obtain work. If a worker finds its deque to be empty, it first checks if it owns another deque that can become active. If so, it switches to that deque. Otherwise, it becomes a *thief*. To obtain work, a thief randomly chooses any deque and attempts to remove, or *steal*, the vertex at the top of the chosen deque, the *victim*. When a thief successfully steals a vertex, it starts a new deque and makes the new deque its active deque.

The algorithm outlined above is almost complete, except for one crucial point: it does not deal with suspended vertices that resume (become ready). In every round, our scheduler checks for suspended vertices that have resumed since the last round. Since there can be arbitrarily many resumed vertices at a check point, a worker cannot handle them by itself without harming performance. To solve this problem, new work is injected into the computation that will execute the resumed vertices in parallel. More specifically, the scheduler partitions the resumed vertices according to the deques to which they belong and, for each deque, spawns a function that will, in a parallel `for` loop, execute the resumed vertices belonging to that deque.

Figure 3 shows the pseudocode for the algorithm, which we now describe in more detail.

Deques and Sets. Each worker *owns* a collection of deques. Each deque is either *ready*, if it has work, or *suspended* if it is empty and there is a suspended vertex which will eventually be returned to it. At any point, one of the deques of each worker is designated as the *active deque*. The worker also has a set `readyDeques` of its (non-active) ready deques, and a set `resumedDeques` of deques for which a suspended vertex has resumed since the last round. A summary of state changes of deques is shown in Figure 2. Table 1 lists the operations and fields we require for operating on deques, sets of deques, and sets of vertices. We assume that all listed operations take (possibly amortized) constant time. In addition, we assume iteration (using `for` loops) over sets is possible and, in particular, that *parallel* operations over vertex sets are possible (using `pfor` loops) in linear work and logarithmic span in the size of the set.

Several implementations of work-stealing deques exist (e.g. [5, 11, 2]) which provide the required guarantees. Our setting satisfies the requirements of these implementations since each deque is always owned by the same single worker. At the end of this section, we describe an implementation of the other required data structures.

Suspended Vertices. When a vertex v suspends and the active deque is q, a callback `callback(v, q)` is installed to be run when v resumes [1]. The callback adds v to the set `q.resumedVertices`, which tracks the (newly) resumed vertices belonging to q, decrements `q.suspendCtr` to indicate that q has one fewer suspended vertex and, if this has not already been done, adds q to the set of resumed deques.

[1]In practice, this may be implemented by, e.g. signal handlers or polling in a separate (system) thread.

Member	Description
	Deques `q`
`q.popTop()`	Pop the top vertex of deque `q`
`q.popBottom()`	Pop the bottom vertex of `q`
`q.pushBottom(v)`	Push `v` to the bottom of `q`
`q.free()`	Free the deque `q`
`q.suspendCtr`	A counter of `q`'s suspended vertices
`q.resumedVertices`	A set of `q`'s resumed vertices
`newDeque()`	Create a new, empty deque
`randomDeque()`	Return *any* allocated deque
	Deque sets `qs`
`qs.add(q)`	Add `q` to set `qs`
`qs.removeAny()`	Remove any deque from `qs`
`qs.remove(q)`	Remove `q` from `qs`
	Vertex sets `vs`
`vs.add(v)`	Add `v` to set `vs`
`vs.clear()`	Clear the set `vs`
`vs.size`	The size of set `vs`

Table 1: Deque and set interfaces

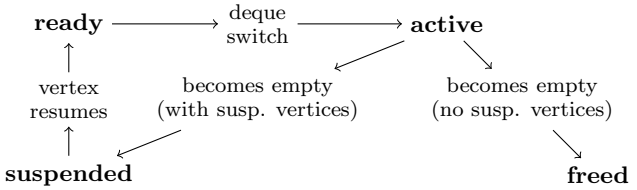


Figure 2: Transitions of deque states

The function `addResumedVertices()` iterates over the set of resumed deques. For each such deque q, a closure is allocated which executes all of the resumed vertices for q in a parallel `for` loop. This closure is encapsulated in a new vertex, which is pushed onto the bottom of q. Finally, q is added to the set of ready deques for this worker.

Scheduling loop. Execution starts by setting one worker's `assignedVertex` to the root of the computation and setting the `activeDeque` of every worker to an empty deque (Line 25 in Figure 3). Execution then jumps to execute the main scheduler loop, shown between lines 31 and 56 of Figure 3. We refer to one iteration of the loop as a *round* of the scheduler. If a worker has an assigned vertex on a round, it takes the following actions:

1. Execute the vertex to get zero, one or two (possibly suspended) children.
2. Handle the right child (if present). If it is suspended, the function `handleChild` increments the suspension counter of the active deque and installs the callback. Otherwise, the child is pushed onto the bottom of the active deque.
3. Call `addResumedVertices()`.
4. Handle the left child (if present) as described above.

Actions 2, 3 and 4 are done in this order so that the left child will have higher priority than the right child and the pfor tree. This ensures that our scheduler is not preemptive: the current task continues running until it finishes.

```
1  function callback(v, q)
2    q.resumedVertices.add(v)
3    q.suspendCtr = q.suspendCtr - 1
4    if (q.resumedVertices.size == 1)
5      resumedDeques.add(q)
6
7  function addResumedVertices()
8    for (q in resumedDeques)
9      v = vertex({ pfor (u in q.resumedVertices)
10                    u.execute() })
11     q.pushBottom(v)
12     readyDeques.add(q)
13     resumedDeques.remove(q)
14     q.resumedVertices.clear()
15
16 function handleChild(v)
17   q = activeDeque
18   if (v.isSuspended)
19     v.installCallback(callback(v, q))
20     q.suspendCtr = q.suspendCtr + 1
21   else
22     q.pushBottom(v)
23
24 // Assign root to worker zero.
25 assignedVertex = NULL
26 activeDeque = newDeque()  // Initial deque
27 if (self == WorkerZero)
28   assignedVertex = rootVertex
29
30 // Run scheduling loop.
31 while (!computationDone)
32   // Execute assigned vertex.
33   if (assignedVertex <> NULL)
34     (left, right) = assignedVertex.execute()
35     if (right <> NULL)
36       handleChild(right)
37     addResumedVertices()
38     if (left <> NULL)
39       handleChild(left)
40     assignedVertex = activeDeque.popBottom()
41   else
42     if (activeDeque.suspendCtr = 0)
43       activeDeque.free()
44       activeDeque = NULL
45     // First, try to resume a ready deque.
46     new = readyDeques.removeAny()
47     if (new <> NULL)
48       activeDeque = new
49     else  // Make steal attempt.
50       victim = randomDeque()
51       assignedVertex = victim.popTop()
52       if (assignedVertex <> NULL)
53         activeDeque = newDeque()
54     addReadyVertices()
55     if (assignedVertex == NULL)
56       assignedVertex = activeDeque.popBottom()
```

Figure 3: Pseudocode for the scheduling algorithm

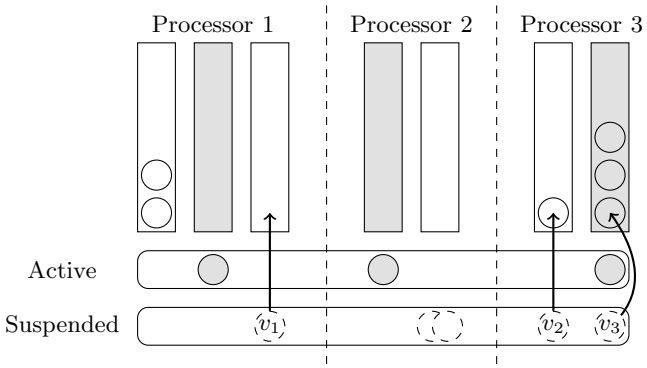


Figure 4: Illustration of work stealing

If a worker doesn't have an assigned vertex, it attempts to switch its active deque to another ready deque. If successful, the new deque becomes the active deque and its bottom vertex becomes the assigned vertex. If all deques are suspended, the worker becomes a thief and steals the top vertex from a randomly chosen deque (note that, unlike in most conventional work stealing schedulers, a steal targets deques and not workers; the victim deque is chosen uniformly at random from all deques). If the steal is successful, `newDeque()` is called to generate a new active deque; the children (if any) of the stolen vertex will be pushed to this new deque. Finally, whether a deque switch or steal attempt occurred, `addResumedVertices()` is called.

Figure 4 illustrates an example state of the scheduler. Active deques and vertices are shown with gray backgrounds. Suspended vertices are shown as dashed circles below the deque with which they are associated. Suppose that, in the next round, suspended vertices v_1, v_2 and v_3 will resume and be returned to their respective deques. Assuming none of the active vertices enable new work, the active deques of processors 1 and 2 will become empty. Processor 1 will switch its active deque. Processor 2 will steal work from one of the other processors and create a new deque. Processor 3 will continue working from its active deque.

Deque and Set implementation. To implement the operations of Table 1 efficiently, the algorithm maintains a global (across all workers) array of deques, called `gDeques`. This may be a growable array (resizes will require a full synchronization and should be rare, in order to amortize the cost of the synchronization and the copy) or, if acceptable for the application, a fixed-size array. There is also a global counter, `gTotalDeques`, which indicates the array index of the next deque to be allocated.

Each worker p maintains a set `emptyDeques` of indices of deques previously owned by p which have been freed by `free()`. Whenever possible, `newDeque()` will reuse a deque from `emptyDeques` rather than allocating a new one. If no empty deques are available, it increments `gTotalDeques` using an atomic fetch-and-add (or other synchronization primitives). The next deque in the array is allocated and becomes the active deque. This implementation of `newDeque()` is shown in Figure 5. The implementation of `free()` does not actually deallocate the deque, but simply adds it to the `emptyDeques` set. The array implementation makes the implementation of `randomDeque()` quite straightforward: it simply chooses a random index between 0 (inclusive) and

```
1 function newDeque()
2   dequeToReturn = emptyDeques.removeAny()
3   if (dequeToReturn == NULL)
4     i = fetch_and_add(gTotalDeques, 1)
5     gDeques[i] = new Deque
6     dequeToReturn = gDeques[i]
7   return dequeToReturn
```

Figure 5: The `newDeque` function

`gTotalDeques` (exclusive). The chosen deque may have been "freed", in which case the steal will fail. Since our worst-case asymptotic analysis assumes that the maximum number of deques are allocated, the (somewhat loose) upper bound we prove is not impacted by stealing from freed deques. Our implementation (Section 6) uses a more optimized policy.

A set of deques may be implemented as a doubly-linked list, allowing for constant-time removal. Since removal is not necessary for sets of vertices, but parallel operations (for which lists are unsuitable) are, vertex sets may be implemented as growable arrays, with amortized constant-time addition. In a language with garbage collection, `clear` can be implemented in constant time by simply destroying the pointer to the array. Without garbage collection, the cost of freeing each element can be charged to the corresponding calls to `add`, and so the amortized cost of the three operations will still be constant.

4. ANALYSIS

To bound the running time of the latency-hiding work-stealing scheduler, we first bound the number of rounds in terms of the work of the computation and the number of steal attempts. Based on this result, in Sections 4.1 and 4.2, we establish an upper bound of $O(\frac{W}{P} + SU(1 + \lg U))$ on the expected number of rounds by bounding the number of steal attempts. Our analysis is based on an analysis of work stealing for dedicated environments [3], which simplifies the analysis of Arora et al. [5] for multiprogrammed environments. Finally, in Section 4.3, we use an amortization argument to show that, on average, each round takes constant time. This gives the desired bound on the running time.

We first establish an upper bound on the number of rounds in an execution in terms of the work of the computation and the number of steal attempts (Lemma 1). Our proof accounts for deque switches performed during execution and the additional work injected into the computation to process the resumed vertices. Such additional work comes in the form of the parallel `for` (`pfor`) loops that execute resumed vertices. For n resumed vertices, such a `pfor` loop unfolds into a tree, which we call a *pfor tree* (its vertices are *pfor vertices*) with $\lg n$ span and n leaves, each of which executes one of the resumed vertices (which itself expands into a dag).

LEMMA 1. *Consider a parallel computation with work W and span S executed on P workers by our algorithm. The number of rounds taken to complete the computation is at most $4\frac{W}{P} + \frac{R}{P}$, where R is the number of steal attempts.*

PROOF. At each round, each worker places a token into one of three buckets: the *work bucket* if it executes an instruction (including a pfor vertex), the *switch bucket* if it switches deques or the *steal bucket* if it attempts to steal. At the end of the computation, the number of tokens in the work bucket is equal to $W + W_{pfor}$, where W_{pfor} is the number of pfor vertices. Since a (binary) tree has at most as many internal vertices as leaves and the leaves of a pfor tree are included in W, we have $W + W_{pfor} \leq 2W$. At a round i, if worker p places a token in the switch bucket, it now has an active deque which is ready, and an assigned vertex. It will therefore place a token in the work bucket on round $i + 1$, so at the end of the computation, there are at most as many tokens in the switch bucket as there are in the work bucket, and the number of tokens in these two buckets together is at most $4W$. By definition, the number of tokens in the steal bucket is exactly R. Since P tokens are placed per round, the number of rounds is as desired. □

Given this result, to bound the number of rounds, it suffices to bound the number of steal attempts by $O(PSU(1+\lg U))$.

4.1 Enabling trees and the potential function

An *enabling tree* is a tree derived from a computation dag, which represents the relationship between vertices in a particular execution. As opposed to the computation dag, an enabling tree is a purely runtime notion and there can be many possible enabling trees for a given dag. Enabling trees are not constructed by the algorithm; we simply use them as technical devices for the analysis. In the unweighted case (e.g. [5]), an edge (u, v) of the dag is an *enabling edge* and is included in the enabling tree if executing u enables v. In this way, tracing up from the leaves of the enabling tree gives a record of when vertices were added to deques.

We extend the concept of an enabling tree to account for heavy edges in a way that will be made precise later in this section. In keeping with the intuitive notion of an enabling tree described above, suspended vertices should be added to the enabling tree at the point at which they become ready. This means that heavy edges will not appear in the enabling tree. Consider the computation dag shown in Figure 6(a). When heavy edges and non-enabling light edges are removed from the dag, the suspended vertices that are the targets of these edges (vertices 4 and 9) will have no parent in the dag. This induces a spanning tree consisting of the components of the dag that are connected with only light edges, shown in Figure 6(b). Suppose that, after vertex 3 executes, vertex 4 resumes. We then consider vertex 3 to have enabled vertices 4, 5 and 6 and place vertex 4 in the enabling tree accordingly[2]. If after vertex 5 executes, vertex 9 becomes ready, this relationship is represented in the enabling tree as well. The final enabling tree is shown in Figure 6(c).

The above discussion glosses over two points. First, the construction violates our assumption that vertices have at most two outgoing edges. This is easily solved by adding an additional vertex and distributing the outgoing edges accordingly, a process sometimes called "ternarization". The

[2]The term "enabling tree", which comes from prior work, unfortunately clashes with our terminology, in which vertex 4 was already enabled and is now ready. One way to make sense of this is to think of the enabling tree as building up an unweighted program which is consistent with the execution of the weighted program. For example, the behavior of our scheduler on the example dag "looks like" the behavior of standard work stealing on a dag where vertex 3 is a 3-way fork enabling vertices 4, 5 and 6 (where the out-degree restriction is temporarily relaxed to allow this).

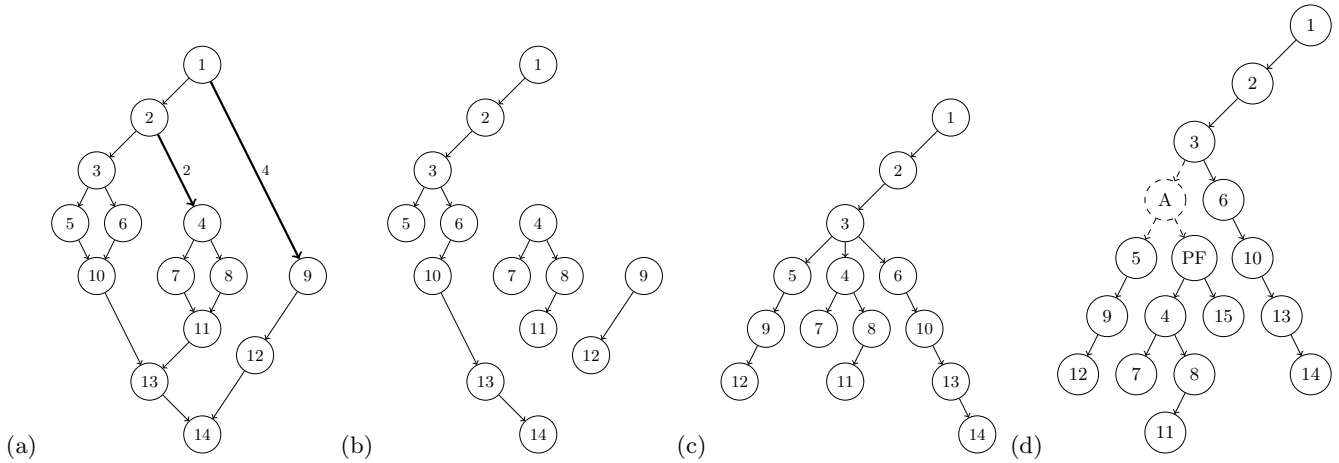


Figure 6: Construction of a possible enabling tree: (a) the input dag (b) spanning forest of the light-edge components (c) the enabling tree (d) the enabling tree with restricted out-degree and pfor trees (assuming a new vertex 15, which resumes at the same time as vertex 4)

exact construction will be detailed below. Second, if multiple suspended vertices resume at the same time, they are not added to the enabling tree directly. Rather, the root of the pfor tree is added at the correct location and it will, in turn, spawn all of the suspended vertices (and their dags). In the context of the example, suppose that there is another vertex, 15, which resumes in the same round as vertex 4. Figure 6(d) shows an enabling tree including an auxiliary vertex A to reduce the out-degree of vertex 3 and a pfor vertex PF, which spawns vertices 4 and 15.

At a more precise level, construction of the enabling tree proceeds as follows. All vertices and all enabling edges (which are all light) are included in the enabling tree. To determine where to add pfor trees in the enabling tree, we proceed by cases on whether the pfor is added to the active deque or a non-active deque. If a pfor vertex v' is added to the active deque, then the scheduler just executed a vertex u from the active deque. Proceed by cases on what kinds of children, if any, u enables. If u enables no children, we simply add an *auxiliary edge* $(u, v', 1)$ to join the pfor tree to u. If u had a left child v, we add an *auxiliary vertex* u' and *auxiliary edges* $(u, u', 1)$, $(u', v', 1)$ and $(u', v, 1)$, where v is considered the left child of u'. If u had a right child r, the edge $(u, r, 1)$ is unchanged. In Figure 6(d), vertex 3 is u, vertex 5 is v, PF is v' and vertex 6 is r.

In the second case, suppose a pfor tree rooted at v' is added to a non-active deque q, and let i be the round in which it is added. If q is non-empty, let u be the bottom vertex in q and j be the round in which it was added to the deque. If q is empty, let u be the last instruction to execute from q and j be the round in which it executed. To maintain the intuitive property that the enabling tree can be used as a record of additions to the deques, we wish to add v' as a descendant of u in the enabling tree. However, it is also important for the analysis to make sure that v' is at a depth in the enabling tree corresponding to when it is added to the deque. To account for the time between when u was executed or added to the deque and when v' is added, we add a chain of $i - j - 1$ auxiliary vertices $u_1, \ldots, u_{i-j-1}$ and auxiliary edges

$$(u, u_1, 1), (u_1, u_2, 1), \ldots (u_{i-j-1}, v, 1)$$

The *depth* $d(v)$ of v in the enabling tree is the length of the path from the root to v in the enabling tree. The depth $d_G(v)$ of v in the original dag G is the length of the longest weighted path from the root to v in G. Let S^* (the "enabling span") be the longest path in the enabling tree at the end of the computation. Observe that, by the time a suspended vertex whose incoming edge has weight δ becomes ready, the worker will not have proceeded deeper than δ levels in the dag (it may even have proceeded fewer, if it backtracked), and so the pfor tree for a suspended vertex v will be reinserted into the enabling tree at a depth no more than a factor of $\lg U$ greater than $d_G(v)$, even considering auxiliary and and pfor vertices. Thus, the enabling span is $O(S(1 + \lg U))$. This key fact will be a direct result of Lemma 2, which shows a number of invariants about the enabling tree and the execution of the scheduler.

LEMMA 2. *Consider an execution of a multithreaded computation represented by a dag G, which results in an enabling tree with enabling span S^*. At the end of a round i, the following conditions hold:*

1. *For all vertices $u \in G$ already executed or currently ready, $d(u) \leq (2 + \lg U) d_G(u)$.*
2. *For all ready or executed pfor vertices u which have depth d in a pfor tree, $d(u) \leq (2 + \lg U) d_G(v) - \lg U + d$ for all v in the pfor tree of u.*
3. *Let u be a suspended vertex whose parent u' came from deque q, with the in-edge (u', u, δ) for some δ, and which was enabled in round $i - k$, for $k \geq 0$. If v was the last instruction executed from deque q and was executed in round $i - j$, for $j \geq 0$, then $d(v) \leq (2 + \lg U) d_G(u') + 2k - j$.*
4. *Let u be a suspended vertex whose parent u' came from deque q, with the in-edge (u', u, δ) for some δ, and which was enabled in round $i - k$, for $k \geq 0$. If v is the assigned vertex of p, then $d(v) \leq (2 + \lg U) d_G(u') + 2(k + 1)$.*
5. *Let $v_1, \ldots, v_k$ be the vertices in any deque, from bottom to top and v_0 be the assigned vertex if this deque is the active deque of a worker. Let $u_0, \ldots, u_k$ be their*

parents in the enabling tree. For all $1 \leq i \leq k$, vertex u_i is an ancestor of u_{i-1} in the enabling tree, and this ancestor relationship is proper for $i > 1$.

The proof appears in the companion technical report [25]. It is now straightforward to show that the enabling tree is no more than a factor of $\lg U$ deeper than the original dag.

COROLLARY 1. *Consider a computation represented by a dag G with span S. When run to completion using the scheduler, if the enabling tree has enabling span S^*, then $S^* \in O(S(1 + \lg U))$.*

PROOF. Let v be the deepest vertex in the enabling tree (that is, $d(v) \geq d(u)$ for all u in the enabling tree). By definition, $S^* = d(v)$. The deepest vertex in the enabling tree will be in G (it will not be a auxiliary vertex or a pfor vertex, since these always have descendants which are in G). By condition 1 of Lemma 2, we have

$$S^* = d(v) \leq (2 + \lg U) d_G(v) \leq (2 + \lg U) S \leq 2S(1 + \lg U)$$

□

We now define the potential function which we will use to show the bound on the number of steal attempts. Let $w(v) = S^* - d(v)$ be the *weight* of a vertex v and define the potential of v at round i as follows:

$$\phi_i(v) = \begin{cases} 3^{2w(v)-1} & v \text{ is assigned at the start of round } i \\ 3^{2w(v)} & \text{otherwise} \end{cases}$$

Each non-active deque with suspended vertices has an extra potential

$$\phi_i^E(q) = \begin{cases} 2 \cdot 3^{2w(v)-2j} & q \text{ is empty} \\ & \text{and last executed } v \text{ in round } i - j \\ 2 \cdot 3^{2w(v)-2j} & v \text{ is the bottom vertex of } q \\ & \text{and was added in round } i - j \\ 0 & q \text{ is the active deque} \end{cases}$$

The extra potential decreases over time, simulating the execution of the chains of auxiliary vertices added to the enabling tree when suspended deques resume. The total potential of a deque q with vertices $v_1, \ldots, v_k$ is:

$$\Phi_i(q) = \phi_i^E(q) + \Sigma_{j=1}^{k} \phi_i(v_j)$$

4.2 Bounding steal attempts

At the beginning of a round i, let

- A_i be the set of assigned vertices.
- D_i be the set of ready deques.
- S_i be the set of suspended deques.
- $\Phi_i(A_i) = \Sigma_{v \in A_i} \phi_i(v)$
- $\Phi_i(D_i) = \Sigma_{q \in D_i} \Phi_i(q)$
- $\Phi_i(S_i) = \Sigma_{q \in S_i} \Phi_i(q)$
- $\Phi_i = \Phi_i(A_i) + \Phi_i(D_i) + \Phi_i(S_i)$

We note that Φ_i is the total potential for the computation at the start of round i. At the start of the computation, the computation consists only of the root, which has weight S^*, so the potential is 3^{2S^*-1}. At the end of the computation, no vertices are remaining, so the final potential is 0.

The next two lemmas show important properties of the potential function: first, that a constant fraction of the total potential of a deque sits at the top of the deque, accessible to a steal. Next, that executing a vertex decreases the total potential by a constant fraction of that vertex's potential.

LEMMA 3 (TOP-HEAVY DEQUES). *Consider any round i and any deque $q \in D_i$. The topmost vertex in q's deque contributes at least 2/3 of the total potential of q.*

PROOF. It can be shown from Condition 5 of Lemma 2 that if $v_1, \ldots, v_k$ are the vertices of q, then $w(u_1) < \cdots < w(u_k)$. The lemma follows from this fact and the definition of the potential function. □

LEMMA 4. *Consider any round i and let $v \in A_i$.*

$$\Phi_i - \Phi_{i+1} \geq \frac{5}{9} \phi_i(v)$$

PROOF. By cases on what scheduler actions follow the execution of v. □

Furthermore, potential does not increase during computation. The proof of this lemma appears in the companion technical report [25].

LEMMA 5. *For any round i, $\Phi_{i+1} \leq \Phi_i$.*

Lemma 6 is used to show how steal attempts decrease the potential. See [5] for the proof.

LEMMA 6 (BALLS AND WEIGHTED BINS). *Suppose that P balls are thrown independently and uniformly at random into P bins, where for $i = 1, \ldots, P$, bin i has a weight W_i. The total weight is $W = \Sigma_{i=1}^{P} W_i$. For each bin i, define the random variable X_i as*

$$X_i = \begin{cases} W_i & \text{if some ball lands in bin } i \\ 0 & \text{otherwise} \end{cases}$$

If $X = \Sigma_{i=1}^{P} X_i$, then for any β in the range $0 < \beta < 1$, we have $Pr[X \geq \beta W] > 1 - \frac{1}{(1-\beta)e}$.

To show the bound on steal attempts, we divide the computation into *phases*, with at least $P(U + 1)$ steal attempts in each phase. We will then bound the number of phases. The reason U is relevant in this bound is that, at any time, each worker may have up to $U+1$ deques, each of which may be targeted by a steal. In the parlance of the Balls and Bins Lemma, there are thus $O(PU)$ bins (deques) into which we throw balls (steal attempts). We first show this bound on the number of deques. In the proof, we consider the implementation of `newDeque()` given at the end of Section 3, in which deques are never deallocated but are recycled.

LEMMA 7. *At all times during execution of the scheduler, no worker owns more than $U + 1$ allocated deques.*

PROOF. By induction on the sequence of actions taken by the scheduler. The only action that adds a deque is a call to `newDeque()`, which only occurs on a steal. A steal only occurs if there are no ready deques, and `newDeque()` only allocates a new deque if there are no empty deques (including the active deque, which will have just been added to the list if it is not suspended). Therefore, if a new deque is allocated, all deques owned by the worker must be suspended. There can be at most U such deques. □

We now show that any phase, with constant probability, decreases the total potential by a constant fraction of the potential in the deques.

LEMMA 8. *Consider any round i and any later round j such that at least $P(U+1)$ steal attempts occur from rounds i (inclusive) to j (exclusive). Then*

$$Pr[\Phi_i - \Phi_j \geq \frac{2}{9}\Phi_i(D_i)] > \frac{1}{4}$$

PROOF. Suppose a steal in round $k > i$ targets a non-empty deque $q \in D_i$, of which v is the top vertex. After round k, vertex v will be assigned (by the thief or by the victim or another thief if there is contention). By Lemma 3, v contributes at least $\frac{2}{3}$ of the potential of q, and assigning it decreases its potential, so

$$\Phi_i(q) - \Phi_{i+1}(q) \geq \frac{2}{3}\phi_i(v) \geq \frac{2}{3}\frac{2}{3}\Phi_i(q) = \frac{4}{9}\Phi_i(q)$$

Next, we use the Balls and Bins Lemma to establish the total decrease in potential. In the setting of the lemma, let $W_q = \frac{4}{9}\Phi_i(q)$ if $q \in D_i$ and $W_q = 0$ otherwise. Therefore, in the setting of the lemma, X is the total decrease in potential after $P(U+1)$ steal attempts and $W = \frac{4}{9}\Phi_i(D_i)$. By the lemma, using $\beta = \frac{1}{2}$, we have

$$Pr[\Phi_j - \Phi_i \geq \frac{2}{9}\Phi_i(D_i)] > 1 - \frac{1}{(1-\frac{1}{2})e} > \frac{1}{4}$$

□

Putting the above results together shows the desired bound on the number of steal attempts.

THEOREM 2. *Consider a computation with work W, span S and suspension width $U > 1$. The number of rounds taken by the latency-hiding work-stealing scheduler on P workers is $O(\frac{W}{P} + SU(1 + \lg U))$ in expectation.*

PROOF. Consider a phase which consists of the rounds $[i, j)$. Let $u \in A_i$. After u is executed on round i, the potential drops by at least $\frac{5}{9}\phi_i(u)$ (by Lemma 4), so we have $\Phi_i - \Phi_j \geq \frac{5}{9}\Phi_i(A_i)$.

Let $q \in S_i$. By the definition of the potential, for some k,

$$\begin{aligned}\Phi_i(q) - \Phi_j(q) &\geq 2 \cdot 3^{2w(v)-2(i-k)} - 2 \cdot 3^{2w(v)-2(j-k)} \\ &= (1 - 3^{2(j-i)})\Phi_i(q) \\ &> \frac{8}{9}\Phi_i(q)\end{aligned}$$

Summing over all deques in S_i, $\Phi_i - \Phi_j \geq \frac{8}{9}\Phi_i(S_i)$.

From Lemma 8, we know that $Pr[\Phi_i - \Phi_j \geq \frac{2}{9}\Phi_i(D_i)] > \frac{1}{4}$, so since $\Phi_i = \Phi_i(D_i) + \Phi_i(A_i) + \Phi_i(S_i)$, with probability at least $\frac{1}{4}$, the potential decreases by a factor of at least $\frac{2}{9}$.

Define a *successful* phase to be a phase in which the total potential decreases by a factor of at least $\frac{2}{9}$. Since the starting potential is 3^{2S^*-1} and the final potential is 0 and potential is always an integer, we require at most

$$(2S^* - 1)\log_{9/7} 3 < 10S^* - 5 \in O(S^*) \in O(S(1 + \lg U))$$

successful phases. Since each phase has $\Theta(PU)$ steal attempts, there are $O(PSU(1+\lg U))$ steal attempts in expectation. The desired bound on the number of rounds follows from Lemma 1. □

For the case $U = 1$, no pfor trees will be added and each worker will maintain exactly one deque because when a worker's deque becomes empty, it will always be freed and subsequently reused for the stolen work. A bound of $O(\frac{W}{P} + S)$ rounds for this case can be shown using the same techniques as for the unweighted case [3, 5]

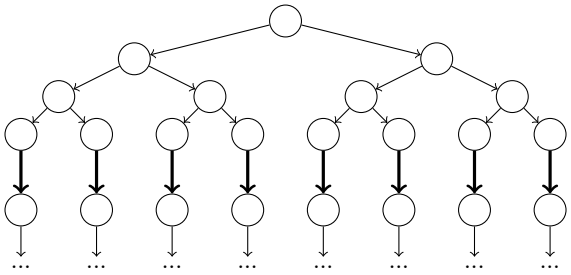


Figure 7: Partial dag for map and reduce

```
1  function distMapReduce(f, g, id, lo, hi)
2    n = hi - lo
3    if n = 0 then return id
4    else if n = 1 then
5      x = getValue(lo) // May suspend
6      return f(x)
7    else
8      piv = (lo + hi) / 2
9      (res1, res2) =
10       fork2(distMapReduce(f, g, id, lo, piv),
11             distMapReduce(f, g, id, piv, hi))
12     return g(res1, res2)
```

Figure 8: Example: distributed map and reduce.

4.3 Running time

Each round of the scheduler takes (amortized) time $O(1)$.

THEOREM 3. *If I_p is the number of instructions executed by worker p, then $I_p \in O(\frac{W}{P} + SU(1 + \lg U))$.*

PROOF. All operations outside of calls to `addResumedVertices()` and callbacks are constant-time. We amortize the non-constant work done by a worker over the rest of the work done by the worker. In particular, we "charge" the work done for a suspended vertex v to v's parent (which was executed by the same worker and spawns at most two children). See the companion technical report for a more detailed version of the proof. □

5. EXAMPLES

The two examples in this section illustrate an application of our techniques and also show how to achieve extremal values of U, which is relevant because the time bound of our scheduler depends on U.

Distributed map and reduce. Consider a computation in which we wish to map a function $f(x)$ over each of a large set of n values x, and then combine the resulting values with an associative binary operation $g(x, y)$ with identity id. To introduce latency, suppose that each value is stored on a different remote server (or must be obtained from a remote user). Pseudocode for this example is shown in Figure 8. In this computation, it is possible for each of the n calls to `getValue()` to be suspended at once, and so $U = n$. The top of the dag for this example is shown in Figure 7.

Server. For an example that minimizes U, consider a program similar to the map and reduce example, but which takes its inputs one-by-one from a user. This code appears

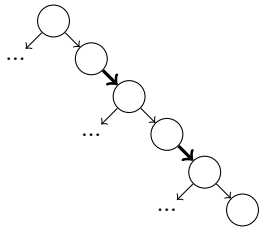


Figure 9: Partial dag for the "server"

```
1 function server(f, g)
2   input = getInput() // May suspend
3   if input = "Done" then return 0
4   else
5     (res1, res2) = fork2(f(input), server(f, g))
6     return g(res1, res2)
```

Figure 10: Example: the "server"

in Figure 10. The code gets input from the user. If the user enters "Done", the identity value is returned up the tree. If the user enters a number x, the computation forks; in parallel, $f(x)$ is computed, and a recursive instance of the server is run. In the end, all of the branches join and the values are reduced with g. In this computation, getting user input may incur a substantial amount of latency. However, `server()` is not called recursively until after `getInput()` returns, and so only one operation may be suspended at a time and $U = 1$. Part of this dag is shown in Figure 9, with ... indicating branches that lead to $f(x)$ computations.

6. IMPLEMENTATION

We have developed a proof-of-concept implementation of our scheduling algorithm by extending a more traditional work-stealing algorithm developed by Spoonhower et al. [29, 28] for a parallel implementation of Standard ML. The language has support for fork-join parallelism and futures. Their scheduler and our extension are written entirely in Standard ML. Our implementation largely follows the description in Section 3 with a few exceptions.

- As usual, our scheduler operates at the granularity of threads rather than instructions and is only invoked when the current thread ends, requires synchronization (with another thread) or suspends.
- Suspended threads are placed in a list, tagged with the event on which they are waiting. Each event is polled when the scheduler is invoked. The overhead of this operation is acceptable for reasonable suspension widths and task granularities.
- We sometimes use theoretically less efficient data structures or policies, favoring simplicity and practicality.
- Rather than targeting a random deque, steals target a worker and then choose randomly from that worker's *ready deques*. This requires synchronization between the two workers but decreases the number of failed steals because steals won't target empty deques.

6.1 Preliminary experiments

We implemented a variant of the map and reduce example of Section 5 and performed a preliminary experimental evaluation. In our implementation of this benchmark, we calculate the Fibonacci number of each input using the naive recursive parallel Fibonacci algorithm and sum the results modulo a large constant. In order to control the latency associated with taking each input, the benchmark simulates a latency of δ milliseconds by sleeping for δ milliseconds and then immediately returning 30. Thus each Fibonacci calculation computes the 30^{th} Fibonacci number. We have run this benchmark using both our latency-hiding scheduler and Spoonhower's traditional work-stealing scheduler.

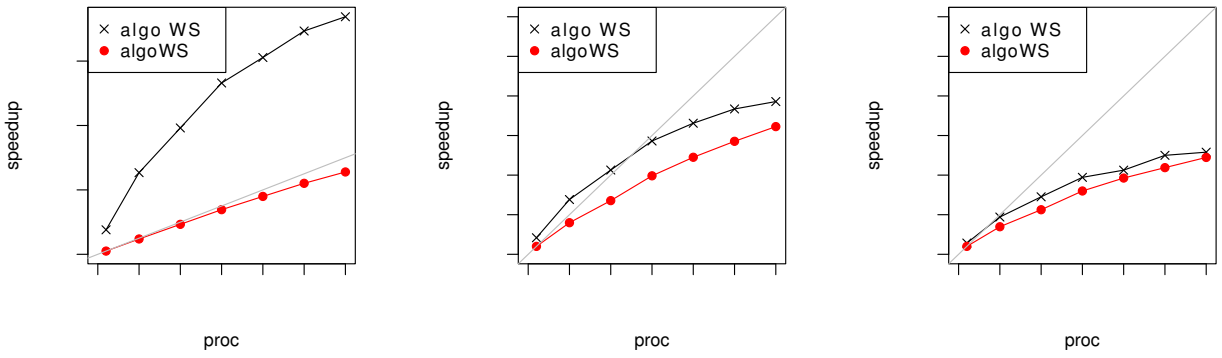
Figure 11 shows the self-speedup curves for our scheduler, labeled LHWS, and the standard work-stealing scheduler, labeled WS. For all curves, the speedup shown is relative to the one-processor run of WS. In all of these experiments, the number of elements (remote server connections)is 5,000. Note that, in this example, this is equal to the suspension width. The first (leftmost) plot shows the case when the latency δ is 500ms (a very high latency, which might represent waiting for user input or for data which requires some computation on a remote server). This plot shows that latency-hiding work stealing delivers superlinear speedups, as much as 3 times larger speedup than standard work stealing. The superlinear speedups are expected with latency hiding because the standard work stealer does not hide latency. In the second plot, δ is 50ms. Latency-hiding still provides substantial speedup benefit. The third plot shows that, when δ is smaller (1ms), there is less benefit to hiding latency. In the limit, if latency is expected to be small, programmers might wish to wait for operations to complete.

7. RELATED WORK

We know of relatively little other work that has considered the problem of scheduling user-level parallel computations which involve operations that may incur a latency.

In terms of the models and analysis presented, perhaps the most closely related work to ours is BATCHER [4], which uses work stealing to schedule threads that perform accesses on a concurrent data structure. In order to avoid synchronizing concurrent accesses, accesses are batched and performed all at once. As a result, an operation on the data structure will block, incurring a latency, until the next batch of operations is executed. A major difference with our work is that, in BATCHER, the scheduler decides when to execute a batch and thus has some control over latency, whereas our scheduler has no control over the latency of an operation. Another difference is that their scheduling algorithm employs two deques per worker, whereas in our algorithm the number of deques can grow dynamically based on the suspension width of the dag and the scheduling decisions.

Many other variants of work stealing have been considered in the literature (e.g., [9, 17, 15, 7, 2]). Mattheis, et al. [24] investigate modifications to work stealing in order to make it suitable for real-time applications, in particular stream processing. They focus on minimizing the completion time (response time) of particular threads; the latency incurred by threads themselves is not considered. Spoonhower [28] presents a comprehensive treatment of work stealing and considers variants with multiple deques per worker. The goal of these variations is to minimize the number of *deviations* from a single-processor depth-first order. In one variation, when a thread waits for another thread or future, the entire deque is suspended and a new one is created. In another,



$\delta = 500$ms $\delta = 50$ms $\delta = 1$ms

Figure 11: Experimental results of prototype implementation

when a suspended thread resumes, a new deque is created to execute it. Neither of these exactly corresponds to our approach, where a delay does not suspend an entire deque, and new deques are created on steals, not resumes.

In more remotely related work, scheduling for reduced latency has been an important consideration in queuing-theoretic scheduling. For an excellent review of this vast literature, we refer the interested reader to Harchol-Balter's book [18]. The focus of the research in this area is minimizing the latency of a job, i.e., the response time, rather than scheduling a single parallel job that includes operations that can incur latency, which is our focus. Most of the work on queuing-theoretic scheduling considers sequential jobs. One exception is the work on gang scheduling in the context of supercomputing. Feitelson, Rudolph and Schwiegelshohn have a nice survey on this topic [13]. Gang scheduling focuses primarily on the problem of scheduling parallel jobs on a number of processors, rather than scheduling a single parallel job for fast completion.

Many systems use lightweight threads, which allow latency hiding but have higher overhead than fine-grained tasks. As an intermediate approach, Concurrent Cilk [32] schedules tasks using work stealing, but lazily promotes tasks to lightweight threads when they perform blocking operations. Since this is essentially accomplished by blocking the deque of the suspended task and spawning a new lightweight worker thread to start on a new deque, the approach is similar to our algorithm, but Concurrent Cilk spawns new worker threads more eagerly than our algorithm creates new deques. Our approach, which continues to schedule suspended tasks as fine-grained tasks, also avoids the additional state and thread-scheduling overhead associated with (even lightweight) threads.

8. CONCLUSION

In this work, we have extended the traditional dag model of parallel computation to model computations with latency-incurring operations such as I/O, secondary storage access and communication with remote machines. We have also developed a *latency-hiding work-stealing* scheduler, which can efficiently schedule latency-incurring computations without stalling workers when one thread incurs latency and suspends. Our algorithm generalizes standard work-stealing schedulers to use multiple deques per worker, and reduces to the standard algorithm on computations in which no threads suspend. Our theoretical analysis shows that the scheduler is able to hide latency from the typically dominant "work" component but can incur an overhead for latency that falls on the critical path. Finally, our prototype implementation and initial evaluation give evidence that this approach can be practical and has the potential to significantly improve the performance of computations that perform latency-incurring operations without penalizing the computations that don't incur such latency.

Acknowledgments

This research is partially supported by the National Science Foundation under grant numbers CCF-1320563 and CCF-1408940 and by Microsoft Research.

We would also like to thank Guilherme Rito for his contributions to early versions of this work, as well as Guy Blelloch, Robert Harper, Angelina Lee and the anonymous reviewers for their helpful feedback.

References

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

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

[3] U. A. Acar and S. K. Muller. A proof of work stealing for dedicated multiprocessors. Technical Report CMU-CS-16-114, Carnegie Mellon University School of Computer Science, Dec. 2016.

[4] K. Agrawal, J. T. Fineman, K. Lu, B. Sheridan, J. Sukha, and R. Utterback. Provably good scheduling for parallel programs that use data structures through

implicit batching. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 84–95, 2014.

[5] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.

[6] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366, 2011.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, Sept. 1999.

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

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

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005.

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

[12] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computing*, 38(3):408–423, 1989.

[13] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling - A status report. In *Job Scheduling Strategies for Parallel Processing (JSSPP), 10th International Workshop*, pages 1–16, 2004.

[14] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011.

[15] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. PLDI '98, pages 212–223, 1998.

[16] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Trans. Program. Lang. Syst.*, 21(2):240–285, Mar. 1999.

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

[18] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action.* Cambridge University Press, 2013.

[19] S. M. Imam and V. Sarkar. Habanero-Java library: a Java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, pages 75–86, 2014.

[20] Intel. *Intel Threading Building Blocks*, 2011.

[21] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. 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.

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

[23] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, 2009.

[24] S. Mattheis, T. Schuele, A. Raabe, T. Henties, and U. Gleim. Work stealing strategies for parallel stream processing in soft real-time systems. In *Proceedings of the 25th International Conference on Architecture of Computing Systems*, ARCS'12, pages 172–183, Berlin, Heidelberg, 2012. Springer-Verlag.

[25] S. K. Muller and U. A. Acar. Latency-hiding work stealing. Technical Report CMU-CS-16-112, Carnegie Mellon University School of Computer Science, May 2015.

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

[27] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts (7. ed.)*. Wiley, 2005.

[28] D. Spoonhower. *Scheduling Deterministic Parallel Programs.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2009.

[29] D. Spoonhower, G. E. Blelloch, R. Harper, and P. B. Gibbons. Space profiling for parallel functional programs. In *International Conference on Functional Programming*, 2008.

[30] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 and APGAS at petascale. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 53–66, 2014.

[31] J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

[32] C. Zakian, T. A. K. Zakian, A. Kulkarni, B. Chamith, and R. R. Newton. Concurrent Cilk: Lazy promotion from tasks to threads in C/C++. In *The 28th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2015)*, 09/2015 2015.