Provably Good Scheduling for Parallel Programs that Use Data Structures through Implicit Batching

Kunal Agrawal
Washington Univ. in St. Louis
kunal@cse.wustl.edu

Jeremy T. Fineman
Georgetown University
jfineman@cs.georgetown.edu

Kefu Lu
Washington Univ. in St. Louis
kefu.lu@wustl.edu

Brendan Sheridan
Georgetown University
bss45@georgetown.edu

Jim Sukha
Intel Corporation
jim.sukha@intel.com

Robert Utterback
Washington Univ. in St. Louis
robert.utterback@wustl.edu

ABSTRACT

Although *concurrent data structures* are commonly used in practice on shared-memory machines, even the most efficient concurrent structures often lack performance theorems guaranteeing linear speedup for the enclosing parallel program. Moreover, efficient concurrent data structures are difficult to design. In contrast, *parallel batched data structures* do provide provable performance guarantees, since processing a batch in parallel is easier than dealing with the arbitrary asynchrony of concurrent accesses. They can limit programmability, however, since restructuring a parallel program to use batched data structure instead of a concurrent data structure can often be difficult or even infeasible.

This paper presents BATCHER, a scheduler that achieves the best of both worlds through the idea of *implicit batching*, and a corresponding general performance theorem. BATCHER takes as input (1) a dynamically multithreaded program that makes arbitrary parallel accesses to an abstract data type, and (2) an implementation of the abstract data type as a batched data structure that need not cope with concurrent accesses. BATCHER extends a randomized work-stealing scheduler and guarantees provably good performance to parallel algorithms that use these data structures. In particular, suppose a parallel algorithm has T_1 work, T_∞ span, and n data-structure operations. Let $W(n)$ be the total work of data-structure operations and let $s(n)$ be the span of a size-P batch. Then BATCHER executes the program in $O((T_1 + W(n) + ns(n))/P + s(n)T_\infty)$ expected time on P processors. For higher-cost data structures like search trees and large enough n, this bound becomes $((T_1 + n\lg n)/P + T_\infty \lg n)$, provably matching the work of a sequential search tree but with nearly linear speedup, even though the data structure is accessed concurrently. The BATCHER runtime bound also readily extends to data structures with amortized bounds.

SPAA'14, June 23–25, 2014, Prague, Czech Republic.

ACM 978-1-4503-2821-0/14/06 ...$15.00.
http://dx.doi.org/10.1145/2612669.2612688.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; E.1 [**Data Structures**]: *Distributed data structures*

Keywords

Data structures, work stealing, scheduler, batched data structure, implicit batching

1. INTRODUCTION

A common approach when using data structures within parallel programs is to employ ***concurrent data structures*** — data structures that can cope with multiple simultaneous accesses. Not only is it challenging to design and analyze concurrent data structures, but the existing performance theorems do not often imply linear speedup for the enclosing program. The straightforward way to analyze a program that uses a concurrent data structure is to assume the worst-case latency for each access. For a limited set of concurrent data structures (see Section 6), the worst-case latency is low enough that this approach is effective. In more general cases, the worst-case latency is often linear in the number of processors in the system (or worse), e.g., for Braginsky and Petrank's lock-free B$^+$-tree [7].[1] If n data-structure accesses each keep a processor busy for $\Omega(P)$ timesteps, then the running time on P processors is at least $\Omega(nP/P) = \Omega(n)$. An $\Omega(n)$ bound means that accesses are essentially sequential — there is no significant speedup guarantee when running on P processors.

Concurrent data structures are in some sense overkill for use within a single parallel program because they are designed to cope with difficult access patterns. Since the data-structure accesses belong to the same enclosing program, they can, in principle, coordinate with each other. A key idea behind our work is to leverage a runtime scheduler and to handle this coordination.

The primary goal of this paper is to 1) describe a good scheduler for executing a broad class of parallel programs that make parallel

[1]The worst case occurs when all processors concurrently insert contiguous keys. The progress bound proven [7] is worse, since the data structure is designed to cope with processor failures. But even assuming no failures and simplifying the data structure, an $\Omega(P)$ worst-case bound still occurs when P processes attempt a compare-and-swap on the same node in the tree.

accesses to data structures, and 2) provide a corresponding general performance theorem for this scheduler. Our performance theorem exhibits an attractive modularity: the data structure may be analyzed in isolation of the program that uses it, and the parallel program may be analyzed without considering the specific implementation of the data structure. This modularity makes the theorem easy to apply while still achieving provably good speedup, e.g., for n parallel accesses to a search tree with large enough n, our scheduling theorem proves a completion time of $\Theta(n \lg n/P)$, which is asymptotically optimal and has linear speedup. We are unaware of any comparable aggregate bounds for concurrent search trees.

Runtime scheduling. This paper focuses on parallel programs expressed through ***dynamic multithreading***[2] (see [11, Ch. 27]), which is common in many parallel languages and libraries, such as Cilk dialects [18, 26], Intel TBB [35], Microsoft Task Parallel Library [39] and subsets of OpenMP [30]. The programmer expresses algorithmic parallelism, through linguistic constructs such as "spawn" and "sync," "fork" and "join," or parallel loops. The programmer does not provide any mapping from subcomputations to processors. The program is typically scheduled using an efficient work-stealing scheduler (e.g., [6]) provided by the runtime system. A parallel program (without parallel data structure accesses) having T_1 ***work*** — the running time on 1 processor, and T_∞ ***span*** — the length of the critical path, can be executed in $O(T_1/P + T_\infty)$ (expected) time on P processors (or workers) using a work-stealing scheduler. This running time is asymptotically optimal and guarantees linear speedup to programs with sufficient parallelism.

This paper generalizes the above result by describing a scheduler and corresponding performance theorem for dynamic multithreaded programs that make parallel accesses to a data structure. This performance theorem implies that parallel programs (with sufficient parallelism) using efficient data-structure implementations can execute with linear speedup, even if the program's work is dominated by data-structure accesses.

Batched data structures. One way to programmatically coordinate data-structure accesses is to use only ***batched data structures***, where the program invokes an entire set of data-structure operations synchronously, and the batched data structure performs these operations collectively in parallel. The main advantages of batched data structures are: (1) only one batch is active at a time, obviating the need for complicated concurrency control within the data structure, (2) parallelism may be used to accelerate individual batches, and (3) they are relatively easy to analyze; if a program generates a sequence of batches, we can simply add the running time of this sequence of batches to the running time of the program. For example, (batched) parallel priority queues [8, 12, 13, 36] have been utilized to prove efficient running time on parallel algorithms such as shortest paths and minimum spanning tree [8, 13, 32].

Getting provably good performance by replacing a concurrent data structure with a batched data structure can require drastic code restructuring, however, since the parallel program must explicitly group accesses into batches. In some cases, such a restructuring may not even be possible. For example, an on-the-fly race detector [29, 5, 34] updates a series-parallel-maintenance data structure on forks and joins while executing an input program. In this application, the data structure must be updated before the program flow continues past the calling point, so it seems impossible to reorganize the operations into batches by restructuring the algorithm.

[2]Dynamic multithreading is also sometimes called "fork-join" parallelism.

Contributions

This paper shows we can achieve a modular analytic abstraction whereby the data structure $\mathcal{D}$ and the enclosing program $\mathcal{C}$ may be analyzed separated, then combined through a strong performance theorem guaranteeing good parallel running time. This result applies to the broad class of dynamically multithreaded computations. We achieve the runtime theorem through the novel technique of implicit batching coupled with an efficient runtime scheduler.

Implicit batching. This paper focuses on the novel technique of ***implicit batching***, which achieves benefits of both concurrent and batched data structures. In implicit batching, the programmer provides two components: (1) a parallel program $\mathcal{C}$ containing parallel accesses to a abstract data type $\mathcal{D}$, and (2) a batched data-structure implementing the data structure $\mathcal{D}$. The scheduler dynamically and transparently organizes the program's parallel accesses to the data structure into batches, with at most one batch executing at a time.

Using implicit batching gives the benefits of batched data structures without restructuring the enclosing program $\mathcal{C}$. The scheduler is responsible for grouping any concurrent accesses to the abstract data type $\mathcal{D}$ into batches and invoking the appropriate implementation of a batched operation. The data structure's batched operation may be implemented using dynamic multithreading (see Section 3 for examples). The data-structure's implementation need not cope with concurrency since at most one batch is executing at any time, and hence locks or atomic operations may be omitted. The scheduler handles all synchronization and communication between the parallel program $\mathcal{C}$ and the data structure $\mathcal{D}$.

Implicit batching closely resembles flat combining [21], where each concurrent access to a data-structure queues up in a list of operation records, and this list of records (i.e., a batch) is later executed sequentially. Implicit batching may be viewed as a generalization of flat combining in that it allows *parallel* implementations of batched operations, instead of only a sequential one allowed by flat combining. Due to sequential batches, flat combining does not guarantee provable speedup guarantees. However, flat combining has been shown to be more efficient in practice than some of the best concurrent data structures under certain loads. Viewing flat combining as a specific implementation of implicit batching, already shows the practical effectiveness of implicit batching — this paper focuses on obtaining a provably good runtime theorem.

Scheduler and performance theorem. Implicit batching poses its own challenges for performance analysis. For a parallel program using implicit batching, which sequence of batches should be analyzed? What overhead does the scheduler incur when creating batches? In general, the performance of a parallel program using implicit batching depends on the particular runtime scheduler used to execute the program.

To yield a performance theorem, we propose ***BATCHER***, a work-stealing scheduler designed for implicit batching. Given a dynamically multithreaded program $\mathcal{C}$ that makes parallel accesses to an ADT $\mathcal{D}$, and a batched implementation of $\mathcal{D}$, it yields the following bound, proven in Section 5:

THEOREM 1. *Consider a dynamically multithreaded program $\mathcal{C}$ having T_1 work and T_∞ span. Let n be the total number of data-structure operations (accesses to ADT $\mathcal{D}$), and m be the maximum number of data-structure operations along any sequential dependency chain. For the given implementation of $\mathcal{D}$, let $W(n)$ be the worst-case total work for n data-structure operations grouped arbitrarily into batches, and let $s(n)$ be the worst-case span of a parallel size-P batched operation.*[3] *Then the expected running time of*

[3]We employ only binary forking, so $s(n) \geq \lg P$ implicitly.

this program on P processors using BATCHER[4] *is at most*

$$O\left(\frac{T_1}{P} + T_\infty + \frac{W(n) + ns(n)}{P} + ms(n)\right) .$$

An important feature about this bound is that T_1 and T_∞ are the work and span of the core program $\mathcal{C}$, independent of the data structure implementation. Similarly, n and m count the data-structure calls in the program, and depend only on the program $\mathcal{C}$, not the data structure implementation. Moreover, $s(n)$ and $W(n)$ are performance measures of the batched implementation $\mathcal{D}$ of the data structure, independent of the enclosing program. We are thus essentially adding the program's cost to the data structure's cost. This bound also applies when the analysis of the batched data structure is amortized, through a more general definition of $s(n)$ (see Section 2 for definitions). For many parallel batched data structures (see Section 3 for examples), this performance theorem implies nearly linear speedup.

The remainder of this paper is organized as follows. Section 2 presents the theoretical model we use to analyze parallel programs that make accesses to parallel data structures. Section 3 provides a high-level overview of BATCHER and applies the performance bound to example batched data structure. Section 4 presents the BATCHER scheduling algorithms, which is analyzed in Section 5; we built a prototype implementation of BATCHER and show preliminary experiments in Section 7. Section 6 discusses related work on using data structures in parallel.

2. DEFINITIONS AND ANALYTIC MODEL

Recall that a programmer provides two inputs to the BATCHER scheduler: (1) A parallel program $\mathcal{C}$ that makes parallel accesses to an abstract data type $\mathcal{D}$, and (2) a batched data structure that implements $\mathcal{D}$ and need only support one batch at a time. This section defines how the parallel program and the batched data structure are modeled.

Execution dag model. In the absence of data-structure operations, the execution of a dynamically multithreaded computation can be modeled as a directed acyclic graph (***dag***) that unfolds dynamically (see [11, Ch. 27]). In this execution dag, each node represents a unit-time sequential subcomputation, and each edge represents control-flow dependencies between nodes. A node that corresponds to a "fork" has two[5] outgoing edges, and a node corresponding to a "join" has two or more incoming edges.

A scheduler is responsible for choosing which nodes to execute on each processor during each timestep. The scheduler may only execute ***ready nodes*** — those unexecuted nodes whose predecessors have all been executed. The convenient feature about the computation dag is that it models the control-flow constraints within the program without capturing the specific choices made by the scheduler. The dag unfolds dynamically — only the immediate successors of executed nodes are revealed to the scheduler. This unfolding can also be nondeterministic. Hence the scheduler must make online decisions. All of our analyses are with respect to the *a posteriori* dag. The two key features of a dag are its ***work***, which is the number of (unit-length) nodes in the dag, and its ***span***, which is the length of the longest directed path through the dag.

[4]Just as in standard work-stealing results, our theoretical bounds assume that the only synchronization of the input algorithm occurs through "syncs" or "joins"; the algorithm or data structure code itself does not use explicit synchronization primitives, e.g., locks or compare-and-swaps.

[5]In general, forks may have arbitrary out-degree, but in this paper we pessimistically assume binary forking.

Extending the dag model to implicit batching. We first model the batched data structure that implements $\mathcal{D}$. An implementation of a batched operation is itself a parallel (sub)computation that may include forks and joins. We thus model the execution of each batch A by its own ***batch dag*** G_A. We use the terms ***batch work***, denoted by w_A, and ***batch span***, denoted by s_A, to refer to the work and span of the batch dag, respectively.

To analyze a batched data structure as a whole, we consider worst-case sequences of arbitrary batches, such that the total number of data structure operations across all batches is n, and each batch contains at most P data structure operations. We define the ***data-structure work***, denoted by $W_P(n)$, to be the maximum total work of any such sequence of batches. We also define the ***data-structure span***, denoted by $s_P(n)$, to be the worst-case span of any batch dag A in any such sequence subject to the the restrictions that $w_A/s_A = O(P)$, meaning that the batch has limited parallelism. In the case when the data structure's analysis is not amortized, the data-structure span may be stated more concisely as the worst-case span of any batch dag that represents a size-P batch, since all batches of the same size have the same span. For data structures with amortized analysis, however, batches with the same number of operations may have different spans — therefore, the batch span is defined in terms of the parallelism of the batch dag rather than the number of operations in the batch. Since P (the number of workers) is static throughout this paper, we use $W(n)$ and $s(n)$ as shorthands for $W_P(n)$ and $s_P(n)$. Note that whereas data-structure work corresponds to the total work of all batches that cumulatively contain n operations, the data-structure span corresponds to the span of a single batch with P operations. Thus far we have not considered a program that makes accesses to the data structure, we have only considered the data structure implementation. It should thus be clear that $W(n)$ and $s(n)$ are metrics of the data structure implementation itself.

We model the enclosing program $\mathcal{C}$, which makes parallel calls to a data structure (these operations will be implicitly batched), as another kind of dag, called the ***core dag*** G. A core dag is just like a standard execution dag, except that it includes two kinds of nodes. Each data-structure operation (that is to be implicitly batched) is represented by a special ***data-structure node***. All other non-data-structure nodes in the dag are called ***core nodes***. Whereas all core nodes by definition take unit-time to execute on a processor, the data-structure nodes represent blocking calls that may take longer to complete. Our metrics for the core dag, however, avoid this issue — we define the ***core work***, which we generally denote by T_1, to be the number of nodes in the core dag, and the ***core span***, denoted by T_∞, to be the longest path through the core dag in terms of number of nodes. We also generally use n to refer to the number of data-structure nodes in the core dag, and m to denote the maximum number of data-structure nodes falling along any directed path through G. Although the core dag includes data-structure nodes whose "execution times" are not defined, the metrics T_1, T_∞, n, and m are functions of only the core program, not the implementation of batched operations.

No extra dependencies between data-structure nodes? It may be surprising that modeling the core dag and batch dags separately as described throughout this section can be sufficient for the analysis of any scheduler. *A priori*, one might expect execution-dependent "happens-before" relationships across all data-structure calls, particularly since the scheduler must group operations into batches. Moreover, one might be surprised that the "length" of data-structure nodes is not modeled anywhere. Nevertheless, in Section 5 we prove that this simple model is sufficient for the BATCHER scheduler, which is a key contribution of the paper.

3. IMPLICIT BATCHING IN BATCHER

This section overviews implicit batching in the context of the BATCHER scheduler with respect to the core and batch dags defined in Section 2. The specific algorithms employed by the scheduler itself are deferred to Section 4. This section also gives a simple example of a program using an implicitly batched data structure to provide concrete examples of applying the performance theorem.

Programming Interface. BATCHER provides distinct interfaces to the algorithm programmer, who writes a program $\mathcal{C}$ that makes parallel accesses to ADT $\mathcal{D}$; and the data-structure programmer, who provides the batched implementation of ADT $\mathcal{D}$. The runtime system stitches together these interfaces and does the scheduling. Figures 1 and 2 (discussed later in this section) show a simple example program making n parallel increments to a shared counter using this interface style.

To perform a data-structure operation, the program $\mathcal{C}$ makes a call into the runtime system, denoted by BATCHIFY here. As far as the algorithm programmer is concerned, BATCHIFY (corresponding to a data-structure node in the core dag) resembles a normal procedure call to access a *concurrent data structure*, and the control flow blocks at this point until the operation completes.

A BATCHER data structure, on the other hand, must provide an implementation of a parallel batched operation, which we denote by BOP. Since BOP is a batched implementation, it takes as input a set (i.e., an array) of operations to the ADT $\mathcal{D}$ to perform. Note that BOP is itself a dynamically multithreaded function that can use spawn/sync or parallel loops to generate parallelism. A single invocation of the `Bop` function corresponds to a single batch dag.

Batching. At a high level, calls to BATCHIFY correspond to data-structure nodes and BATCHER is responsible for implicitly batching these data structure operations and then executing these batches by calling BOP. When a worker p encounters a data-structure node u (i.e., p executes a call to BATCHIFY), p alerts the scheduler to the operation by creating an operation record op for that operation and placing it in a particular memory location reserved for this processor. Eventually, op will be part of some batch A and the scheduler will call BOP on A. Unlike core nodes, however, the data-structure node can logically block for longer than one time step and u's successor(s) in the dag do not become ready until after this call to BOP returns, that is, the operation corresponding to u is actually performed on the data structure as part of a batch. Thus from the perspective of the core program, a data-structure node u has the same semantics as a blocking access to a concurrent data structure.

Inherent to implicit batching is the idea that the batch the scheduler invokes only one batch at a time. Hence the data-structure implementation need not cope with concurrency, simplifying the data-structure design. The following invariant states this property for BATCHER.

INVARIANT 1. *At any time during a BATCHER execution, at most one batch is executing.*

There are many other choices that go into a scheduler for implicit batching. For BATCHER, we made specific choices guided by the goal of proving a performance theorem. Three of the main questions are what basic type of scheduler to use, how large batches should be, and when and how batches are launched. As far as the low-level details are concerned, we chose in favor of simplicity where possible. BATCHER is a distributed work-stealing scheduler. BATCHER also restricts batch sizes, as stated by the following invariant; this size cap ameliorates application of the main theorem as it simplifies the analysis of any specific data structure.

INVARIANT 2. *In a BATCHER execution, batches contain at most P data-structure nodes.*

Finally, whenever an operation record is created and no batch is currently in progress, BATCHER immediately launches a new batch; it does not wait for a certain number of operations to accrue; this decision is important for the theoretical analysis. Therefore, batches can contain as few as one operation. Launching a batch includes some (parallel) setup to gather all operation outstanding operation records, executing the provided (parallel) batched operation BOP thereby inducing a batch dag, and some (parallel) cleanup after completing. Since the setup/cleanup overhead is scheduler dependent, we account for the overhead separately, and the batch dag comprises only the steps of BOP.

Intuition behind the analysis. The analysis of BATCHER (Section 5) relies on specific features of the scheduling algorithm (Section 4). Nevertheless, we have already exposed one significant difficulty: since batches launch as soon as possible, some batches may contain just a single data-structure node. If this were true for every batch, then all operations would be sequentialized according to Invariant 1, and it would seem impossible to show good speedup. In addition, the batch setup and cleanup overhead is the same, regardless of batch size; therefore, having many small batches may incur significant overhead.

Fortunately, small batches fall into two cases, both being good. (1) Many data structure nodes accrue while a small batch is executing. These will be part of the next batch, meaning that the next batch will be large and make progress toward the batch work $W(n)$. (2) Not many data structure nodes are accruing. Then the core dag is not blocked on too many data-structure nodes, and progress is being made on the core work T_1. In both cases, the setup and cleanup overhead of the small batch can be amortized either against the work done in the next batch or the work done in the core dag.

Example and applying the performance bound

To understand the BATCHER performance bound (Theorem 1), let us turn to some specific examples. We are not developing new batched data structures here — the point is only to illustrate the power of batched data structures, and to see how to apply the bound.

As a simple example, consider a core program that makes n completely parallel increments to a shared counter, as given by Figure 1. This example is for illustration only, and is not intended to be very deep. This program has $\Theta(n)$ core work and $\Theta(\lg n)$ core span (with binary forking). The shared counter is an abstract data type that supports a single operation INCREMENT, which atomically adds a value (possibly negative) to the counter and returns its current value.

Concurrent counter. A trivial concurrent counter uses atomic primitives like fetch-and-add to INCREMENT. If the primitive is mutually exclusive (which is true for fetch-and-add on current hardware), then n INCREMENTs take $\Omega(n)$ time. The total running time of the program is thus $\Omega(n)$ regardless of the number of processors.

One could instead use a provably efficient concurrent counter, e.g., by using the more complicated combining funnels [37, 38]. Doing so would indeed yield a good overall running time, but these techniques are not applicable to more general data structures. As we shall see next, the implicitly batched counter achieves good asymptotic speedup with a trivial implementation.

Batched counter. Figure 2 shows a sample batched counter. Here, when the core program makes an INCREMENT call, it creates an operation record which is handed-off to the scheduler. The scheduler later runs the batch increment BOP on a set of increments. The main subroutine of the batched operation is "parallel prefix sums",

```
1  parallel_for i = 1 to n
2      do B[i] = INCREMENT(A[i])
```

Figure 1: A parallel loop that performs n parallel updates to a shared counter. Here, $A[1..n]$ is an array of values by which to increment (or decrement if negative) the counter, and $B[1..n]$ holds any return values from the INCREMENTs.

```
3  struct OpRecord {int value; int result;}

INCREMENT(int x)
4  OpRecord op
5  op.value = x
6  BATCHIFY(this, op) //ask the scheduler to batch op
7  return op.result

BOP(OpRecord D[1..size])
 8  let v be the value of the counter
 9  D[1]'s value field = v + D[1]'s value
10  perform parallel-prefix-sums on value fields of D[1..size],
                storing sums into result fields of D[1..size]
    // now D[i]'s result = sum_{k=1}^{i} D[k]'s value
11  set the counter to D[size]'s result
```

Figure 2: A batched-counter implementation. As we shall see in Section 4, line 6 *logically* blocks, but the processor does not spin-wait. The BOP is called by the scheduler automatically.

which ***in parallel*** computes $\sum_{k=1}^{i} D[k]$ for every i. It is easy to prove that returning $\sum_{k=1}^{i} D[k]$ yields linearizable [25] counter operations. Prefix sums is a commonly used and powerful primitive in parallel algorithms, and hence we consider this 4-line implementation of BOP to be trivial. Adaptations of Ladner and Fischer's approach to prefix sums [28] to the fork-join model have $O(x)$ work and $(\lg x)$ span for x elements.

To analyze the execution of this program using BATCHER, we need only bound $W(n)$, the total work of arbitrarily batching n operations, and $s(n)$, the span of a batched operation that processes P operation records (performs P increment operations). Since the work of prefix sums is linear, we have $W(n) = \Theta(n)$. Since a size-P batch has $O(\lg P)$ span (dominated by prefix sums), we have $s(n) = O(\lg P)$. We thus get the bound $O(\frac{T_1+n\lg P}{P} + m\lg P + T_\infty)$ for performing n INCREMENTs, with at most m along any path. The core dag of Figure 1 has $T_1 = O(n)$, $T_\infty = O(\lg n)$, $m = 1$, so we have a running time of $O(\frac{n\lg P}{P} + \lg n)$ for $n > P$. This nearly linear speedup is much better than for the trivial counter.

Applying BATCHER to a search tree. There exists an efficient batched 2-3 tree [33] in the PRAM model, and it is not too hard to adapt this algorithm to dynamic multithreading. The main challenge in a search tree is when all inserts occur in the same node of the tree, e.g., when inserting P identical keys. The main idea of this batched search tree is to first sort the new elements, then insert the middle element and recurse on each half of the remaining elements. This process allows for each of the new keys to be separated by existing keys without concurrency control. It is not obvious how to leverage the same idea in a concurrent search tree.

See [33] for details of the batched search tree. Suffice it to say that a size-x batch is dominated by two steps: 1) a parallel search for the location of each key in the tree, having $O(x\lg n)$ work and $O(\lg n + \lg x)$ span, and 2) a parallel sort of the x keys, having $O(x\lg x)$ work. The data-structure span is thus $s(n) = O(\lg n + sort(P))$, where $sort(P) = O(\lg P \lg\lg P)$ [10] is the span of a parallel sort on P elements in the dynamic-multithreading model. The data-structure work $W(n)$ is maximized for n/P batches of size $x = P$, yielding $W(n) = O(n\lg n)$ data-structure work. Applying Theorem 1, we get a running time of $O(\frac{T_1+W(n)+s(n)}{P} + ms(n) + T_\infty) = O(\frac{T_1+n\lg n+n\lg P\lg\lg P}{P} + m\lg n + m\lg P\lg\lg P + T_\infty)$. For large enough n (specifically, $n = \Omega(P^{\lg\lg P})$, this reduces to $O(\frac{T_1+n\lg n}{P} + m\lg n + T_\infty)$, which is asymptotically optimal in the comparison model and provides linear speedup for programs with sufficient parallelism. For instance, a program obtained by substituting the increment operation with an insert in Figure 1 would yield the running time of $O(n\lg n/P)$, implying linear speedup, even though the program only performs data structure accesses.

Amortized LIFO stack. We now briefly describe an example, namely a LIFO stack, which has amortized performance bounds. The data structure is an array that supports two operations: a PUSH that inserts an element at the end of the array, and a POP that removes and returns the last element. Such an array can be implemented using a standard table doubling [11] technique, whereby the underlying table is rebuilt (in parallel) whenever it becomes too full or too empty. To PUSH a batch of x elements into an n-element array, check if $n + x$ elements fit in the current array. If so, in parallel simply insert the ith batch element into the $(n + i)$th slot of the array. If not, first resize the array by allocating new space and copying all existing elements in parallel. POPs can be simultaneously supported by breaking the batch into a PUSH phase followed by a POP phase.

To analyze this data structure, the (amortized) work of a size-x batch is $\Theta(x)$, yielding $W(n) = \Theta(n)$ (worst case). The work of any individual batch, however, can be as high as $\Theta(n)$ when a table doubling occurs. More importantly, any batch A that has w_A batch work has batch span $s_A = O(\lg w_A)$. Hence any batch A that performs $w_A \geq P^2$ work has parallelism $w_A/s_A = \Omega(P^2/\lg P)$. We thus conclude that the data-structure span is $s(n) = O(\lg(P^2)) = O(\lg P)$. Plugging these bounds into Theorem 1, we get a total running time of $O(\frac{T_1+n\lg P}{P} + m\lg P + T_\infty)$.

4. THE BATCHER SCHEDULER

This section presents the high-level design of the BATCHER scheduler, a variant of a distributed work-stealing scheduler. We use P to refer to the number of ***workers***, or threads/cores, given to the scheduler. Since BATCHER is a distributed scheduler, there is no centralized scheduler thread and the operation of the scheduler can be described in terms of state-transition rules followed by each of the P workers. First, we describe the internal state that BATCHER maintains in order to implicitly batched data-structure operations and to coordinate between executing the core and batch dags. We then describe how batches are launched and how load-balancing is done using work-stealing.

BATCHER state. The BATCHER scheduler maintains three categories of shared state: (1) collections for tracking the implicitly batched data-structure operations, (2) status flags for synchronizing the scheduler, and (3) deques for each worker tracking execution-dag nodes (see Section 2) and used by work stealing. With the exception of one global flag, most of this state is distributed across workers, with each worker only managing specific updates according to the provided rules that define the scheduler.

To track active data-structure nodes, BATCHER maintains two arrays. When a worker encounters a data-structure node (executes a call to BATCHIFY(op)), instead of accessing the data-structure di-

rectly, an operation record *op* is created and placed in the ***pending array*** and the data-structure node is suspended. BATCHER guarantees that each worker has at most one suspended node / pending operation at any time; therefore this pending array may be maintained as a size-P array, with a dedicated slot for each of the P workers. BATCHER also maintains a ***working set***, which is a densely packed array of all the operation records being processed as part of the currently executing batch.

To synchronize batch executions, BATCHER maintains a single global ***active-batch*** flag. In addition, each worker p has a local ***work-status*** flag (denoted $Status[p]$), which describes the status of p's current data-structure node. BATCHER guarantees that at any instant, each worker has at most one data-structure node u that it is trying to execute. For concreteness, think in terms of the following four states for worker status $Status[p]$:

- `pending`, if p has an operation record *op* for a suspended data structure node u in the pending array.
- `executing`, if p has an operation record *op* for a suspended data structure node u in the working set, i.e., a batch containing u is currently executing.
- `done`, if the batch A containing u has completed its computation, but p has not yet resumed the suspended node u.
- `free`, if p has no suspended data-structure node.

If $Status[p]$ is `pending`, `executing`, or `done`, we say p is ***trapped*** on operation u. Otherwise, we say p is ***free***.

Finally, BATCHER maintains two deques of ready nodes on each worker: a ***core deque*** for ready nodes from the core dag, and a ***batch deque*** for ready nodes from a batch dag. In particular, the deques in BATCHER obey the following invariant:

INVARIANT 3. *Ready nodes belonging to the core dag G are always placed on some worker's core deque, whereas ready nodes that belong to some batch A's batch dag G_A are always placed on some worker's batch deque.*

Associated with these deques, each worker p also has an ***assigned node*** — the node that p is currently executing. At any instant, the assigned node of p may conceptually be associated with either the core deque or the batch deque, depending on the type of node being executed by that worker. Some workers may be executing core nodes while others are executing batch nodes.

Background: traditional work stealing. In a traditional work-stealing scheduler [6], each of P workers maintains a core deque of ready nodes, and at any time, a worker p has at most one assigned node u that the worker is currently executing. When u finishes, it may enable at most 2 nodes. If 1 or 2 node(s) are enabled, p assigns one to itself and places the other (if any) at the bottom of its deque. If none are enabled, then p removes the node at the bottom of its deque and assigns it to itself. If p's deque is empty, then p becomes a ***thief***, randomly picks a ***victim*** worker and steals from the top of the victim's deque. If the victim's deque is not empty, then the steal attempt ***succeeds***, otherwise it ***fails***.

BATCHER algorithm. BATCHER uses a variant of work stealing, with some augmentations to support implicit batching. Free workers and trapped workers behave quite differently. Initially all workers are free, and all ready nodes belong to the core dag, and BATCHER behaves similarly to traditional work stealing. As data-structure nodes are encountered, however, the situation changes. The scheduling rules are outlined in Figure 3 and described below.

Free workers behave closest to traditional work stealing. A free worker is allowed to execute any node (core or batch), but it only steals if both of its deques are empty. Specifically, if either deque is nonempty, the worker executes a node off the nonempty deque, and

When p is free and both deques are empty:
    steal from random victim, using alternating-steal policy

When a data-structure node u is assigned to (free) worker p:
    insert operation record into $pending[p]$
    $Status[p] =$ `pending`
    suspend u
    // p is now trapped

When p is trapped and its batch deque is empty:
    if $Status[p] =$ `done`
      then $Status[p] =$ `free`
        resume executing the core deque from
                suspended data-structure node u
        // p is now free
      else if global batch flag = 0 and
          compare-and-swap(global batch flag, 0, 1)
        then run LAUNCHBATCH
        else steal from random victim's batch deque

Figure 3: Scheduler-state transition rules invoked by workers with empty deques. When the appropriate deque is not empty, the worker removes the bottom node from the deque and executes it.

any newly enabled nodes are placed on the same deque. BATCHER thus maintains the following invariant:

INVARIANT 4. *Workers that have `free` status in BATCHER can have at most one of their deques non-empty, i.e., they have nodes either on the batch deque or core deque, but not both.*

If both deques are empty, however, the free worker performs a steal attempt according to an ***alternating-steal policy***: each worker's kth steal attempt (successful or not) is from a random victim's core deque if k is even, and from a random victim's batch deque if k is odd. The alternating-steal policy is important to achieve the performance bound in Section 5.

When a (free) worker p executes a data-structure node u, p first inserts the corresonding operation record *op* in its dedicated slot in the pending array, and then it changes its own status to `pending`. At this point, the p becomes trapped on u, and according to Invariant 4, it has an initially empty batch deque.

Unlike free workers, which are allowed to execute both core and batch work, trapped workers are only allowed to execute nodes from a batch deque. If a trapped worker p has a nonempty batch deque, it simply selects a node off the batch deque as in traditional work stealing. If it has an empty batch deque, however, it performs the following step. First, it checks whether its data-structure node u has finished, i.e., if $Status[p] =$ `done`. If so, it changes its own status to `free` and resumes from the suspended data-structure node on the core deque. Otherwise, it checks the global batch status flag and tries to set it using an atomic operation if no batch is executing. If successful in setting the flag, p "launches" a batch. If it is unsuccessful (someone else successfully set the flag and launched a batch) or if a batch is already executing (status flag was already 1) it simply tries to steal from a random victim's deque. BATCHER guarantees that if no batch is executing, then all workers have status either `pending`, `done` or `free`; therefore, only pending workers can succeed in launching a new batch.

```
LAUNCHBATCH()
1  parallel_for i = 1 to P
       do if Status[i] = pending then Status[i] = executing
2  compact all executing op records, moving them from
                                   pending array to working set
         // using parallel prefix sums subroutine
3  execute BOP (the actual parallel batch) on records in working set
4  parallel_for i = 1 to P
       do if Status[i] = executing then Status[i] = done
           remove done op records from the working set.
5  reset global batch-status flag to 0
```

Figure 4: Pseudocode for launching a batch. This method executes as an ordinary task in a dynamic multithreaded computation, i.e., it may run using any number of workers between 1 to P workers, depending on how work-stealing occurs.

Launching a batch. Launching a batch corresponds to injecting the parallel task LAUNCHBATCH (see Figure 4), i.e., by inserting the root of the subdag induced by this code on a worker's batch deque. This process has five steps. First, the pending array is processed in parallel, changing the status of all `pending` workers to `executing`, thereby acknowledging the operation record. Second, the `executing` records are packed together in the working-set array, which can be performed in parallel using a parallel prefix sums computation. Third, the actual batched operation (BOP) is executed on the records in working set. Fourth, the pending array is again processed in parallel, changing the status of all `executing` workers to `done`. Finally, the batch-status flag is reset to 0. In practice, several of these steps can be merged, but we are not concerned about these low-level optimizations in this paper.

As mentioned above, launching a batch incurs some overhead, such as updating status fields and compacting the pending array into working-set, beyond the execution of the batched operation BOP itself. We refer to this overhead as the ***batch-setup overhead***. Note that this set-up procedure is itself a dynamic multithreaded program with $\Theta(P)$ work and $\Theta(\lg P)$ span, primarily due to the cost of the **parallel_for** and parallel prefix sums computations over P elements. This set-up work is performed in exactly the same way as the batched operation BOP is performed — that is, the nodes of this procedure are placed on batch deques and are executed in parallel (via work-stealing) by workers working on these deques. Note, however, that for the purposes of the dag metrics, the overhead is *not* counted as part of the batch work, batch span, or data-structure span defined in Section 2; this omission is by design since the overhead is a function of the scheduler, not the input program. This fact is one of the challenges in proving that BATCHER has a good running time, and it is exacerbated by the fact that the overhead is as high for a batch containing 1 operation as it is for a batch containing P operations. Nevertheless, we shall show (Section 5) that BATCHER is provably efficient.

Not trapped long. The following lemma shows that a worker is not trapped for very long by a particular operation (at most 2 batches).

LEMMA 2. *Once the operation record for a data-structure node u is put into the pending array, at most two batches execute before the node completes.*

PROOF. Consider an operation u whose status changes at time t to `pending`. Any batch finishing before time t does not delay u. Any batch A launched after time t observes u in the pending array, and incorporates it in A and completes it; this accounts for one batch execution. According to Invariant 1, there can only be one batch that is launched before t and finishes after t, which accounts for the second batch. □

Correctness of state changes. Each worker is responsible for changing its own state from `done` to `free` and `free` to `pending`. There is thus no risk of any races on these state changes. The changes from `pending` to `executing` and `executing` to `done` may be performed by an arbitrary worker, but these changes occur as part of the parallel computation LAUNCHBATCH. Since LAUNCHBATCH is itself race free and only one LAUNCHBATCH occurs at a time (protected by the global batch-status flag), these transitions are also safe.

5. ANALYSIS OF BATCHER

We now analyze the performance of BATCHER. We first provide some definitions and the statement of the completion time bounds. Then we use a potential function argument to prove these bounds.

Definitions and theorem statements. We will analyze the running time using the computation dag G and the set of batch dags that represent batches generated due to implicit batching performed by BATCHER. We will analyze the running time using an arbitrary parameter τ, which will be later related to the data-structure span $s(n)$. We define a few different types of batches. A batch A is ***τ-wide*** if its batch work is more than $P\tau$. A batch is ***τ-long*** if its batch span is more than τ. These definitions only count the work and span within the batched operations themselves, not the batch-setup overhead due to BATCHER. Since τ is implied, we often drop it and call batches wide or long. Finally, a batch is ***popular*** if it processes more than $P/4$ operation records; that is, it contains more than $P/4$ data structure nodes. A batch is ***big*** if it is either long, wide or popular, or if it occurs immediately before or after a long, wide or popular batch. All other batches are ***small***.

The above definitions are with respect to individual batches that arise during an execution. We next define a property of the data structure itself, analogous to data-structure work (Section 2).

DEFINITION 1. *Consider any sequence of parallel batched operations and a real value τ. The **τ-trimmed span of the sequence** of batches is the sum of the spans of the* long *batches in the sequence. The **τ-trimmed span of a data structure**, denoted by $S_\tau(n)$, is the worst-case τ-trimmed span for n data-structure nodes grouped arbitrarily into batches.*

We now state our main theorem, a bound on the total running time of a BATCHER computation, which is proven at the end of this section. The restriction that $\tau \geq \lg P$ arises from binary forking.

THEOREM 3. *Consider a computation with T_1 core work, T_∞ core span, and n data-structure nodes with at most m falling along any path through the dag. For any $\tau \geq \lg P$, let $S_\tau(n)$ and $W(n)$ denote the worst-case τ-trimmed span and total work of the data structure, respectively. Then BATCHER executes the program in $O\left(\frac{T_1+W(n)+n\tau}{P} + T_\infty + S_\tau(n) + m\tau\right)$ expected time on P processors.*

This theorem holds for any $\tau \geq \lg P$; however, it does not provide intuition about which τ is best. There is a tradeoff: increasing τ increases $n\tau$ and $m\tau$, but decreasing τ increases $S_\tau(n)$ since more batches become long. As we shall see at the end of this section, setting $\tau = s(n)$, the data-structure span (defined in Section 2), yields Theorem 1 as a corollary since other terms dominate $S_\tau(n)$.

Intuition behind the analysis. As with previous work-stealing analyses, our analysis separately bounds the total number of pro-

cessor steps devoted to various activities; in our case, these activities are core work, data-structure work, stealing (and failed steal attempts), and the batch-setup overhead. We then divide this total by P, since each processor performs one processor step per timestep, to get the completion time.

It is relatively straightforward to see that the number of processor steps devoted to core work is T_1 and the number of time steps devoted to data structure work is $W(n)$. The difficulty is in bounding the number of steal attempts and the batch set up overhead. To bound the number of steal attempts, we adopt a *potential function* argument similar to Arora et al.'s work-stealing analysis [2], henceforth referred to as ABP. In the ABP analysis, each ready node is assigned a potential that decreases geometrically with its distance from the start of the dag. For traditional work stealing, one can prove that most of the potential is in the ready nodes at the top of the deques, as these are the ones that occur earliest in the dag. Therefore, $\Theta(P)$ random steal attempts suffice to process all of these nodes on top of the deques, causing the potential to decrease significantly. Therefore, one can prove that $O(PT_\infty)$ steal attempts are sufficient to reduce the potential to 0 in expectation.

The ABP analysis does not directly apply to bounding the number of steal attempts by BATCHER for the following reason. When a data structure node u becomes ready and is assigned to worker p, p places the corresponding operation record in the pending array and u remains assigned (control flow does not go past u) until results from u are available. But u may contain most of the potential of the entire computation (particularly if p's deque is empty; in this case u has all of p's potential). Since u cannot be stolen, steals are no longer effective in reducing the potential of the computation until the batch containing u completes. To cope with this difficulty, we apply different progress arguments to big batches and small batches.

Bounding steal attempts during big batches: For big batches, we apply the ABP potential function to each batch's computation dag. Nodes in a batch dag are never "suspended" in the way data-structure nodes are, so the ABP argument applies nearly directly. We charge this case against the τ-trimmed span or the data-structure work. (As a technical detail, we must also show that P steal attempts overall equate to $\Omega(P)$ steal attempts from batch deques in order to complete the argument.)

Bounding other steal attempts: Unfortunately, small batches do not contribute to the τ-trimmed span, so the above approach does not apply.[6] Instead, we apply extra machinery to bound these steal attempts. The intuition is that if many steal attempts actually occur during a small batch, then the batch should complete quickly (i.e., within $O(\tau)$ timesteps). On the other hand, if few steal attempts occur then the workers are being productive anyway, since they are doing useful work (either core work or data-structure work) instead of stealing. To apply this intuition more formally within the ABP framework, we augment each data-structure node to comprise a chain of τ "dummy nodes," which captures these cases by appropriate potential decreases in the augmented dag.

Setup: dag augmentation and potential function. We create an augmented computation dag, the τ−***execution dag*** $G(\tau)$, by adding a length $\Theta(\tau)$ chain of ***dummy nodes*** before each data-structure node in the computation dag. The work of this dag is $\mathcal{W}_{G(\tau)} = T_1 + O(n\tau)$ and span is $\mathcal{S}_{G(\tau)} = T_\infty + O(m\tau)$.

For the purpose of the analysis, we suppose the scheduler executes the augmented dag instead of the original dag. The scheduler operates with one corresponding difference: when a worker encounters a data-structure node, this node remains assigned to the worker, but $\Theta(\tau)$ nodes of the dummy-node chain are placed at the bottom of its core deque. If a worker p steals from another worker p''s core deque and a dummy node is on the top of that deque, then p steals and immediately processes the dummy node. This steal is considered a successful steal attempt. When a worker returns from a batch, all the dummy nodes on the bottom of its deque disappear. Note that dummy nodes are only for accounting. Operationally, this runtime system is identical to the one described in Section 4, except the analysis now just counts some unsuccessful steals as successful steals. More precisely, whenever a dummy node is stolen from a victim's deque, the corresponding steal in the real execution is unsuccessful because the victim's deque was empty.

We now define the potentials using this augmented dag. Each node in G has ***depth*** $d(u)$ and ***weight*** $w(u) = \mathcal{S}_G - d(u)$. Similarly, for a node u in the batch dag G_A, $d(u)$ is its depth in that dag, and its weight is $w(u) = s_A - d(u)$. The weights are always positive.

DEFINITION 2. *The* ***potential*** Φ_u *of a node* u *is* $3^{2w(u)-1}$ *if* u *is assigned, and* $3^{2w(u)}$ *if* u *is ready.*

The ***core potential*** of the computation is the sum of potentials of all (ready or assigned) nodes $u \in G$. The ***batch potential*** is the sum of the potentials of all $u \in G_A$ where A is the currently active batch (if one exists). The following structural lemmas follow in a straightforward manner from the arguments used throughout the ABP paper [2], so we state them without proof here. [7]

LEMMA 4. *The initial core potential is* 3^{S_G} *and it never increases during the computation.* □

LEMMA 5. *Let* $\Phi(t)$ *denote the potential of the core dag at time* t. *If no trapped worker's deque is empty, then after* $2P$ *subsequent steal attemps from core deques the core potential is at most* $\Phi(t)/4$ *with probability at least* $1/4$. □

LEMMA 6. *Suppose a computation (core or batch) has span* S, *and that every "round" decreases its potential by a constant factor with at least a constant probability. Then the computation completes after* $O(S)$ *rounds in expectation, and the total number of rounds is* $O(S + \lg(1/\varepsilon))$ *with probability at least* $1-\varepsilon$. □

The following two lemmas extend Lemmas 4 and 5 to batch potentials. The proofs of these lemmas can also be derived from ABP proofs in a similar manner.

LEMMA 7. *The batch potential* Φ_A *increases from* 0 *to* 3^{2s_A} *when* A *becomes ready, and never increases thereafter.* □

LEMMA 8. *Let* $\Phi_A(t)$ *be the potential of batch* A *at time* t. *After* $2P$ *subsequent steal attempts from batch deques, the potential of* A *is at most* $\Phi_A(t)/4$ *with probability at least* $1/4$. □

Since different arguments are required for big and small batches, we partition steal attempts into three categories. A ***big-batch steal attempt*** is any steal attempt that occurs on a timestep during which a big batch is executing. A ***trapped steal attempt*** is a steal attempt made by a trapped worker (a worker whose status is not `free`) on a timestep when no big batch is active. A ***free steal attempt*** is a steal attempt by a free worker (a worker whose status is `free`) on

[6] Adding even P steal attempts for each of potentially n small batches would result in $\Omega(nP)$ steal attempts or $\Omega(n)$ running time, i.e., no parallelism.

[7] ABP does not explicitly capture these three lemmas as claims in their paper — some of their proof is captured by "Lemma 8" and "Theorem 9" of [2], but the rest falls to interproof dicussion within the paper .

a timestep when no big batch is active. We can now bound the different types of steal attempts and the batch-setup overhead.

Big-batch steal attempts. The big-batch steal attempts are bounded by the following lemma. The proof of this lemma is the most straightforward of the three cases.

LEMMA 9. *The expected number of big-batch steal attempts is* $O(n\tau + PS_\tau(n) + W(n))$.

PROOF. We first prove that if L is the set of big batches, the expected number of big batch steal attempts is $O(P\sum_{A\in L} s_A)$.

Consider a particular big batch A. When the first round starts, the potential of the batch is 3^{2s_A} (Lemma 7). Divide the steal attempts that occur while the batch is executing into rounds of $4P$ steal attempts, except for the last round, which may have fewer. While A is executing, at least half the steal attempts are from batch deques, since all the trapped steals are from batch deques, and half the free steals are from batch deques by the alternating steal policy. Therefore, in every round, at least $2P$ steal attempts are from batch deques. Applying Lemma 8, the potential of the batch decreases by a constant factor with probability $1/4$ during each round. Therefore, applying Lemma 6, we can conclude that there are expected $O(s_A)$ rounds while A is active.

We use linearity of expectation to add over all big batches. We first add over long, wide and popular batches. The total span of long batches is $S_\tau(n)$ by definition. There are at most $W(n)/P\tau$ wide batches, and at most n/P popular batches. If they are not also long, they have span less than τ. We triple the number to account for batches before and after long, wide or popular batches. Therefore, we can see that there are the number of rounds during big batches ar $O(S_\tau(n) + n\tau/P + W(n)/P)$. Since each round has at most $4P$ steal attempts, we get the desired bound. □

Free steal attempts. Here, each "round" consists of $4P$ consecutive free steal attempts (during which no big batch is active). Recall that when a worker becomes trapped, it places $\Theta(\tau)$ dummy nodes on the bottom of its core deque. We say that a round is ***bad*** if, at the beginning of the round, some trapped worker's core deque is empty (does not have any core nodes or dummy nodes). Otherwise, a round is ***good***. Note that bad rounds only occur while some batch is executing; otherwise no worker is trapped. We bound good and bad rounds separately.

Good rounds do not have the problem of too much potential being concentrated in a suspended data-structure node of a trapped worker. During a good round, there is more potential in the dummy nodes than the suspended data-structure node itself, and steal attempts reduce potential.

LEMMA 10. *The number of good rounds is* $O(\mathcal{S}_G)$ *in expectation and* $O(\mathcal{S}_G + \lg(1/\varepsilon))$ *with probability at least* $1-\varepsilon$. *Therefore, the number of free steal attempts in good rounds is* $O(P\mathcal{S}_G)$ *in expectation and* $O(P\mathcal{S}_G + P\lg(1/\varepsilon))$ *with probability at least* $1-\varepsilon$.

PROOF. During a good round, there are $4P$ total steal free steal attempts, and thus by the alternating steal policy, half of these ($2P$) are from core deques. Since no trapped worker's deque is empty when the round begins, we can apply Lemma 5 to conclude that each round decreases the core potential by a constant factor with a constant probability; being interrupted by a big batch only decreases the potential further. We can then apply Lemma 6 to conclude that there are $O(\mathcal{S}_G)$ rounds show the requisite bound; multiplying by P gives the bound on the number of free steal attempts during good rounds. □

We can now bound the number of bad rounds using the following intuition. The number of bad rounds is small since small batches have small spans, chances are most small batches finish before any trapped worker runs out of dummy nodes.

LEMMA 11. *The total number of free steal attempts during bad rounds is* $O(n\tau)$ *in expectation.*

PROOF. A worker p places $\Theta(\tau) = b\tau$ dummy nodes, for constant b, on its core deque when it becomes trapped. There is a bad round if its core deque is stolen from at least $b\tau$ times before p becomes free again. There are two cases:

Case 1: worker p is trapped for $k\tau$ rounds, for some constant k; applying a Chernoff bounds, during $k\tau$ rounds, each core deque is stolen from $< k_1\tau + k_2\lg P$ times with probability $> (1-1/P^2)$ for appropriate settings of constants k_1 and k_2. If $\tau \geq \lg P$ and $b = k_1 + k_2$, then p's deque runs out of dummy nodes with probability $< 1/P^2$. Since there can be at most $k\tau$ bad rounds, we get the expected number of bad rounds $O(\tau/P)$.

Case 2: worker p is trapped for more than $k\tau$ rounds, for constant k. From Lemma 2, we know that p is trapped for at most 2 batches, say A_1 and A_2. Therefore, at least one of A_1 and A_2, say A_i, must be active for more than $k\tau/2$ rounds. We first bound the number of rounds during which A_i can be active, with high probability. If A_i is active throughout a round, then there are at least $2P$ steal attempts from batch deques during the round r (since half the free steal attempts hit batch deques) and Lemma 8 applies. If a batch starts or ends during r, its potential decreases by a constant factor trivially. We can then apply Lemma 6 to show that with probability at least $1-\varepsilon$ the batch A_i is active for $O(s_{A_i} + \lg(1/\varepsilon)) = O(\tau + \lg(1/\varepsilon))$ rounds, since A_i is not long. (p is waiting for the small batch A_2; therefore, the preceding batch A_1 is also not long.) We know that $O(\tau + \lg 1/\varepsilon) < k_1\tau + k_2\lg 1/\varepsilon$ for some constants k_1 and k_2; we set $\varepsilon = 1/P^2$ and $k/2 = k_1 + 2k_2$. The probability that A_i is active for $k\tau/2$ rounds is at most $1/P^2$. There can be at most $P\tau$ bad rounds for A_i, since each round takes at least one timestep, and a small batch has at most $P\tau$ work. Therefore, the expected number is at most $O(\tau/P)$.

Adding over the n batches that can trap a worker, and over P workers, gives us $O(n\tau)$ in total. □

COROLLARY 12. *Ignoring the batch-setup overhead, the expected number of steps taken by free processors when no big batch is active is* $O(T_1 + W(n) + n\tau + P\mathcal{S}_G)$.

PROOF. A free worker is either working (at most $T_1 + W(n)$ steps) or stealing (bounded by Lemmas 10 and 11). □

Trapped steal attempts and batch-setup overhead. We next analyze the steal attempts by trapped workers during small batches. The key idea is as follows. Recall that a worker is trapped by a batch A only if it has a pending data structure node whose operation record is being processed by A or will be processed by the succeeding batch A' (see Lemma 2). If more than $P/2$ workers are trapped on a A, then either A or A' must be popular, in which case A is called big. Therefore, at most $P/2$ workers a be trapped by a small batch.

LEMMA 13. *The expected number of processor steps taken due to batch-setup overhead and trapped steal attempts is* $O(T_1 + W(n) + n\tau + P\mathcal{S}_G + PS_\tau(n))$.

PROOF. The batch-setup overhead is $O(P)$ per batch. After it launches, each batch executes for at least 1 timestep and only one batch executes at a time. For big batches, during this one timestep,

the workers perform P steps of either work (bounded by $T_1 + W(n)$) or big batch steals (bounded by Lemma 9). We can amortize the batch-setup overhead against these P steps. For small batches, at least $P/2$ processors are free and again they perform either work or free steals (bounded by Corollary 12), and we can amortize the batch-setup overhead against this quantity. Adding these gives us the bound on batch-setup overhead.

Even if we pessimistically assume that trapped workers do nothing but steal during small batches, since at least half the workers are free, we can amortize these steals against the steps taken by free workers which either work or steal or perform batch setup steps. □

Overall running time. We can now bound the overal running time. We combine the bounds from Lemmas 9, 10 and 11,and substitute $\mathcal{S}_G = T_\infty + m\tau$ and divide by P (since there are P workers performing these steps) to prove Theorem 3.

PROOF OF THEOREM 3. From Lemmas 9, 10, 11 and 13, we know that the expected number of big-batch steal attempts is $O(n\tau + PS_\tau(n) + W(n))$, free steal attempts is $O(PT_\infty + Pm\tau + n\tau)$, and trapped steal attempts is $O(T_1 + W(n) + n\tau + P\mathcal{S}_G + PS_\tau(n))$. The total batch-setup overhead is $O(T_1 + W(n) + n\tau + P\mathcal{S}_G + PS_\tau(n))$. Adding the total work and dividing by P gives the result. □

We can now set an appropriate value for τ to get the bound on BATCHER performance. This corollary is equivalent to Theorem 1.

COROLLARY 14. *BATCHER executes the program described in Theorem 3 in expected time* $O\left(\frac{T_1+W(n)+ns(n)}{P} + ms(n) + T_\infty\right)$.

PROOF. We get this bound by setting τ to be equal to the data structure span $s(n)$. Recall that long batches are defined as batches with batch span longer than τ, and τ-trimmed span $S_\tau(n)$ is defined as the sum of the spans of all long batches. Recall, also, from the definition of the data-structure span $s(n)$ is defined as follows: For any sequence of batches comprising a total of n data structure nodes, such that no batch contains more than P data structure nodes, $s(n)$ is the worst case span of any batch individual A that also has parallelism limited by $w_A/s_A = O(P)$.

Since the program has a total of n data-structure nodes, and BATCHER only generates batches with at most P data structure nodes, the only batches with $s_A > s(n)$ are those where $w_A/s_A = \Omega(P)$. Now, say L is the set of long batches. For all $A \in L$, we have $w_A = \Omega(Ps_A)$, since all other batches have span smaller than $s(n)$, hence also smaller than τ, since $s(n) = \tau$. That is, the long batches are all batches with large parallelism. Therefore, $W(n) \geq \sum_{A\in L} w_A = \sum_{A\in L} \Omega(Ps_A)$. Since $S_\tau(n) = \sum_{A\in L} s_A$, we conclude that $W(n) = \Omega(PS_\tau(n))$, or $W(n)/P = \Omega(S_\tau(n))$. The bound follows from Theorem 3 as $W(n)/P$ dominates $S_\tau(n)$. □

6. RELATED WORK

BATCHER most closely resembles various software combining techniques, designed primarily to reduce concurrency overhead in concurrent data structures. In some combining techniques [15, 21, 31], each processor inserts a request in a shared queue and a single processor sequentially executes all outstanding requests later. These works provide empirical efficiency, but we are not aware of any theory bounding the running time of an algorithm using these combiners. BATCHER improves upon these techniques by operating on the "request queue" in parallel and by providing runtime theory. Other software-combining techniques include (static) combining trees [20] or (dynamic) combining funnels [38] which apply directly to data structures with combinable operations like lock objects, counters, or stacks. These do have a provably $O(\lg P)$ overhead, but do not address more general structures

Several related mechanisms designed for dynamic multithreading have a grounding in theory. Reducers [17] in Cilk can be used to eliminate contention on some shared global variables, but are not designed to replace a generic concurrent data structure, since they create local views on each processor rather than maintain a single global view. It is also unclear how to analyze reducers that include highly variable amortized costs. Helper locks [1] provide a mechanism that allows blocked workers to help complete the critical section that is blocking them and is not specifically designed for data structures. Conceptually, one can use this mechanism to execute batches; however, directly applying the analysis of [1] leads to worse completion time bounds compared to using BATCHER.

Concurrent data structures themselves are widely studied [24]. Most theoretical work on concurrent data structures focuses on correctness and forward-progress guarantees like linearizability [25], lock freedom [23], or wait freedom [22]. While wait-free structures often include a worst-case performance bound, the bound may not be satisfying when applied in the context of an enclosing algorithm. For example, a universal wait-free construction of [9] has a worst-case cost that includes a factor of P, the number of processors, which implies serializing all data structure operations. Experimental studies of various concurrent B-tree data structures alone spans over 30 years of research [3, 4, 27, 7]. These results typically fall short of bounds on running time, with [4] being one exception assuming uniformly random accesses.

Several batched search trees exist, including 2-3 trees [33], weight-balanced B-trees [14], and red-black trees [16]. Moreover, some of these data structures [14, 16] exhibit good practical performance.

7. EXPERIMENTAL EVALUATION

We implemented a prototype of BATCHER within the Cilk-5 [19] runtime system. Our preliminary evaluation, presented here, is based around a skip-list data structure. Note that the primary contribution of this paper is the theory; these experiments are meant to be only proof of concept, not a comprehensive study. Nevertheless, the results indicate that implicit batching is a promising direction, at least for expensive data structures and large-enough batches. For the particular experiment here, BATCHER's performance on 1 processor is comparable to that of a sequential skip list, and hence the overhead is not prohibitive. In addition, BATCHER provides speedup when running on multiple processors.

We conducted experiments on a 2-socket machine with 8 cores per socket running Ubuntu 12.04. The processor was Intel Xeon E5-2687W. The machine has 64GB of RAM and 20MB of L3 cache per socket. For our experiments, we pinned the threads to a single socket on this machine.

BATCHER and skip-list implementations. We implemented the BATCHER scheduler by modifying the Cilk-5 runtime system, essentially as described in Section 4. The main difference between the theoretical and the practical design is within the LAUNCHBATCH operation (Figure 4). Because we are running on only 8 cores, we used a sequential implementation for the status changes status changes (lines 1 and 4) and the compaction (line 2).

Our batch insert (BOP) into the skip list has three steps. 1) build a new skip from a set of records, 2) perform searches for these nodes in the main skip list, and 3) splice the new list into the main list. Since the new list is small (batch size), we perform steps 1 and 3 sequentially, whereas the searches into the large main list in step 2 are performed in parallel. The core program is simply a

parallel-for loop that inserts into the skip list in each iteration (e.g., as in Figure 1). Note that this is a bad case for BATCHER since all of the work happens within the data structure, and hence the overheads are tested.

Experimental scaling. In all our experiments, we first initialize the skip list with an initial size. We then timed the insertion of 100,000 additional elements into this skip list. To simulate bigger batches without the NUMA effects of going to multiple sockets, each BATCHIFY call creates 100 insertion records. We compared BATCHER with a sequential implementation where all 100,000 elements are inserted sequentially (without concurrency control).

Figure 5 shows the throughput of BATCHER and a sequential skip list with `initialSize` 20,000, 100,000, 1 million, 10 million and 100 million (e.g. `BAT20000` shows BATCHER's with initial size 20,000). For initial size 20,000 and 100,000, `SEQ` performs better than `BAT` on a single processor. This is because inserts into small skip lists are so cheap that BATCHER's overheads begin to dominate. However, even on these small skip lists, BATCHER provides speedup, and outperforms the sequential skip list on multiple processors. For larger skip lists, the inserts get expensive enough that they dominate BATCHER's overhead, and BATCHER performs comparably with the sequential list even on 1 processor.

More interestingly, BATCHER's speedup increases as the skip list gets larger. At size 100 million, BATCHER provides a speedup of about 3× on 6 processors, and 3.33× on 8 workers.

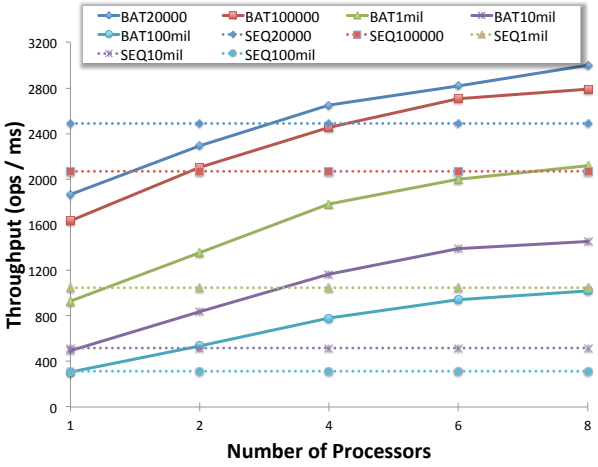

Figure 5: Throughput of BATCHER and sequential skip list insertion for various initial sizes of skip lists (higher is better).

Flat combining. We view flat combining [21] as a special case of implicit batching where batches execute sequentially. They show that flat combining significantly outperforms a good concurrent skip list, at least for certain workloads, validating the idea of implicit batching. On our experiments, flat combining and BATCHER perform similarly 1 processor. However, the performance of flat combining decreases with increasing cores (their experiments also show this). In contrast, the prototype BATCHER implementation shows speedup.

8. CONCLUSIONS AND FUTURE WORK

BATCHER scheduler is provably efficient, and preliminary experiments indicate that it could provide speedup in practice, especially when data structure operations are expensive enough to amortize the overheads. There are several open questions remaining. Is it possible to remove or reduce the $O(\lg P)$ overhead by using a more clever communication mechanism? What data structures are easily and efficiently expressible by this batch mechanism? Does BATCHER improve the performance of real parallel programs? Finally, although BATCHER is designed with work-stealing in mind, note that it may also be applicable to pthreaded programs that use data structures. A pthreaded program could run as normal, with data-structure calls replaced by BATCHER calls allowing work-stealing to operate over the data structure batches while static pthreading operates over the main program.

Acknowledgements

This research was supported by National Science Foundation grants CCF-1340571, CCF-1150036, CCF-1218017, CCF-1218188, and CCF-1314633.

9. REFERENCES

[1] K. Agrawal, C. E. Leiserson, and J. Sukha. Helper locks for fork-join parallel programming. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 245–256, Jan. 2010.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 119–129, 1998.

[3] R. Bayer and M. Schkolnick. Concurency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[4] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, July 17–20 2005.

[5] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 133–144, June 27–30 2004.

[6] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[7] A. Braginsky and E. Petrank. A lock-free B+tree. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 58–67, 2012.

[8] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, pages 4–21, 1998.

[9] P. Chuong, F. Ellen, and V. Ramachandran. A universal construction for wait-free transaction friendly data structures. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 335–344, 2010.

[10] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 226–237, 2010.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[12] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 722–731. Springer, 1998.

[13] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: an alternative to Fibonacci heaps with

applications to parallel computation. *Communications of the ACM*, 31:1343–1354, 1988.

[14] S. Erb, M. Kobitzsch, and P. Sanders. Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates. In *Proceedings of the Symposium on Experimental Algorithms (SEA)*, 2014. to appear.

[15] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 325–334, 2011.

[16] L. Frias and J. Singler. Parallelization of bulk operations for STL dictionaries. In *Euro-Par Workshops*, volume 4854 of *LNCS*, pages 49–58. Springer, 2007.

[17] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 79–90, Aug. 2009.

[18] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998.

[19] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998.

[20] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 64–75. ACM, 1989.

[21] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, 2010.

[22] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[23] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15:745–770, 1993.

[24] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[25] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[26] Intel Corporation. *Intel Cilk Plus Language Extension Specification, Version 1.1*, 2013. Document 324396-002US. Available from `http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm`.

[27] T. Johnson and D. Shasha. The performance of current B-tree algorithms. *ACM Trans. Database Syst.*, 18(1):51–101, 1993.

[28] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, Oct. 1980.

[29] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of Supercomputing*, pages 24–33, 1991.

[30] OpenMP Architecture Review Board. OpenMP specification and features. `http://openmp.org/wp/`, May 2008.

[31] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*, pages 182–204, 1999.

[32] R. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. In *Int. Conference on Parallel Processing*, pages 14–20, 1985.

[33] W. J. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries in 2-3 trees. In *Proceedings of the International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 597–609, 1983.

[34] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 531–542, 2012.

[35] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[36] P. Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel Distributed Computing*, 49(1):86–97, 1998.

[37] N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, Nov. 1996.

[38] N. Shavit and A. Zemach. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

[39] The Task Parallel Library. `http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`, Oct. 2007.