Parallel Combining: Benefits of Explicit Synchronization

Vitaly Aksenov[1]
ITMO University, Saint-Petersburg, Russia and Inria, Paris, France
aksenov@corp.ifmo.ru

Petr Kuznetsov
LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France
petr.kuznetsov@telecom-paristech.fr

Anatoly Shalyto
ITMO University, Saint-Petersburg, Russia
shalyto@mail.ifmo.ru

Abstract

A *parallel batched* data structure is designed to process synchronized *batches* of operations on the data structure using a parallel program. In this paper, we propose *parallel combining*, a technique that implements a *concurrent* data structure from a parallel batched one. The idea is that we explicitly synchronize concurrent operations into batches: one of the processes becomes a *combiner* which collects concurrent requests and initiates a parallel batched algorithm involving the owners (*clients*) of the collected requests. Intuitively, the cost of synchronizing the concurrent calls can be compensated by running the parallel batched algorithm.

We validate the intuition via two applications. First, we use parallel combining to design a concurrent data structure optimized for *read-dominated* workloads, taking a dynamic graph data structure as an example. Second, we use a novel parallel batched *priority queue* to build a concurrent one. In both cases, we obtain performance gains with respect to the state-of-the-art algorithms.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies, Computing methodologies → Parallel computing methodologies, Theory of computation → Distributed computing models, Theory of computation → Parallel computing models

Keywords and phrases concurrent data structure, parallel batched data structure, combining

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.11

Related Version The full version of the paper is available at [3], `http://arxiv.org/abs/1710.07588`.

1 Introduction

To ensure correctness of concurrent computations, various synchronization techniques are employed. Informally, synchronization is used to handle conflicts on *shared data*, e.g., resolving data races, or *shared resources*, e.g., allocating and deallocating memory. Intuitively, the more sophisticated conflict patterns a concurrent program is subject to – the higher are the incurred synchronization costs.

[1] This work was financially supported by the Government of Russian Federation (Grant 08-08)

© Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto;
licensed under Creative Commons License CC-BY

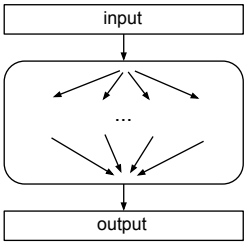

Figure 1 Execution of a parallel program.

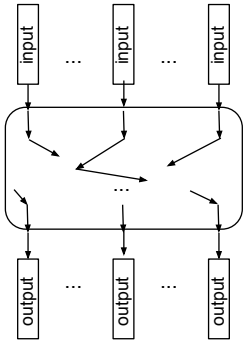

Figure 2 Execution on a concurrent data structure.

Let us consider a concurrent-software class which we call *parallel programs*. Provided an input, a parallel program aims at computing an output that satisfies a *specification*, i.e., an input-output relation (Figure 1). To boost performance, the program distributes the computation across multiple parallel processes. Parallel programs are typically written for two environments: for *static multithreading* and *dynamic multithreading* [11]. In *static multithreading*, each process is given its own program and these programs are written as a composition of *supersteps*. During a superstep, the processes perform conflict-free individual computations and, when done, synchronize to accumulate the results. In *dynamic multithreading*, the program is written using dynamically called *fork-join* mechanisms (or similar ones, e.g., `#pragma omp parallel` in OpenMP [6]). In both settings, synchronization only appears in a specific form: memory allocation/deallocation, and aggregating superstep computations or thread scheduling [17].

General-purpose *concurrent data structures*, such as stacks, binary search trees and priority queues, operate in a much less controlled environment. They are programmed to accept and process asynchronous operation calls, which come from multiple concurrent processes and may interleave arbitrarily. If we treat operation calls as inputs and their responses as outputs, we can say that inputs and outputs are distributed across the processes (Figure 2). It is typically expected that the interleaving operations match the high-level sequential semantics of the data type [24], which is hard to implement efficiently given diverse and complicated data-race patterns often observed in this kind of programs. Therefore, designing efficient and correct concurrent data structures requires a lot of ingenuity from the programmer. In particular, one should strive to provide the "just right" amount of synchronization. *Lock-based* data structures obviate data races by using fine-grained locking ensuring that contested data is accessed in a mutually exclusive way. *Wait-free* and *lock-free* data structures allow data races but mitigate their effects by additional mechanisms, such as *helping* where one process may perform some work on behalf of other processes [23].

As parallel programs are written for a restricted environment with simple synchronization patterns, they are typically easier to design than concurrent data structures. In this paper, we suggest benefiting from this complexity gap by building concurrent data structures *from* parallel programs. We describe a methodology of designing a concurrent data structure from its *parallel batched* counterpart [2]. A parallel batched data structure is a special case of a parallel program that accepts *batches* (sets) of operations on a given sequential data type and executes them in a parallel way. In our approach, we *explicitly* synchronize concurrent operations, assemble them into batches, and apply these batches on an *emulated* parallel batched data structure.

More precisely, concurrent processes share a set of *active requests* using any combining algorithm [31, 21, 15, 16]. One of the processes with an active request becomes a *combiner* and forms a *batch* from the requests in the set. Under the coordination of the combiner, the owners of the collected requests, called *clients*, apply the requests in the batch to the parallel batched data structure. As we show, this technique becomes handy when the overhead of *explicitly synchronizing* operation calls in batches is compensated by the advantages of involving clients into the computation using the parallel batched data structure.

We discuss two applications of parallel combining and experimentally validate performance gains. First, we design concurrent implementations optimized for *read-dominated* workloads given a sequential data structure. Intuitively, updates are performed sequentially and read-only operations are performed by the clients in parallel under the coordination of the combiner. In our performance analysis, we considered a *dynamic graph* data structure [25] that can be accessed for adding and removing edges (updates), as well as for checking the connectivity between pairs of vertices (read-only). Second, we apply parallel combining to *priority queue* that is subject to sequential bottlenecks for minimal-element extractions, while most insertions can be applied concurrently. As a side contribution, we propose a novel parallel batched priority queue, as no existing batched priority queue we are aware of can be efficiently used in our context. Our perfomance analysis shows that implementations based on parallel combining may outperform state-of-the-art algorithms.

Structure. The rest of the paper is organized as follows. In Section 2, we give preliminary definitions. In Section 3, we outline parallel combining technique. In Sections 4 and 5, we present applications of our technique. In Section 6, we report on the outcomes of our performance analysis. In Section 7, we overview the related work. We conclude in Section 8.

2 Background

Data types and data structures. A sequential *data type* is defined via a set of operations, a set of responses, a set of states, an initial state and a set of transitions. Each transition maps a state and an operation to a new state and a response. A *sequential implementation* (or *sequential data structure*) corresponding to a given data type specifies, for each operation, a sequential read-write algorithm, so that the specification of the data type is respected in every sequential execution.

We consider a system of n asynchronous *processes* (processors or threads of computation) that communicate by performing primitive operations on shared *base objects*. The primitive operations can be reads, writes, or *conditional* operations, such as test&set or compare&swap. A *concurrent implementation* (or *concurrent data structure*) of a given data type assigns, for each process and each operation of the data type, a state machine that is triggered whenever the process invokes an operation and specifies the sequence of *steps* (primitives on the base objects) the process needs to perform to complete the operation. We require the implementations to be *linearizable* with respect to the data type, i.e., we require that operations *take effect* instantaneously within their intervals [24].

Batched data structures. A *batched implementation* (or *batched data structure*) of a data type exports one function `apply`. This operation takes a *batch* (set) of data type operations as a parameter and returns responses for these operations that are consistent with some sequential application of the operations to the current state of the data structure, which is updated accordingly. We also consider extensions of the definition where we explicitly define

the "batched" data type via the *set* [30] or *interval* [10] linearizations. Such a data type takes a batch and a state, and returns a new state and a vector of responses.

For example, in the simplest form, a batched implementation may sequentially apply operations from a batch to the sequential data structure. But batched implementations may also use parallelism to accelerate the execution of the batch: we call these *parallel* batched implementations. We consider two types of parallel batched implementations: *static-multithreading* ones and *dynamic-multithreading* ones [11].

Static multithreading. A parallel batched data structure specifies a distinct (sequential) code to each process in PRAM-like models (PRAM [27], Bulk synchronous parallel model [37], Asynchronous PRAM [18], etc.). For example, in this paper, we provide a batched implementation of a priority queue in the Asynchronous PRAM model. The Asynchronous PRAM consists of n sequential processes, each with its own private local memory, communicating through the shared memory. Each process has its own program. Unlike the classical PRAM model, each process executes its instructions independently of the timing of the other processors. Each process performs one of the four types of instructions per tick of its local clock: global read, global write, local operation, or synchronization step. A synchronization step for a set S of processes is a logical point where each processor in S waits for all the processes in S to arrive before continuing its local program.

Dynamic multithreading. Here the parallel batched implementation is written as a sequential read-write algorithm using concurrency keywords specifiying logical parallelism, such as *fork*, *join* and *parallel-for* [11]. An execution of a batch can be presented as a directed acyclic graph (*DAG*) that unfolds dynamically. In the DAG, nodes represent unit-time sequential subcomputations, and edges represent control-flow dependencies between nodes. A node that corresponds to a "fork" has two or more outgoing edges and a node that corresponds to a "join" has two or more incoming edges. The batch is executed using a *scheduler* that chooses which DAG nodes to execute on each process. It can only execute *ready* nodes: not yet executed nodes whose predecessors have all been executed. The most commonly used *work-stealing* scheduler (e.g., [5]) operates as follows. Each process p is provided with a deque for ready nodes. When process p completes node u, it traverses successors of u and collects the ready ones. Then p selects one of the ready successors for execution and adds the remaining ready successors at the bottom of its deque. When p's deque is empty, it becomes a *thief*: it randomly picks a victim processor and steals from the top of the victim's deque.

3 Parallel Combining

In this section, we describe the *parallel combining* technique in a parameterized form: the parameters are specified depending on the application. We then discuss how to use the technique in transforming parallel batched programs into concurrent data structures.

3.1 Combining Data Structure

Our technique relies on a *combining* data structure $\mathbb{C}$ (e.g., the one used in [21]) that maintains a set of requests to a data structure and determines which process is a combiner. If the set of requests is not empty then exactly one process should be a combiner.

Elements stored in $\mathbb{C}$ are of `Request` type consisting of the following fields: 1) the method to be called and its input; 2) the response field; 3) the status of the request with a value in an application-specific `STATUS_SET`; 4) application-specific auxiliary fields. In our applications,

Listing 1 Parallel combining: pseudocode

```
1  Request:
2   method
3   input
4   res
5   status ∈ STATUS_SET
6   ...
7
8  execute(method, input):
9   req ← new Request()
10  req.method ← method
11  req.input ← input
12  req.status ← INITIAL
13  if ℂ.addRequest(req):
14  // combiner
15  A ← ℂ.getRequests()
16  COMBINER_CODE
17  ℂ.release()
18  else:
19  while req.status = INITIAL:
20  nop
21  CLIENT_CODE
22  return
```

`STATUS_SET` contains, by default, values `INITIAL` and `FINISHED`: `INITIAL` meaning that the request is in the initial state, and `FINISHED` meaning that the request is served.

$\mathbb{C}$ supports three operations: 1) `addRequest(r : Request)` inserts request `r` into the set, and the response indicates whether the calling process becomes a combiner or a client; 2) `getRequests()` returns a non-empty set of requests; and 3) `release()` is issued by the combiner to make $\mathbb{C}$ find another process to be a combiner.

In the following, we use any black-box implementation of $\mathbb{C}$ providing this functionality [31, 21, 15, 16].

3.2 Specifying Parameters

To perform an operation, a process executes the following steps (Listing 1): 1) it prepares a request, inserts the request into $\mathbb{C}$ using `addRequest(·)`; 2) if the process becomes the combiner (i.e., `addRequest`(·) returned *true*), it collects requests from $\mathbb{C}$ using `getRequests()`, then it executes algorithm `COMBINER_CODE`, and, finally, it calls `release()` to enable another active process to become a combiner; 3) if the process is a client (i.e., `addRequest`(·) returned *false*), it waits until the status of the request becomes not `INITIAL` and, then, executes algorithm `CLIENT_CODE`.

To use our technique, one should therefore specify `COMBINER_CODE`, `CLIENT_CODE`, and appropriately modify `Request` type and `STATUS_SET`.

Note that *sequential combining* [31, 21, 16, 14] is a special case of parallel combining in which all the work is done by the combiner, and the client code is empty.

3.3 Parallel Batched Algorithms

We discuss how to build a concurrent data structure given a parallel batched one in one of two forms: for static or dynamic multithreading.

In the static multithreading case, each process is provided with a distinct version of `apply` function. We enrich `STATUS_SET` with `STARTED`. In `COMBINER_CODE`, the combiner collects the requests, sets their status to `STARTED`, performs the code of `apply` and waits for the clients to become `FINISHED`. In `CLIENT_CODE` the client waits until its request has `STARTED` status, performs the code of `apply` and sets the status of its request to `FINISHED`.

Suppose that we are given a parallel batched implementation for dynamic multithreading. One can turn it into a concurrent one using parallel combining with the *work-stealing* scheduler. Again, we enrich `STATUS_SET` with `STARTED`. In `COMBINER_CODE`, the combiner collects the requests and sets their status to `STARTED`. Then the combiner creates a working deque, puts there a new node of computational DAG with `apply` function and starts the work-stealing routine on processes-clients. Finally, the combiner waits for the clients to become `FINISHED`. In `CLIENT_CODE`, the client creates a working deque and starts the work-stealing routine.

In Section 5, we illustrate the use of parallel combining and parallel batched programs on the example of a priority queue.

4 Read-Optimized Concurrent Data Structures

Before discussing parallel batched algorithms, let us consider a natural application of parallel combining: data structures optimized for *read-dominated* workloads.

Suppose that we are given a sequential data structure D that supports *read-only* (not modifying the data structure) operations, the remaining operations are called *updates*. We assume a scenario where read-only operations dominate over other updates.

Now we explain how to set parameters of parallel combining for this application. At first, `STATUS_SET` consists of three elements `INITIAL`, `STARTED` and `FINISHED`. `Request` type does not have auxiliary fields.

In `COMBINER_CODE` (Listing 2 Lines 1-19), the combiner iterates through the set of collected requests A: if a request contains an update operation then the combiner executes it and sets its status to `FINISHED`; otherwise, the combiner adds the request to set R. Then the combiner sets the status of requests in R to `STARTED`. After that the combiner checks whether its own request is read-only. If so, it executes the method and sets the status of its request to `FINISHED`. Finally, the combiner waits until the status of the requests in R become `FINISHED`.

In `CLIENT_CODE` (Listing 2 Lines 21-24), the client checks whether its method is read-only. If so, the client executes the method and sets the status of the request to `FINISHED`.

▶ **Theorem 1.** *Algorithm in Listing 2 produces a linearizable concurrent data structure from a sequential one.*

Proof. Any execution of the algorithm can be viewed as a series of non-overlapping combining phases (Listing 2, Lines 2-19). We can group the operations into batches by the combining phase in which they are applied.

Each update operation is linearized at the point when the combiner applies this operation. Note that this is a correct linearization since all operations that are linearized before are already applied: the operations from preceding combining phases were applied during

Listing 2 Parallel combining in application to read-optimized data structures.

```
1  COMBINER_CODE:
2   R ← ∅
3
4   for r ∈ A:
5   if isUpdate(r.method):
6   apply(D, r.method, r.input)
7   r.status ← FINISHED
8   else:
9   R ← R ∪ r
10
11  for r ∈ R:
12  r.status ← STARTED
13  if req.status = STARTED:
14  apply(D, req.method, req.input)
15  req.status ← FINISHED
16
17  for r ∈ R:
18  while r.status = STARTED:
19  nop
20
21 CLIENT_CODE:
22  if not isUpdate(req.method):
23  apply(D, req.method, req.input)
24  req.status ← FINISHED
```

the preceding phases, while the operations from the current combining phase are applied sequentially by the combiner.

Each read-only operation is linearized at the point when the combiner sets the status of the corresponding request to `STARTED`. By the algorithm, a read-only operation observes all update operations that are applied before and during the current combining phase. Thus, the chosen linearization is correct. ◀

To evaluate the approach in practice we implement a concurrent *dynamic graph* data structure by Holm et al. and execute it in read-dominated environments [25] (Section 6.1).

5 Priority Queue

Priority queue is an abstract data type that maintains an ordered multiset and supports two operations:

- `Insert`(v) – inserts value v into the set;
- $v \leftarrow$ `ExtractMin`() – extracts the smallest value from the set.

To the best of our knowledge, no prior parallel implementation of a priority-queue [32, 13, 9, 33] can be efficiently used in our context: their complexity inherently depends on the total number of processes in the system, regardless of the actual batch size. We therefore introduce a novel heap-based parallel batched priority-queue implementation in a form of `COMBINER_CODE` and `CLIENT_CODE` convenient for parallel combining. The concurrent priority queue is then derived from the described below parallel batched one using the approach presented in Section 3.3.

Here we give only the brief overview of our parallel batched algorithm. Please refer to the full version of the paper [3] for a detailed description.

In Section 6.2, we show that the resulting concurrent priority queue is able to outperform manually crafted state-of-the-art implementations.

5.1 Sequential Binary Heap

Our batched priority queue is based on the *sequential binary heap* by Gonnet and Munro [19], one of the simplest and fastest sequential priority queues. We briefly describe this algorithm below.

A binary heap of size m is represented as a complete binary tree with nodes indexed by $1, \ldots, m$. Each node v has at most two children: $2v$ and $2v+1$ (to exist, $2v$ and $2v+1$ should be less than or equal to m). For each node, the *heap property* should be satified: the value stored at the node is less than the values stored at its children.

The heap is represented with *size* m and an array a where $a[v]$ is the value at node v. Operations `ExtractMin` and `Insert` are performed as follows:

- `ExtractMin` records the value $a[1]$ as a response, copies $a[m]$ to $a[1]$, decrements m and performs the *sift down* procedure to restore the heap property. Starting from the root, for each node v on the path, we check whether value $a[v]$ is less than values $a[2v]$ and $a[2v+1]$. If so, then the heap property is satisfied and we stop the operation. Otherwise, we choose the child c, either $2v$ or $2v+1$, with the smallest value, swap values $a[v]$ and $a[c]$, and continue with c.
- `Insert(`x`)` initializes a variable val to x, increments m and traverses the path from the root to a new node m. For each node v on the path, if $val < a[v]$, then the two values are swapped. Then the operation continues with the child of v that lies on the path from v to node m. Reaching node m the operation sets its value to val.

The complexity is $O(\log m)$ steps per operation.

5.2 Setup

The heap is defined by its size m and an array a of Node objects. Node object has two fields: value val and boolean $locked$ (In the full version [3] it has an additional field).

`STATUS_SET` consists of three items: `INITIAL`, `SIFT` and `FINISHED`.

A Request object consists of: a method *method* to be called and its input argument v; a result res field; a *status* field and a node identifier *start*.

5.3 ExtractMin Phase

Combiner: ExtractMin preparation. The combiner withdraws requests A from combining data structure $\mathbb{C}$. It splits A into sets E and I: the set of ExtractMin requests and Insert requests. Then it finds $|E|$ nodes $v_1, \ldots, v_{|E|}$ of heap with the smallest values using the Dijkstra-like algorithm in $O(|E| \cdot \log |E|)$ steps: (i) create a heap of nodes ordered by values, put there the root 1; (ii) at each of the next $|E|$ steps withdraw the node v with the minimal value from the heap; (iii) put two children of v, $2v$ and $2v+1$, to the heap. The $|E|$ withdrawn nodes are the nodes with the $|E|$ minimal values. For each request $E[i]$, the combiner sets $E[i].res$ to $a[v_i].val$, $a[v_i].locked$ to `true`, and $E[i].start$ to v_i.

The combiner proceeds by pairing Insert requests in I with ExtractMin requests in E using the following procedure. Suppose that $\ell = \min(|E|, |I|)$. For each $i \in [1, \ell]$, the combiner sets $a[v_i].val$ to $I[i].v$ and $I[i].status$ to `FINISHED`, i.e., this Insert request becomes

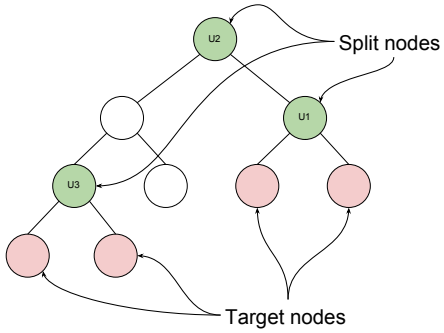

Figure 3 Split and target nodes.

completed. Then, for each $i \in [\ell + 1, |E|]$, the combiner sets $a[v_i].val$ to the value of the last node $a[m]$ and decrements m, as in the sequential algorithm. Finally, the combiner sets the status of all requests in E to `SIFT`.

Clients: ExtractMin phase. Briefly, the clients sift down the values in nodes $v_1, \ldots, v_{|E|}$ in parallel using hand-over-hand locking: the *locked* field of a node is set whenever there is a *sift down* operation working on that node.

A client c waits until the status of its request becomes `SIFT`. c starts sifting down from $req.start$. Suppose that c is currently at node v. c waits until the *locked* fields of the children become `false`. If $a[v].val$, the value of v, is less than the values in its children, then *sift down* is finished: c unsets $a[v].locked$ and sets the status of its request to `FINISHED`. Otherwise, let w be the child with the smallest value. Then c swaps $a[v].val$ and $a[w].val$, sets $a[w].locked$, unsets $a[v].locked$ and continues with node w.

If the request of the combiner is ExtractMin, it also runs the code above as a client. The combiner considers the ExtractMin phase completed when all requests in E have status `FINISHED`.

5.4 Insert Phase

For simplicity, we describe first the sequential algorithm.

At first, the combiner removes all completed requests from I. Then it initializes new nodes $m + 1, \ldots, m + |I|$ which we call *target nodes* and increments m by $|I|$. The nodes for which the subtrees of both children contain at least one target node are called *split nodes*. (See Figure 3 for an example of how target and split nodes can be defined.)

The combiner collects the values of the remaining Insert requests and sorts them: $r_1, \ldots, r_{|I|}$. Then it sets the status of these requests to `FINISHED`.

Now, we introduce InsertSet class: it consists of two sorted lists A and B. The combiner starts the following recursive procedure at the root with InsertSet s: $s.A$ contains $r_1, \ldots, r_{|I|}$ while $s.B$ is empty. Suppose that the procedure is called on node v and InsertSet s. Let min be the minimum out of the first element of $s.A$ and the first element of $s.B$. If v is a target node then the combiner sets $a[v].res$ to min and withdraws m from the corresponding list. Otherwise, the combiner compares $a[v].res$ with min: if $a[v].res$ is smaller, then it does nothing; otherwise, it appends $a[v].res$ to the end of $s.B$ (note that $s.B$ remains sorted because $s.B$ consists only of values that were ancestors in the heap), withdraws min from the corresponding list and sets $a[v].res$ to min.

If v is not the split node the combiner calls the recursive procedure on the child with target nodes in the subtree and with InsertSet s. Otherwise, the combiner calculates inL and inR – the number of target nodes in the left and right subtrees of v. Suppose, for simplicity, that inL is less than inR (the opposite case can be resolved similarly). The combiner splits s into two parts: create InsertSet s_L, move $\min(inL, |s.A|)$ first values from $s.A$ to $s_L.A$ and move $\min(inL - |s_L.A|, |s.B|)$ first values from $s.B$ to $s_L.B$. Finally, it calls the recursive procedure on the left child with InsertSet s_L and on the right child with InsertSet s.

This algorithm works in $O(\log m + c \log c)$ steps (and can be optimized to $O(\log m + c)$ steps), where m is the size of the queue and c is the number of Insert requests to apply. Note that this algorithm is almost non-parallelizable due to its small complexity, and our parallel algorithm is only developed to reduce constant factors.

Now, we construct a parallel algorithm for the Insert phase. We enrich Node object with the IntegerSet field *split*. The combiner sets the *start* field of the first client ($i[1].start$) to the root 1, while *start* fields of other clients to the right children of split nodes (we have exactly $|I| - 1$ split nodes). Then it intializes the *split* field of the root as the IntegerSet s is initialized at the beginning of the sequential algorithm: list A contains values of requests while list B is empty.

Each client waits until the *split* field of the corresponding *start* node is non-null. Then it reads this IntegerSet: the values from this set should be inserted in the subtree. Finally, the client performs the procedure similar to the recursive procedure from the sequential algorithm except for one difference: when it reaches a split node instead of going recursively to left and right children, it splits InsertSet to s_L and s_R of sizes inL and inR, puts s_R into the *split* field of the right child (in order to wake another client) and continues with the left child and s_L.

For further details about the parallel algorithm we refer to the full version of the paper [3].

6 Experiments

We evaluate Java implementations of our data structures on a 4-processor AMD Opteron 6378 2.4 GHz server with 16 threads per processor (yielding 64 threads in total), 512 Gb of RAM, running Ubuntu 14.04.5 with Java 1.8.0_111-b14 and HotSpot JVM 25.111-b14.

6.1 Concurrent Dynamic Graph

To illustrate how parallel combining can be used to construct read-optimized concurrent data structures, we took the sequential dynamic graph implementation by Holm et al. [25]. This data structure supports two update methods: an insertion of an edge and a deletion of an edge; and one read-only method: a connectivity query that tests whether two vertices are connected.

We compare our implementation based on parallel combining (PC) with *flat combining* [21] as a combining data structure against three others: (1) Lock, based on ReentrantLock from java.util.concurrent; (2) RW Lock, based on ReentrantReadWriteLock from java.util.concurrent; and (3) FC, based on *flat combining* [21]. The code is available at `https://github.com/Aksenov239/concurrent-graph`.

We consider workloads parametrized with: 1) the fraction x of connectivity queries (50%, 80% or 100%, as we consider read-dominated workloads); 2) the set of edges E: edges of a single random tree, or edges of ten random trees; 3) the number of processes P (from 1 to 64). We prepopulate the graph on 10^5 vertices with edges from E: we insert each edge with probability $\frac{1}{2}$. Then we start P processes. Each process repeatedly performs operations:

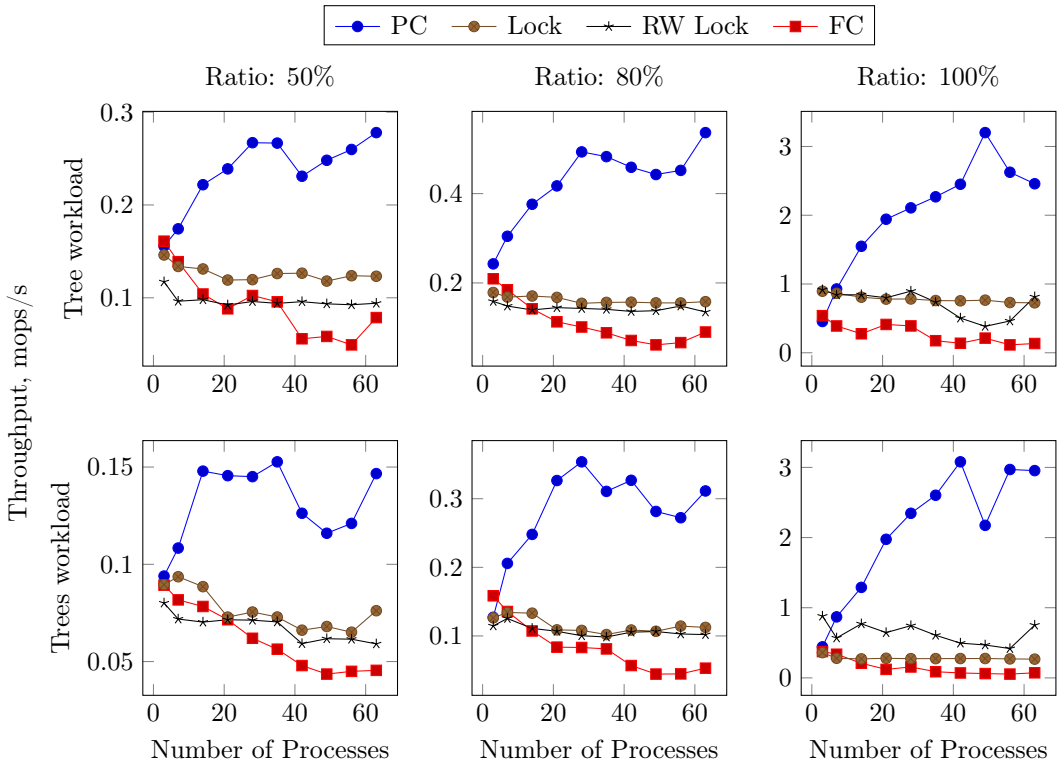

Figure 4 Dynamic graph implementations.

1) with probability x, it calls a connectivity query on two vertices chosen uniformly at random; 2) with probability $1 - \frac{x}{2}$, it inserts an edge chosen uniformly at random from E; 3) with probability $1 - \frac{x}{2}$, it deletes an edge chosen uniformly at random from E.

We denote the workloads with E as a single tree as *Tree* workloads, and other workloads as *Trees* workloads. *Tree* workloads are interesting because they show the degenerate case: the dynamic graph behaves as a dynamic tree. In this case, about 50% of update operations successfully change the spanning fprest, while other update operations only check the existence of the edge and do not modify the graph. *Trees* workloads are interesting because a reasonably small number (approximately, 5-10%) of update operations modify the maintained set of all edges and the underlying complex data structure that maintains a spanning forest (giving in total the squared logarithmic complexity), while other update operations can only modify the set of edges but cannot modify the underlying complex data structure (giving in total the logarithmic complexity).

For each setting and each algorithm, we run the corresponding workload for 10 seconds to warmup HotSpot JVM and then we run the workload five more times for 10 seconds. The average throughput of the last five runs is reported in Figure 4.

From the plots we can infer two general observations: PC exhibits the highest thoughput over all considered implementations and it is the only one whose throughput scales up with the number of the processes. On the 100% workload we expect the throughput curve to be almost linear since all operations are read-only and can run in parallel. The plots almost confirm our expectation: the curve of the throughput is a linear function with coefficient $\frac{1}{2}$ (instead of the ideal coefficient 1). We note that this is almost the best we can achieve: a combiner typically collects operations of only approximately half the number of working

processes. In addition, the induced overhead is still perceptible, since each connectivity query works in just logarithmic time. With the decrease of the fraction of read-only operations we expect that the throughput curve becomes flatter, as plots for the 50% and 80% workloads confirm.

It is also interesting to point out several features of other implementations. At first, FC implementation works slightly worse than Lock and RW Lock. This might be explained as follows. Lock implementations (`ReentrantLock` and `ReentrantReadWriteLock`) behind Lock and RWLock implementations are based on CLH Lock [12] organized as a *queue*: every competing process is appended to the queue and then waits until the previous one releases the lock. Operations on the dynamic graph take significant amount of time, so under high load when the process finishes its operation it appends itself to the queue in the lock without any contention. Indeed, all other processes are likely to be in the queue and, thus, no process can contend. By that the operations by processes are serialized with almost no overhead. In contrast, the combining procedure in FC introduces non-negligible overhead related to gathering the requests and writing them into requests structures.

Second, it is interesting to observe that, against the intuition, RWLock is not so superior with respect to Lock on read-only workloads. As can be seen, when there are update operations in the workload RWLock works even worse than Lock. We relate this to the fact that the overhead hidden inside `ReentrantReadWriteLock` spent on manipulation with read and write requests is bigger than the overhead spent by `ReentrantLock`. With the increase of the percentage of read-only operations the difference between Lock and RWLock diminishes and RWLock becomes dominant since read operations become more likely to be applied concurrently (for example, on 50% it is normal to have an execution without any parallelization: read operation, write operation, read operation, and so on). However, on 100% one could expect that RWLock should exhibit ideal throughput. Unfortunately, in this case, under the hood `ReentrantReadWriteLock` uses compare&swap on the shared variable that represents the number of current read operations. Read-only operations take enough time but not enough to amortize the considerable traffic introduced by concurrent compare&swaps. Thus, the plot for RWLock is almost flat, getting even slightly worse with the increase of the number of processes, and we blame the traffic for this.

6.2 Priority Queue

We run our algorithm (PC) with *flat combining* [21] as a combining data structure against six state-of-the-art concurrent priority queues: (1) the lock-free skip-list by Linden and Johnson (Linden SL [28]), (2) the lazy lock-based skip-list (Lazy SL [23]), (3) the non-linearizable lock-free skip-list by Herlihy and Shavit (SkipQueue [23]) as an adaptation of Lotan and Shavit's algorithm [34], (4) the lock-free skip-list from Java library (JavaLib), (5) the binary heap with flat combining (FC Binary [21]), and (6) the pairing heap with flat combining (FC Pairing [21]). [2] The code is available at `https://github.com/Aksenov239/FC-heap`.

We consider workloads parametrized by: 1) the initial size of the queue S ($8 \cdot 10^5$ or $8 \cdot 10^6$); and 2) the number P of working processes (from 1 to 64). We prepopulate the queue with S random integers chosen uniformly from the range $[0, 2^{31} - 1]$. Then we start P processes, and each process repeatedly performs operations: with equal probability it either inserts a random value taken uniformly from $[0, 2^{31} - 1]$ or extracts the minimum value.

[2] We are aware of the cache-friendly priority queue by Braginsky et al. [8], but we do not have its Java implementation.

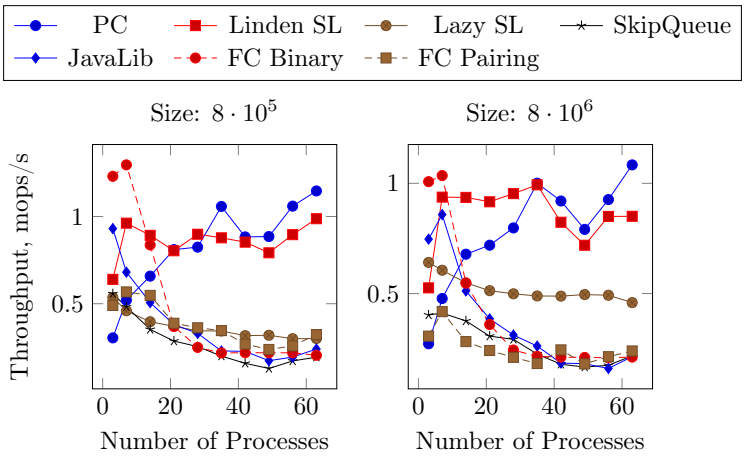

Figure 5 Priority Queue implementations.

For each setting and each algorithm, we run the corresponding workload for 10 seconds to warmup HotSpot JVM and then we run the workload five more times for 10 seconds. The average throughput of the last five runs is reported in Figure 5.

On a small number of processes (< 15), PC performs worse than other algorithms. With respect to Linden SL, Lazy SL, SkipQueue and JavaLib this can be explained by two different issues:

- Synchronization incurred by PC is not compensated by the work done;
- Typically, a combiner collects operations of only approximately half the processes, thus, we "overserialize", i.e., only $\frac{n}{2}$ operations can be performed in parallel.

In contrast, on small number of processes, the other four algorithms can perform operations almost with no contention. With respect to algorithms based on flat combining, FC Binary and FC Pairing, our algorithm is simply slower on one process than the simplest sequential binary and pairing heap algorithms.

With the increase of the number of processes the synchronization overhead significantly increases for all algorithms (in addition to the fact that FC Binary and FC Pairing cannot scale). As a result, starting from 15 processes, PC outperforms all algorithms except for Linden SL. Linden SL relaxes the contention during ExtractMin operations, and it helps to keep the throughput approximately constant. At approximately 40 processes the benefits of the parallel batched algorithm in PC starts prevailing the costs of explicit synchronization, and our algorithms overtakes Linden SL.

It is interesting to note that FC Binary performs very well when the number of processes is small: the overhead on the synchronization is very small, the processes are from the same core and the simplest binary heap performs operations very fast.

7 Related Work

To the best of our knowledge, Yew et al. [38] were the first to propose combining concurrent operations. They introduced a *combining tree*: processes start at distinct leaves, traverse upwards, and gain exclusive access by reaching the root. If, during the traversal, two processes access the same tree node, one of them adopts the operations of another and continues the traversal, while the other waits until its operations are completed. Several improvements of this technique have been discussed, such as adaptive combining tree [36], barrier implementations [20, 29] and counting networks [35].

A different approach was proposed by Oyama et al. [31]. Here the data structure is protected by a lock. A thread with a new operation to be performed adds it to a list of submitted requests and then tries to acquire the lock. The winner of the lock performs the pending requests on behalf of other processes from the list in LIFO order. The main drawback of this approach is that all processes have to perform CAS on the head of the list. The *flat combining* technique presented by Hendler et al. [21] addresses this issue by replacing the list of requests with a *publication list* which maintains a distinct *publication record* per participating process. A process puts its new operation in its publication record, and the publication record is only maintained in the list if the process is "active enough". This way the processes generally do not contend on the head of the list. Variations of flat combining were later proposed for various contexts [15, 16, 26, 14].

Hierarchical combining [22] is the first attempt to improve performance of combining using the computational power of clients. The list of requests is split into blocks, and each of these blocks has its own combiner. The combiners push the combined requests from the block into the second layer implemented as the standard flat combining with one combiner. This approach, however, may be sub-optimal as it does not involve *all* clients. Moreover, this approach works only for specific data structures, such as stacks or unfair synchronous queues, where operations could be combined without accessing the data structure.

In a different context, Agrawal et al. [2] suggested to use a *parallel batched* data structure instead of a concurrent one. They provide provable bounds on the running time of a dynamic multithreaded parallel program using P processes and a specified *scheduler*. The proposed scheduler extends the work-stealing scheduler by maintaining separate *batch* work-stealing deques that are accessed whenever processes have operations to be performed on the abstract data type. A process with a task to be performed on the data structure stores it in a *request array* and tries to acquire a global lock. If succeeded, the process puts the task to perform the batch update in its batch deque. Then all the processes with requests in the request array run the work-stealing routine on the batch deques until there are no tasks left. The idea of [2] is similar to ours. However, their algorithm is designed for systems with the fixed set of processes, whereas we allow the processes to join and leave the execution. From a more formal perspective, our goals are different: we aim at improving the performance of a *concurrent data structure* while their goal was to establish bounds on the running time of a *parallel program in dynamic multithreading*. Furthemore, implementing a concurrent data structure from its parallel batched counterpart for dynamic multithreading is only one of the applications of our technique, as sketched in Section 3.3.

8 Concluding remarks

Besides performance gains, parallel combining can potentially bring other interesting benefits.

First, a parallel batched implementation is typically provided with bounds on the running time. The use of parallel combining might allow us to derive bounds on the operations of resulting *concurrent* data structures. Consider, for example, a binary search tree. To balance the tree, state-of-the-art concurrent algorithms use the relaxed AVL-scheme [7]. This scheme guarantees that the height of the tree never exceeds the contention level (the number of concurrent operations) plus the logarithm of the tree size. Applying parallel combining to a parallel batched binary search tree (e.g., [4]), we get a concurrent tree with a strict logarithmic bound on the height.

Second, the technique might enable the first ever concurrent implementation of certain data types, for example, a *dynamic tree* [1].

As shown in Section 6, our concurrent priority queue performs well compared to state-of-the-art algorithms. A possible explanation is that the underlying parallel batched implementation is designed for *static* multithreading and, thus, it has little synchronization overhead. This might not be the case for implementations based on *dynamic* multithreading, where the overhead induced by the scheduler can be much higher. We intend to explore this distinction in the forthcoming work.

References

1. Umut A. Acar, Vitaly Aksenov, and Sam Westrick. Brief Announcement: Parallel Dynamic Tree Contraction via Self-Adjusting Computation. In *SPAA*, pages 275–277. ACM, 2017.
2. Kunal Agrawal, Jeremy T Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *SPAA*, pages 84–95. ACM, 2014.
3. Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. Parallel Combining: Benefits of Explicit Synchronization. *CoRR*, abs/1710.07588, 2018. `arXiv:1710.07588`.
4. Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *SPAA*, pages 253–264. ACM, 2016.
5. Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.
6. OpenMP Architecture Review Board. OpenMP Application Interface. `http://www.openmp.org`, 2008.
7. Luc Bougé, Joaquim Gabarro, Xavier Messeguer, Nicolas Schabanel, et al. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries. Technical report, ENS Lyon, 1998.
8. Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. CBPQ: High performance lock-free priority queue. In *Euro-Par*, pages 460–474. Springer, 2016.
9. Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998.
10. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Specifying Concurrent Problems: Beyond Linearizability and up to Tasks. In *DISC*, pages 420–435, 2015.
11. Thomas H Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. The MIT press, 3rd edition, 2009.
12. Travis Craig. Building FIFO and priorityqueuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, University of Washington, 02 1993., 1993.
13. Narsingh Deo and Sushil Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, 1992.
14. Dana Drachsler-Cohen and Erez Petrank. LCD: Local Combining on Demand. In *OPODIS*, pages 355–371. Springer, 2014.
15. Panagiota Fatourou and Nikolaos D Kallimanis. A highly-efficient wait-free universal construction. In *SPAA*, pages 325–334. ACM, 2011.
16. Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*, volume 47, pages 257–266. ACM, 2012.
17. Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the Cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.
18. Phillip B Gibbons. A more practical PRAM model. In *SPAA*, pages 158–168. ACM, 1989.
19. Gaston H Gonnet and J Ian Munro. Heaps on heaps. *SIAM Journal on Computing*, 15(4):964–971, 1986.

20 Rajiv Gupta and Charles R Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, 18(3):161–180, 1989.

21 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364, 2010.

22 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable Flat-Combining Based Synchronous Queues. In *DISC*, pages 79–93. Springer, 2010.

23 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2012.

24 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

25 Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.

26 Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Flat combining synchronized global data structures. In *7th International Conference on PGAS Programming Models*, page 76, 2013.

27 Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.

28 Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *International Conference On Principles Of Distributed Systems*, pages 206–220. Springer, 2013.

29 John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

30 Gil Neiger. Set-Linearizability. In *PODC*, page 396, 1994.

31 Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, volume 16, 1999.

32 Maria Cristina Pinotti and Geppino Pucci. Parallel priority queues. *Information Processing Letters*, 40(1):33–40, 1991.

33 Peter Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, 1998.

34 Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *IPDPS*, pages 263–268. IEEE, 2000.

35 Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions on Computer Systems (TOCS)*, 14(4):385–428, 1996.

36 Nir Shavit and Asaph Zemach. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

37 Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

38 Pen-Chung Y, Nian-Feng T, et al. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 100(4):388–395, 1987.