

# Parallel Combining: Benefits of Explicit Synchronization

Vitaly Aksenov<sup>1,2</sup>, Petr Kuznetsov<sup>3</sup>, and Anatoly Shalyto<sup>1</sup>

<sup>1</sup> ITMO University, Russia

<sup>2</sup> Inria Paris, France

<sup>3</sup> LTCI, Télécom ParisTech, Université Paris-Saclay

**Abstract.** *Parallel batched* data structures are designed to process synchronized *batches* of operations in a parallel computing model. 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 of parallel combining. First, we use our technique 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.

## 1 Introduction

Efficient concurrent data structures have to balance parallelism and synchronization. Parallelism implies performance, while synchronization maintains consistency. Efficient “concurrency-friendly” data structures (e.g., sets based on linked lists [21,22] or binary search trees [12,14]) are conventionally designed using hand-crafted fine-grained locking. In contrast, “concurrency-averse” data structures (e.g., staks and queues) are subject to frequent sequential bottlenecks and solutions based on combining (e.g., [23]), where all requests are synchronized and applied sequentially, perform surprisingly well compared to fine-grained ones [23]. Typically, a general data structure combines features of “concurrency-friendliness” and “concurrency-averseness”, and an immediate question is how to implement it in the most efficient way.

In this paper, we suggest a methodology of building a concurrent data structure from its *parallel batched* counterpart [2]. A parallel batched data structure applies a *batch* (set) of operations using parallelism. The algorithm distributes the work between the processes assuming the synchronized application of operations that reduces the number of possible interleavings. As a result, designing parallel batched algorithms is much easier than their concurrent counterparts.

In the technique proposed here, we *explicitly* synchronize concurrent operations, assemble them into batches, and apply these batches on an *emulated* parallel batched data structure. Processes share a set of active requests using any *combining* algorithm [15]: one of the active processes 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.

This technique becomes handy when the overhead on explicit synchronization of processes is compensated by the advantages of involving clients into the combining procedure using the parallel batched data structure. In the extreme case of concurrency-averse data structures, such as queues and stacks, having control over the batch can be used to *eliminate* certain requests and bypass sequential bottlenecks, even though there are no benefits of parallelizing the execution.<sup>1</sup> But as we show in the paper, combining *and* parallel batching pay off for data structures that offer some degree of parallelism, such as dynamic graphs and priority queues.

We discuss two applications of parallel combining and experimentally validate their benefits. 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 [27] that can be accessed for adding and removing edges (updates), as well as for checking connectivity between pairs of vertices (read-only). Second, we apply parallel combining to *priority queue* that are 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 performance 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 introduce several notions on data structures. In Section 3, we outline parallel combining. In Section 4, we present a novel parallel batched priority queue in a form convenient for parallel combining. In Section 5, we report the results of experiments. In Section 6, we overview the related work. We conclude in Section 7.

## 2 Background

**Data types and data structures.** A sequential *data type* is defined by a state machine consisting of a set of operations, a set of responses, a set of states, an

<sup>1</sup> Note that sequential combining [33,23,17,15], typically used in the concurrency-averse case, is a degenerate case of our parallel combining in which the combiner applies the batch sequentially.

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 primitives, such as Test&Set or Compare&Swap. A *concurrent implementation* (or *concurrent data structure*) for a given data type assigns, for each process and each operation of the data type, a deterministic 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 concurrent operations *take effect* instantaneously within their intervals [26].

**Batched data structures.** A *batched implementation* (or *batched data structure*) of a data type exports only one *apply* operation. 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 them on the data structure. We also consider extensions of the definition where we explicitly define the “batched” data type via the *set* [32] or *interval* [9] 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 [10].

**Static multithreading.** A parallel batched data structure specifies a distinct (sequential) code to each process in PRAM-like models: PRAM [29], Bulk synchronous parallel model [39], 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 processors, 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 implementation of a batch is written as a sequential read-write algorithm using concurrency keywords specifying logi-

cal parallelism, such as *spawn*, *sync*, *parallel-for* [10]. 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 “spawn” has two or more outgoing edges and a node that corresponds to a “sync” 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. Typically, the *work-stealing* scheduler is used (e.g., [5]) operating 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  $p$  assigns one of the ready successors to itself and puts the rest 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 and Applications

In this section, we describe the *parallel combining* technique in a parameterized form: the parameters are specified depending on the application.

**Combining Data Structure** Our technique relies on a combining data structure  $\mathbb{C}$  (e.g., [23]) that maintains a set of requests to 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 from an application-specific **STATUS\_SET**; 4) application-specific auxiliary fields. In our applications **STATUS\_SET** contains at least **INITIAL** and **FINISHED** values: **INITIAL** is set during the initialization and **FINISHED** means that the request is served.

$\mathbb{C}$  supports three operations: 1) **addRequest**( $r$  : **Request**) inserts request  $r$  into the set and returns whether this 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 rest of the paper we assume any black-box implementation of  $\mathbb{C}$  [33,23,17,15].

**Specifying Parameters** To perform an operation, a process executes the following steps (Figure 1): 1) it inserts the request into  $\mathbb{C}$  using **addRequest**( $\cdot$ ) and checks whether it becomes a combiner; 2) if it is the combiner, collects requests from  $\mathbb{C}$  using **getRequests**(), then it executes algorithm **COMBINER\_CODE**, and, finally, calls **release**() to enable another active process to become a combiner; 3) if the process is a client, it waits until the status of the request is not **INITIAL** and, then, executes algorithm **CLIENT\_CODE**.

```

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

```

Fig. 1: Parallel combining: pseudocode

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

Note that *sequential combining* [33,23,17,15] is a special case of parallel combining: we simply need to use the proper algorithm behind black box `C`. Now we discuss two interesting applications of parallel combining.

### 3.1 Read-optimized Concurrent Data Structures

Parallel combining can boost a sequential data structures under *read-dominated* workloads, i.e., when most operations do not modify the data structure. Such operations are called *read-only* operations, while other operations are called *update* operations.

Suppose that we are given a sequential data structure  $D$  that supports update and read-only operations. 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.

```

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
14  if req.status = STARTED:
15    apply(D, req.method, req.input)
16    req.status ← FINISHED
17
18  for r ∈ R:
19    while r.status = STARTED:
20      nop
21
22  CLIENT_CODE:
23    if not isUpdate(req.method):
24      apply(D, req.method, req.input)
25      req.status ← FINISHED

```

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

In `COMBINER_CODE` (Figure 2 Line 1), 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` (Figure 2 Line 21), 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 Figure 2 produces a linearizable concurrent data structure from a sequential one.*

*Proof.* Any execution can be split into combining phases (Figure 2 Lines 2-19) which do not intersect. 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 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 we apply it to the sequential *dynamic graph* data structure by Holm et al. [27] (Section 5.1).

### 3.2 Parallel Batched Algorithms

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

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. 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 the static multithreading case, each process is provided with a distinct version of `apply` function. Again, 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`.

We apply this technique to the *priority queue*. As a side result, we introduce a novel parallel batched implementation in a form of `COMBINER_CODE` and `CLIENT_CODE`. We had to design a new algorithm since no known implementation [34,13,8,35] 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.

## 4 Priority Queue with Parallel Combining

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

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

### 4.1 Batched Priority Queues: Review

To make use of parallel combining we first need to define a *batched* priority queue. Several batched priority queues were proposed in the literature for different parallel machines [29].

Pinotti and Pucci [34] proposed a batched priority queue for a  $p$ -processor CREW PRAM implemented as a heap each node of which contains  $p$  values in sorted order: only batches of size  $p$  are accepted.

Deo and Prasad [13] proposed a similar batched priority queue for a  $p$ -processor EREW PRAM which can accept batches of any size  $\leq p$ . But the batch processing time is proportional to  $p$ . For example, even if the batch consists of only one operations and only one process is involved in the computation, the execution still takes  $O(p \log m)$  time on a queue of size  $m$ .

Brodal et al. [8] proposed a batched priority queue that accepts batches of `Insert` and `DecreaseKey` operations, but not the batches of `ExtractMin` operations. The priority queue maintains a set of pairs (key, element) ordered by keys. `Insert` operation takes a pair  $(k, e)$  and inserts it. `DecreaseKey` operations takes a pair  $(d, e)$ , searches in the queue for a pair  $(d', e)$  such that  $d < d'$  and then replaces  $(d', e)$  with  $(d, e)$ .

Sanders [35] developed a randomized distributed priority queue for MIMD. MIMD computer has  $p$  processing elements that communicate via asynchronous message-passing. Again, the batch execution time is proportional to  $p$ , regardless of the batch size and the number of involved processes.

Bingmann et al. [3] described a variation of a priority queue by Sanders [35] for external memory and, thus, it has the same issue.

To summarize, all earlier implementations we are aware of are tailored to a fixed number of processes  $p$ . As a result, (1) the running time of the algorithm

always depends on  $p$ , regardless of the batch size and the number of involved processes; (2) once a data structure is constructed, we are unable to introduce more processes into system and use them efficiently. To respond to this issues, we propose a new parallel batched algorithm that applies a batch of size  $c$  to a queue of size  $m$  in  $O(c \cdot (\log c + \log m))$  RMRs for CC or DSM models. By that, our algorithm can use up to  $c \approx \log m$  processes efficiently.

## 4.2 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  $1, \dots, m$ . 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 satisfied: 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  $val \leftarrow x$ , 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.

## 4.3 Combiner and Client. Classes

Now, we describe our novel parallel batched priority queue in the form of **COMBINER.CODE** and **CLIENT.CODE** that fits the parallel combining framework described in Section 3. The code of necessary classes is presented in Figure 3, **COMBINER.CODE** is presented in Figure 4, and **CLIENT.CODE** is presented in Figure 5.

We introduce a sequential object **InsertSet** (Figure 3 Lines 1-20) that consists of two sorted linked lists  $A$  and  $B$  supporting *size* operation  $|\cdot|$ . The size of **InsertSet**  $S$  is  $|S| = |S.A| + |S.B|$ . **InsertSet** supports operation **split**:  $(X, Y) \leftarrow S.\text{split}(\ell)$ , which splits **InsertSet**  $S$  into two **InsertSet** objects  $X$  and  $Y$ , where  $|X| = \ell$  and  $|Y| = |S| - \ell$ . This operation is executed sequentially



```

1 class InsertSet:
2     List A, B
3
4     split(int l):
5         Pair<InsertSet, InsertSet>
6         L ← min(l, |A| + |B| - 1)
7         X ← new InsertSet()
8         if |A| ≥ L:
9             for i in 1..L:
10                 a ← X.A.removeFirst()
11                 X.A.append(a)
12         else:
13             for i in 1..L:
14                 b ← X.B.removeFirst()
15                 X.B.append(b)
16
17         if L = 1:
18             return (X, this)
19         else:
20             return (this, X)
21
22 class Node:
23     V val
24     bool locked
25     InsertSet split
26
27 class Request:
28     method: { ExtractMin, Insert }
29     V v
30     V res
31     STATUS_SET status
32     int start
33
34     // for client_insert(req)
35     // specifies the segment of leaves
36     // in a subtree of start node
37     int l, r

```

Fig. 3: Parallel Priority Queue. Classes

in  $O(L = \min(\ell, |S| - \ell))$  steps. The split operation works as follows: 1) new InsertSet  $T$  is created (Line 7); 2) if  $|S.A| \geq L$  then the first  $L$  values of  $S.A$  are moved to  $T.A$  (Lines 9-11); otherwise, the first  $L$  values from  $S.B$  are moved to  $T.B$  (Lines 13-15); note that either  $|S.A|$  or  $|S.B|$  should be at least  $L$ ; 3) if  $L = \ell$  then  $(T, S)$  is returned; otherwise,  $(S, T)$  is returned (Lines 17-20).

The heap is defined by its size  $m$  and an array  $a$  of Node objects. Node object (Figure 3 Lines 21-24) has three fields: value  $val$ , boolean  $locked$  and InsertSet  $split$ .

STATUS\_SET consists of three items: INITIAL, SIFT and FINISHED.

A Request object (Figure 3 Lines 26-36) consists of: a method  $method$  to be called and its input argument  $v$ ; a result  $res$  field; a  $status$  field; a node identifier  $start$ .

#### 4.4 ExtractMin Phase

**Combiner: ExtractMin preparation** (Figure 4 Lines 1-52). First, the combiner withdraws requests  $A$  from the combining data structure  $\mathbb{C}$  (Line 1). If the size of  $A$  is larger than  $m$ , the combiner serves the requests sequentially (Lines 3-7). Intuitively, if there are more Insert requests than the number of nodes in the corresponding binary tree, we cannot insert them in parallel.

In the following, we assume that the size of the queue is at least the size of  $A$ . The combiner splits  $A$  into sets  $E$  and  $I$  (Lines 9-16): the set of ExtractMin requests and the set of Insert requests. Then it finds  $|E|$  nodes  $v_1, \dots, v_{|E|}$  of heap with the smallest values (Lines 18-28), e.g., using the Dijkstra-like algorithm

```

1  A ← C.getRequests()
2
3  if m ≤ |A|:
4      apply A sequentially
5      for r ∈ A:
6          r.status ← FINISHED
7      return
8
9  E ← ∅
10 I ← ∅
11
12 for r ∈ A:
13     if isInsert(r):
14         I ← I ∪ r
15     else:
16         E ← I ∪ r
17
18 bestE ← new int[|E|]
19 heap ← new Heap<Pair<V, int>>()
20
21 heap.insert((a[1], 1))
22
23 for i in 1..|E|:
24     (v, id) ← heap.extract_min()
25     bestE[i] ← id
26     heap.insert((a[2 · v], 2 · v))
27     heap.insert(
28         (a[2 · v + 1], 2 · v + 1))
29
30 for i in 1..|E|:
31     E[i].res ← a[bestE[i]].val
32     a[bestE[i]].locked ← true
33     E[i].start ← bestE[i]
34
35 l ← min(|E|, |I|)
36 for i in 1..l:
37     a[bestE[i]] ← I[i].v
38     I[i].status ← FINISHED
39
40 for i in l + 1..|E|:
41     a[bestE[i]] ← a[m]
42     m--
43
44 for i in 1..|E|:
45     E[i].status ← SIFT
46
47 if req.status = SIFT:
48     client_extract_min(req)
49
50 for i in 1..|E|:
51     while E[i].status = SIFT:
52         nop
53
54 I ← I[1 + 1..|I|]
55
56 I[1].start ← 1
57 I[1].l ← 2⌊log2(m+|I|)⌋
58 I[1].r ← 2 · I[1].l - 1
59
60 for i in 2..|I|:
61     t ← m + i
62     power ← 1
63     while t > 1:
64         p ← t / 2
65         if 2 · p = t:
66             break
67         t ← p
68         power ← 2 · power
69
70 if t = 1:
71     t ← 2
72
73 I[i].start ← t + 1
74 I[i].l ← I[i].start · power
75 I[i].r ← I[i].l + power - 1
76
77 // L and R are global variables
78 // necessary for client_insert(req)
79 L ← m + 1
80 R ← m + |I|
81
82 m ← m + |I|
83
84 args ← new V[|I|]
85 for i in 1..|I|:
86     args[i] ← I[i].v
87
88 sort(args)
89
90 a[1].split ← new InsertSet()
91 for i in 1..|I|:
92     a[1].split.A.append(args[i])
93
94 for i in 1..|I|:
95     I[i].status ← SIFT
96
97 if req.status = SIFT:
98     client_insert(req)
99
100 for i in 1..|I|:
101     while I[i].status = SIFT:
102         nop

```

Fig. 4: Parallel Priority Queue. COMBINER.CODE

```

1 CLIENT_CODE:
2   if isInsert(req):
3       if req.status = SIFT:
4           client_insert(req)
5       else:
6           client_extract_min(req)
7   req.status ← FINISHED
8   return
9
10 client_extract_min(Request req):
11   v ← req.start
12   while 2 · v ≤ m:
13       while a[2 · v].locked
14           nop
15       if 2 · v + 1 ≤ m:
16           while a[2 · v + 1].locked:
17               nop
18           c ← 2 · v
19           if 2 · v + 1 ≤ m
20               and a[2 · v] > a[2 · v + 1]:
21               c ← 2 · v + 1
22           if a[c] > a[v]:
23               a[v].locked ← false
24               break
25           else:
26               swap(a[c], a[v])
27               a[c].locked ← true
28               a[v].locked ← false
29           v ← c
30   return
31
32 // Integers that specifies
33 // a segment of target nodes,
34 // i.e., m + 1 and m + |I|
35 global int L, R
36
37 targets_in_subtree(l, r): int
38   return min(r, R) - max(l, L) + 1
39
40 client_insert(Request req):
41   v ← req.start
42   while a[v].split = null:
43       nop
44   S ← a[v].split
45   a[v].split ← null
46
47   l ← req.l
48   r ← req.r
49
50   while v ∉ [L, R]:
51       a ← S.A.first()
52       b ← S.B.first()
53       if a[v] < min(a, b):
54           x ← a[v]
55           a[v] ← min(a, b)
56           if a < b:
57               S.A.pollFirst()
58           else:
59               S.B.pollFirst()
60           S.B.append(x)
61
62   mid ← (l + r) / 2
63   inL ←
64       targets_in_subtree(l, mid)
65   inR ←
66       targets_in_subtree(mid + 1, r)
67
68   if inL = 0:
69       v ← 2 · v + 1
70       l ← mid + 1
71
72   if inR = 0:
73       v ← 2 · v
74       r ← mid
75
76   if inL ≠ 0 and inR ≠ 0:
77       (S, T) ← S.split(inL)
78       a[2 · v + 1].split ← T
79       v ← 2 · v
80       r ← mid
81
82   if |S.A| ≠ 0:
83       a[v] ← S.A.first()
84   else:
85       a[v] ← S.B.first()
86   return

```

Fig. 5: Parallel Priority Queue. CLIENT\_CODE

in  $O(|E| \cdot \log |E|)$  primitive steps or  $O(|E|)$  RMRs. For each request  $E[i]$ , the combiner sets the  $E[i].res$  to  $a[v_i].val$ ,  $a[v_i].locked$  to **true**, and  $E[i].start$  to  $v_i$  (Lines 30-33).

The combiner proceeds to *eliminate* Insert requests in  $I$  by pairing them with ExtractMin requests in  $E$  (Lines 35-38). 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 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 decreases  $m$ , as in the sequential algorithm (Lines 40-42). Finally, the combiner sets the status of all requests in  $E$  to **SIFT** (Lines 44-45).

**Clients: ExtractMin phase** (Figure 5 Lines 11-29). Briefly, the clients sift down the values in nodes  $v_1, \dots, v_{|E|}$  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** (Lines 13-17). If  $a[v].val$ , the value of  $v$ , is less than the values in its children, then *sift down* is finished (Lines 23-24):  $c$  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  $w$  (Lines 26-29).

If the request of the combiner is ExtractMin, it also runs the code above as a client (Figure 4 Lines 47-48). The combiner considers the ExtractMin phase completed when all requests in  $E$  have status **FINISHED** (Lines 50-52).

#### 4.5 Insert Phase

**Combiner: Insert preparation** (Figure 4 Lines 53-99). For Insert requests, the combiner removes all completed requests from  $I$  (Line 53). Nodes  $m + 1, \dots, m + |I|$  have to be leaves, because we assume that the size of  $I$  is at most the size of the queue. We call these leaves *target nodes*. The combiner then finds all *split nodes*: nodes for which the subtrees of both children contain at least one target node. (See Figure 6 for an example of how target and split nodes can be defined.)

Since we have  $|I|$  target nodes, there are exactly  $|I| - 1$  split nodes  $u_1, \dots, u_{|I|-1}$ :  $u_i$  is the lowest common ancestor of nodes  $m + i$  and  $m + i + 1$ . They can be found in  $O(|I| + \log m)$  primitive steps (Lines 55-72): starting with node  $m + i$  go up the heap until a node becomes a left child of some node  $pr$ ; this  $pr$  is  $u_i$ . We omit the discussion about the fields  $l$  and  $r$  of  $I[i]$ : they represent the smallest and the largest leaf identifiers in the subtree of  $u_i$  at the lowest level, and they are used to calculate the number of leaves at the lowest level that are newly inserted, i.e.,  $m + 1, \dots, m + |I|$ , in constant time. The combiner sets  $I[1].start$  to the root (the node with identifier 1), (Line 55) and, for each  $i \in [2, |I|]$ , it sets  $I[i].start$  to the right child of  $u_{i-1}$  (node  $2 \cdot u_{i-1} + 1$ ) (Line 70). Then the combiner creates an InsertSet object  $X$ , sorts the arguments of the requests in  $I$ , puts them to  $X.A$  and sets  $a[1].split$  to  $X$  (Lines 81-89). Finally, it sets the status fields of all requests in  $I$  to **SIFT** (Lines 91-92).

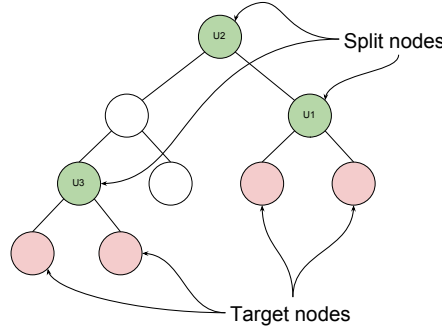


Fig. 6: Split and target nodes

**Clients: Insert phase** (Lines 41-85). Consider a client  $c$  with an in-completed request  $req$ . It waits while  $a[req.start].split$  is null (Lines 42-43). Now  $c$  is going to insert values from  $InsertSet\ a[req.start].split$  to the subtree of  $req.start$ . Let  $S$  be a local  $InsertSet$  variable initialized with  $a[req.start].split$ . For each node  $v$  on the path,  $c$  inserts values from  $S$  into the subtree of  $v$ .  $c$  calculates the minimum value  $x$  in  $S$  (Lines 51-53): the first element of  $S.A$  or the first element of  $S.B$ . If  $a[v].val$  is bigger than  $x$ , then the client removes  $x$  from  $S$ , appends  $a[v].val$  to the end of  $S.B$  and sets  $a[v].val$  to  $x$  (Lines 54-60). Note

that by the algorithm  $S.B$  contains only values that were stored in the nodes above node  $v$ , thus, any value in  $S.B$  cannot be bigger than  $a[v].val$  and after appending  $a[v].val$   $S.B$  remains sorted. Then the client calculates the number  $inL$  of target nodes in the subtree of the left child of  $v$  and the number  $inR$  of target nodes in the subtree of the right child of  $v$  (Lines 63-66, to calculate these numbers in constant time we use fields  $l$  and  $r$  of the request). If  $inL = 0$ , then all the values in  $S$  should be inserted into the subtree of the right child of  $v$ , and  $c$  proceeds with the right child  $2v + 1$  (Lines 69-70). If  $inR = 0$ , then, symmetrically,  $c$  proceeds with the left child  $2v$  (Lines 73-74). Otherwise, if  $inL \neq 0$  and  $inR \neq 0$ ,  $v$  is a split node and, thus, there is a client that waits at the right child  $2v + 1$ . Hence,  $c$  splits  $S$  to  $(X, Y) \leftarrow S.split(inL)$  (Line 77): the values in  $X$  should be inserted into the subtree of node  $2v$  and the values in  $Y$  should be inserted into the subtree of node  $2v + 1$ . Then  $c$  sets  $a[2v + 1].split$  to  $Y$ , sets  $S$  to  $X$  and proceeds to node  $2v$  (Lines 78-80). When  $c$  reaches a leaf  $v$  it sets the value  $a[v].val$  to the only value in  $S$  (Lines 82-85) and sets the status of the request  $req$  to **FINISHED** (Line 7).

If the request of the combiner is an incompleted Insert, it runs the code above as a client (Figure 4 Lines 94-95). The combiner considers the Insert phase completed when all requests in  $I$  have status **FINISHED** (Lines 97-99).

#### 4.6 Analysis

Now we provide theorems on correctness and time bounds.

**Theorem 2.** *Our concurrent priority queue implementation is linearizable.*

*Proof.* The execution can be split into combining phases (Figure 4 Lines 1-99) which do not intersect. We group the operations into batches corresponding to the combining phase in which they are applied.

Consider  $i$ -th combining phase. We linearize all the operations from the  $i$ -th phase right after the end of the corresponding `getRequests()` (Line 1) in the following order: at first, we linearize ExtractMin operations in the increasing order of their responses, then, we linearize Insert operations in any order.

To see that this linearization is correct it is enough to prove that the combiner and the clients apply the batch correctly.

**Lemma 1.** *Suppose that the batch of  $i$ -th combining phase contains a ExtractMin operations and  $b$  Insert operations with arguments  $x_1, \dots, x_b$ . Let  $V$  be the set of values stored in the priority queue before  $i$ -th phase. The combiner and the clients apply this batch correctly:*

- *The minimal  $a$  values  $y_1, \dots, y_a$  in  $V$  are returned to ExtractMin operations.*
- *After an execution the set of values stored in the queue is equal to  $V \cup \{x_1, \dots, x_b\} \setminus \{y_1, \dots, y_a\}$ . and the values are stored in nodes with identifiers  $1, \dots, |V| - b + a$ .*
- *After an execution the heap property is satisfied for each node.*

*Proof.* The first statement is correct, because the combiner chooses the smallest  $a$  elements from the priority queue and sets them as the results of ExtractMin requests (Lines 18-33).

The second statement about the set of values straightforwardly follows from the algorithm. During ExtractMin phase the combiner finds  $a$  smallest elements, replaces them with  $x_1, \dots, x_{\min(a,b)}$  and with values from the last  $a - \min(a,b)$  nodes of the heap: the set of values in the priority queue becomes  $V \cup \{x_1, \dots, x_{\min(a,b)}\} \setminus \{y_1, \dots, y_b\}$  and the values are stored in nodes  $1, \dots, |V| - a + \min(a,b)$ . Then, the *sift down* is initiated, but it does not change the set of values and it does not touch nodes other than  $1, \dots, |V| - a + \min(a,b)$ . During Insert phase the values  $x_{\min(a,b)+1}, \dots, x_b$  are inserted and new nodes which are used in Insert phase are  $|V| - a + \min(a,b) + 1, \dots, |V| - a + b$ . Thus, the final set of values is  $V \cup \{x_1, \dots, x_b\} \setminus \{y_1, \dots, y_a\}$  and the values are stored in nodes  $1, \dots, |V| - a + b$ .

The third statement is slightly tricky. At first, the combiner finds  $a$  smallest elements that should be removed and replaces them with  $x_1, \dots, x_{\min(a,b)}$  and with values from the last  $a - \min(a,b)$  nodes of the heap. Suppose that these  $a$  smallest elements were at nodes  $v_1, \dots, v_a$ , sorted by their depth (the length of the shortest path from the root) in non-increasing order. These nodes form a connected subtree where  $v_a$  is the root of the heap. Suppose that they do not form a connected tree or  $v_a$  is not the root of the heap. Then there exists a node  $v_i$  which parent  $p$  is not  $v_j$  for any  $j$ . This means that the values in nodes  $v_1, \dots, v_a$  are not the smallest  $a$  values: by the heap property a value at  $p$  is smaller than the value at  $v_i$ .

Now  $a$  processes perform *sift down* from the nodes  $v_1, \dots, v_a$ . We show that when a node  $v$  is unlocked, i.e., its `locked` field is set to false, the value at  $v$  is the smallest value in the subtree of  $v$ . This statement is enough to show that the heap property holds for all nodes after ExtractMin phase, because at that point all nodes are unlocked.

Consider an execution of *sift down*. We prove the statement by induction on the number of unlock operations. Base. No unlock happened and the statement is satisfied for all unlocked nodes, i.e., all the nodes except for  $v_1, \dots, v_a$ . Transition. Let us look right before the  $k$ -th unlock: the unlock of a node  $v$ . The left child  $l$  of  $v$  should be unlocked and, thus,  $l$  contains a value that is the smallest in its subtree. The same statement holds for the right child  $r$  of  $v$ .  $v$  chooses the smallest value between the value at  $v$  and the values at  $l$  and  $r$ . This value is the smallest in the subtree of  $v$ . Thus, the statement holds for  $v$  when unlocked.

After that, the algorithm applies the incompleted  $b - \min(a, b)$  Insert operations. We name the nodes with at least one target node in the subtree as *modified*. Modified nodes are the only nodes which value can be changed and, also, each modified node is visited by exactly one client. To prove that after the execution the heap property for each modified node holds: we show by induction on the depth of a modified node that if a node  $v$  is visited by a client with InsertSet  $S$  then: (1)  $S.A$  is sorted; (2)  $S.B$  is sorted and contains only values that were stored in ancestors of  $v$  after ExtractMin phase; and (3)  $v$  contains the smallest value in its subtree when the client finishes with it. Base. In the root  $S.A$  is sorted,  $S.B$  is empty and the new value in the root is either the first value in  $S.A$  or the current value in the root, thus it is the smallest value in the heap. Transition from depth  $k$  to depth  $k + 1$ . Consider a modified node  $v$  at depth  $k + 1$  and its parent  $p$ . Suppose that  $p$  was visited by a client with InsertSet  $S_p$ . By induction,  $S_p.A$  is sorted and  $S_p.B$  is sorted and contains only the values that were in ancestors. Then the client chooses the smallest value in  $p$ : either  $a[p]$ , the first value of  $S_p.A$  or the first value of  $S_p.B$ . Note that after any of these three cases  $S_p.A$  and  $S_p.B$  are sorted and  $S_p.B$  contains only values from ancestors and node  $p$ :

- $a[p]$  is the smallest, then  $S_p.A$  and  $S_p.B$  do not change;
- we poll the first element of  $S_p.A$  or  $S_p.B$ ;  $S_p.A$  and  $S_p.B$  are still sorted; then we append  $a[p]$  to  $S_p.B$ , and  $a[p]$  has to be the biggest element in  $S_p.B$ , since  $S_p.B$  contains only the values from ancestors.

Then the client splits  $S_p$  and some client, possibly, another one, works on  $v$  with IntegerSet  $S$ . Since,  $S$  is a subset of  $S_p$  then  $S.A$  is sorted and  $S.B$  is sorted and contains only the values from ancestors (ancestors of  $p$  and, possibly,  $p$ ). Finally, the client chooses the smallest value to appear in the subtree: the first value of  $S.A$ , the first value of  $S.B$  and  $a[v]$ .

**Theorem 3.** *Suppose that the combiner collects  $c$  requests using `getRequests()`. Then the combiner and the clients apply these requests to a priority queue of size  $m$  using  $O(c + \log m)$  RMRs in CC and DSM models each and  $O(c \cdot (\log c + \log m))$  RMRs in CC and DSM models in total.*

*Proof.* Suppose that the batch consists of  $a$  ExtractMin operations and  $b$  Insert operations.

The combiner splits requests into two sets  $E$  and  $I$  ( $O(c)$  RMRs, Lines 9-16). Then it finds  $a$  nodes with the smallest values ( $O(a \log a)$  primitive steps, but

$O(a)$  RMRs, Lines 18-28) using Dijkstra-like algorithm. After that, the combiner sets up ExtractMin requests, sets their status to SIFT and eliminates Insert requests ( $O(a)$  RMRs, Lines 30-45).

The clients participate in ExtractMin phase. At first, each client waits for its status to change (1 RMR). Then the client performs at most  $\log m$  iterations of the loop (Line 12): waits on the *locked* fields of the children ( $O(1)$  RMRs, Lines 13-17); reads the values in the children ( $O(1)$  RMRs, Line 20); compares these values with the value at the node, possibly, swap the values, lock the proper child and unlock the node ( $O(1)$  RMRs, Lines 22-29). When the client stops it changes the status (1 RMR, Line 7).

The combiner waits for the change of the status of the clients ( $O(a)$  RMRs, Lines 50-52). Summing up, in ExtractMin phase each client performs  $O(\log m)$  RMRs and the combiner performs  $O(a + \log m)$  RMRs, giving  $O(c + c \cdot \log m)$  RMRs in total.

The combiner throws away completed Insert requests ( $O(b)$  primitive steps and 0 RMRs, Line 53). Then it finds the split nodes ( $O(\log m + b)$  primitive steps, but 0 RMRs, Lines 55-72). After that the combiner sorts arguments of remaining Insert requests, sets their status to SIFT and sets up the initial InsertSet ( $O(b \cdot \log b)$  primitive steps, but  $O(b)$  RMRs, Line 81 and Line 92).

The clients participate in Insert phase. At first, a client  $t$  waits while the corresponding InsertSet is null (1 RMR, Lines 41-43). Suppose that it reads the InsertSet  $S$  and starts the traversal down. The client performs at most  $\log m$  iterations of the loop (Line 50): choose the smallest value ( $O(1)$  RMRs, Lines 51-60), find whether to split InsertSet ( $O(1)$  RMRs, Lines 62-74), split InsertSet (calculated below, Line 77) and passe one InsertSet to another client ( $O(1)$  RMRs, Lines 78-80). Now let us calculate the number of RMRs spent in Line 77. Suppose that there are  $k$  iterations of the loop and the size of  $S$  at iteration  $i$  is  $s_i$ . At  $i$ -th iteration split works in  $O(\min(s_{i+1}, s_i - s_{i+1})) = O(s_i - s_{i+1})$  primitive steps and RMRs. Summing up through all iterations we get  $O(s_1) = O(b)$  RMRs spent by  $t$  in Line 77. Finally,  $t$  sets the value in the leaf ( $O(1)$  RMRs, Lines 82-85) and changes the status ( $O(1)$  RMRs, Line 7).

The combiner waits for the change of the status of the clients ( $O(b)$  RMRs, Lines 97-99). Summing up, in Insert phase the clients and the combiner perform  $O(b + \log m)$  RMRs each. Consequently, the straightforward bound on the total number of RMRs is  $O(c^2 + c \cdot \log m)$  RMRs.

To get the improved bound we carefully calculate the total number of RMRs spent on the splits of InsertSets in Line 77. This number equals to the number of values that are moved to newly created sets during the splits. For simplicity we suppose that inserted values are bigger than all the values in the priority queue and, thus, each InsertSet contains only the newly inserted values. This assumption does not affect the bound. Consider now the inserted value  $v$ . Suppose that  $v$  was moved  $k$  times and at  $i$ -th time it was moved during the split of InsertSet with size  $s_i$ . Because  $v$  is moved during split only to the set with the smaller size:  $s_1 \geq 2 \cdot s_2 \geq \dots \geq 2^{k-1} \cdot s_k$ .  $k$  is less than  $\log c$ , because  $s_1 \leq c$ , and, thus,  $v$  was moved no more than  $\log c$  times. This means, that in total during the splits of



InsertSets no more than  $c \cdot \log c$  values are moved to new sets, giving  $O(c \cdot \log c)$  RMRs during the splits. This gives us a total bound of  $O(c \cdot (\log c + \log m))$  RMRs during Insert phase.

To summarize, the combiner and the clients perform  $O(c + \log m)$  RMRs each and  $O(c \cdot (\log c + \log m))$  RMRs in total.

## 5 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.

### 5.1 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. [27]. 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) against three others: (1) Lock, based on ReentrantLock from `java.util.concurrent`; (2) RW Lock, based on ReadWriteReentrantLock from `java.util.concurrent`; and (3) FC, based on *flat combining* [23]. 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: 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 tree, while other update operations only check the existence of the edge and do not modify the graph. *Trees* workloads are interesting because a reasonable small number (approximately, 5-10%) of update operations modify the 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 only

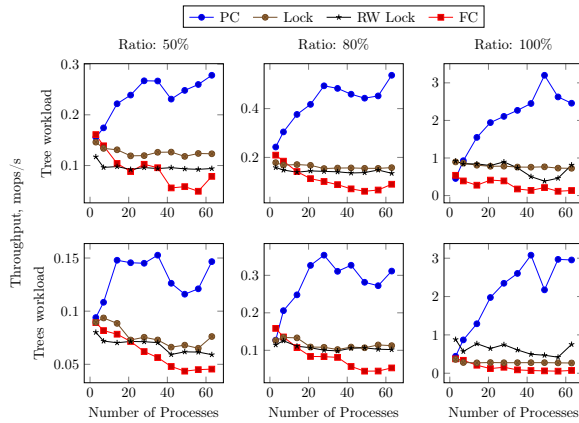


Fig. 7: Dynamic graph implementations

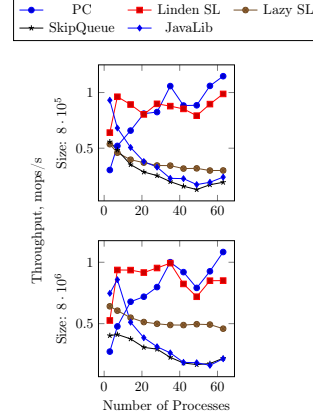


Fig. 8: Priority Queue implementations

can modify the set of edges and 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 7.

From the plots we can make two general claims: PC exhibits the highest throughput over all considered implementations and it is the only implementation which throughput scales up with the number of the processes. On 100% workload we expect the throughput curve to be almost linear since all operations are read-only and can be parallelized. The plots almost confirm our expectation: the curve of the throughput is a linear function with coefficient  $\frac{1}{2}$  (instead of ideal coefficient 1). We note that this 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 percentage of read-only operations we expect that the throughput curve becomes flatter since the update operations cannot be parallelized. Plots for 50% and 80% workloads confirm the last claim.

Besides explaining the behaviour of PC curves it is interesting to point several features of other implementations. At first, FC implementation works slightly worse than Lock and RW Lock. We explain this with two facts:

- The algorithm (ReentrantLock) behind Lock and RWLock implementations is based on CLH Lock [11]. Under high-load CLH lock can be seen as flat combining with less overhead: we already know the next process to perform an operation and there is no reason to have a special combiner process that gathers the requests.

- ReentrantLock is written and highly optimized by experts.

Second, it is interesting to observe that RWLock implementation is not so superior to Lock even on read-only workloads. This is due to the fact that ReentrantReadWriteLock is still implemented using queue and under high-load the processes spend more time on waiting to insert themselves into the queue rather than performing an operation.

## 5.2 Priority Queue

We compare our algorithm (PC) against four state-of-the-art concurrent priority queues: the lock-free skip-list by Linden and Johnson (Linden SL [30]), the lazy lock-based skip-list (Lazy SL [25]), the non-linearizable lock-free skip-list by Herlihy and Shavit (SkipQueue [25]) as an adaptation of Lotan and Shavit’s algorithm [36], and the lock-free skip-list from Java library (JavaLib).<sup>2</sup> The code is available at <https://github.com/Aksenov239/FC-heap>.

We consider workloads parametrized by: 1) the initial size  $S$  ( $8 \cdot 10^5$  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.

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 8.

For less than 15 processes, PC performs worse than other algorithms. This can be explained by two different issues:

- PC incurs considerable amount of synchronization in comparison to the work done;
- Typically, a combiner collects operations of only approximately half the processes, thus, we oversequentialize, i.e., only  $\frac{n}{2}$  operations can be performed in parallel.

These issues are observable on small number of processes because other algorithms can perform operations with small amount of contention leading to the similar problem as with the binary search tree. But with the increase of the number of processes the synchronization overhead significantly increases especially for Lazy SL, SkipQueue and JavaLib. As a result, starting from 15 processes, PC outperforms these three algorithms. Linden SL relaxes the contention during ExtractMin operations, and it helps to keep the throughput approximately constant.

This can be explained by the cost of the synchronization: on small number of processes, PC incurs considerable synchronization while other algorithms can

<sup>2</sup> We are aware of the cache-friendly priority queue by Braginsky et al. [7], but we do not have its Java implementation.

perform operations almost with no contention. The overhead due to contention for other algorithms (especially for Lazy SL, SkipQueue and JavaLib) significantly increases with the number of processes. As a result, starting from 15 processes, PC outperforms these three algorithms. 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.

## 6 Related Work

To the best of our knowledge, the first attempt to combine concurrent operations was introduced by Yew et al. [40]. They introduced a *combining tree*: processes start at the leaves and traverse upwards to 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 stops its traversal and waits until its operations are completed. Several improvements of this technique have been discussed, such as adaptive combining tree [38], barrier implementations [20,31] and counting networks [37].

A different approach was proposed by Oyama et al. [33]. 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 process that acquires the lock performs the pending requests on behalf of other processes from the list in LIFO order, and later removes them from the list. Its main drawback is that all processes have to perform CAS on the head of the list. The *flat combining* technique presented by Hendler et al. [23] 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 sufficiently active. This way the process generally does not need to perform CAS on the head of the list. Variations of flat combining were later proposed for various contexts [16,17,28,15].

*Hierarchical combining* [24] is the first attempt to improve performance of combining using the computation 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 have *all* clients participating. 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.

Agrawal et al. [2] decide to use a *parallel batched* data structure instead of a concurrent one in a different context. 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 dequeues 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 dequeues until there are no tasks left. The idea of [2] is similar to ours. However, 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 for dynamic multithreading*. Implementing a concurrent data structure from its parallel batched counterpart for dynamic multithreading is only one of the applications of our parallel combining, as shown in Section 3.2.

## 7 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 [6]. This scheme provides the linear guarantee for the height of the tree: it does not exceed the number of concurrent operations plus the logarithm of the 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 5, our concurrent priority queue performs well compared to state-of-the-art algorithms. We assume that one of the reasons 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 check this 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. Timo Bingmann, Thomas Keh, and Peter Sanders. A bulk-parallel priority queue in external memory with stxxl. In *International Symposium on Experimental Algorithms*, pages 28–40. Springer, 2015.
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. 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.
7. Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. Cbpq: High performance lock-free priority queue. In *Euro-Par*, pages 460–474. Springer, 2016.
8. 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.
9. Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Specifying concurrent problems: Beyond linearizability and up to tasks - (extended abstract). In *DISCs*, pages 420–435, 2015.
10. Thomas H. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, chapter 6, pages 151–169. The MIT press, 3rd edition, 2009.
11. 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.
12. Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *Euro-Par*, pages 229–240. Springer, 2013.
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, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. *ACM SIGPLAN Notices*, 49(8):343–356, 2014.
15. Dana Drachsler-Cohen and Erez Petrank. Lcd: Local combining on demand. In *OPODIS*, pages 355–371. Springer, 2014.
16. Panagiota Fatourou and Nikolaos D Kallimanis. A highly-efficient wait-free universal construction. In *SPAA*, pages 325–334. ACM, 2011.
17. Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*, volume 47, pages 257–266. ACM, 2012.
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. Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.
22. Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, pages 3–16, 2006.
23. Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364, 2010.

24. Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable flat-combining based synchronous queues. In *DISC*, pages 79–93. Springer, 2010.
25. Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.
26. Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
27. 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.
28. 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.
29. Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.
30. 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.
31. 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.
32. Gil Neiger. Set-linearizability. In *PODC*, page 396, 1994.
33. 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.
34. Maria Cristina Pinotti and Geppino Pucci. Parallel priority queues. *Information Processing Letters*, 40(1):33–40, 1991.
35. Peter Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, 1998.
36. Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *IPDPS*, pages 263–268. IEEE, 2000.
37. Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions on Computer Systems (TOCS)*, 14(4):385–428, 1996.
38. Nir Shavit and Asaph Zemach. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.
39. Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.
40. 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.