Applications and Extensions of Parallel Self-adjusting Computation

Anubhav Baweja

CMU-CS-21-133

August 2021

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy Blelloch, Chair
Umut Acar

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Copyright © 2021 Anubhav Baweja

Keywords: Parallel Algorithms, Self-adjusting Computation, Binary Search Trees, Dynamic Convex Hull

To my parents Rita and Aneesh, and my brother Ashish.

Abstract

Self-adjusting computation is an approach for automatically dynamizing static algorithms. That is, given an algorithm which supports a set of query operations on a given input data, self-adjusting computation can help support efficient update operations on the input data simultaneously. Recently, Anderson et al. (SPAA '21) proposed a framework for writing parallel algorithms in such a framework with provable bounds on the work and span of the algorithms written in the framework, while also providing compelling empirical evidence of efficiency. This report deals with two key applications of the framework: binary search trees with parallel operations such as Filter and MapReduce, and a dynamic convex hull algorithm inspired by Overmars' algorithm. This report also discusses an imperative extension of parallel self-adjusting computation: the framework presented by Anderson et al. has a write-once restriction for its key variables, and some parallel algorithms and data structures such as Breadth-first search and hash tables are imperative by nature, proving a need for imperative parallel self-adjusting computation.

Acknowledgments

I would like to thank my advisor Guy Blelloch for guiding me all along the project and helping me with both the practical and theoretical aspects of the project. I would also like to thank Daniel Anderson, who explained his implementation of the PSAC framework to me and helped me use and modify it, along with advising me on C++ related implementation details.

Contents

1 Introduction **1**
1.1 Related Work . . . 2
1.2 Framework . . . 3
1.3 Stability and Computation Distance . . . 4
1.4 Subarray Sums . . . 5

2 Binary Search Trees **9**
2.1 Structure of a Node . . . 9
2.2 Basic operations: Split and Join . . . 10
2.3 Filter and Mapreduce . . . 13
2.3.1 Filter Implementation . . . 13
2.3.2 Mapreduce Implementation . . . 14
2.4 Computation Distance for Batch Inserts . . . 15
2.5 Granularity Control . . . 18
2.6 Benchmarks . . . 19

3 Dynamic Convex Hull **23**
3.1 Overview of Overmars' Algorithm [24] . . . 23
3.2 Using Binary Search Trees . . . 24
3.3 Implementation . . . 25
3.4 Computation Distance for Convex Hull . . . 28
3.5 Granularity Control . . . 28
3.6 Benchmarks . . . 30

4 Imperative Parallel Self-Adjusting Computation **33**
4.1 Previous Work and Assumptions . . . 33
4.2 Write Nodes . . . 34
4.3 Parallel Order Maintenance . . . 35
4.4 Implementation . . . 35
4.5 Breadth First Search . . . 37

Bibliography **39**

List of Figures

2.1 Absolute throughput (input updates per second) vs number of cores (initial computation) . . . 20
2.2 Absolute throughput (input updates per second) vs number of cores (updates) . . 20
2.3 Absolute throughput (input updates per second) vs update size 21
2.4 Relative Speedup with respect to sequential baseline vs number of cores (initial computation) . . . 21
2.5 Relative Speedup with respect to sequential baseline vs number of cores (updates) 22
2.6 Relative Speedup with respect to sequential baseline vs update size 22

3.1 Convex Hull: Absolute throughput (input updates per second) vs number of cores 31
3.2 Convex Hull: Speedup relative to sequential baseline vs number of cores 31
3.3 Convex Hull: throughput and speedup vs update size 32

List of Tables

2.1 Filter Benchmark 19
2.2 Mapreduce Benchmark 19

3.1 Convex Hull Benchmark 30
3.2 Granularity Control Benchmarks for Convex Hull 32

Chapter 1

Introduction

Self-adjusting Computation is a framework for automatically dynamizing static data structures and algorithms [1]. That is, given an algorithm which can handle queries efficiently on some given data, a self-adjusting computation framework allows the programmer to handle updates on the data while automatically updating the results of existing queries within this framework without any additional programming overhead: only the parts of the query that are relevant to the change in the data are re-evaluated while updating the result of the query. Moreover, in a lot of cases, there is a small or no increase in the asymptotic complexity of these queries, while providing the benefit of efficient updates.

The approach works by encapsulating data in special structures so that we can keep track of where parts of it are read in the program. By keeping track of these data dependencies, if some data is updated, then the change can be propagated in such a way that only the parts of the algorithm that rely on that data are affected.

In this thesis, we will explore a framework proposed by Anderson et al. [7] used for implementing parallel self-adjusting computations, and discuss its various applications and extensions. Specifically, we will investigate binary search trees and dynamic convex hulls in the framework, and the cost of propagating changes. The current parallel self-adjusting computation (PSAC) framework only supports algorithms which obey a write-once restriction on the special structures mentioned before, so we will also be exploring a version of the framework that can support imperative code: allowing multiple writes to the same tracked variable.

This framework can be used to dynamize a wide variety of parallel algorithms along with good theoretical bounds on the work and span. The primary idea is to track all data dependencies in a Series-Parallel (SP) tree, which was first introduced by Feng et al [17]. This is a binary tree where computations are stored at different Series (S) nodes in the tree. The two children of an S node are executed sequentially (first left, then right). There are also Parallel (P) nodes such that their two children store computations that are executed in parallel. Anderson et al. [7] extended this idea to add Read (R) nodes to create RSP trees, which help track dependencies between reads and writes to help propagate changes efficiently. These R nodes have an associated *closure*, which is a computation where the value read by the R node is in scope. This closure is

rerun by the change propagation algorithm if the value read by the node changes.

Programs written in the above framework write input data and non-local values that depend on them into *modifiables references* or *mods* for short. These mods store a list of all read nodes that read from them so that changes can be propagated appropriately if the value inside the mod changes. The above work, similar to the other previous work on sequential self-adjusting computation [5], restricts the class of programs to the ones which write to mods only once. Note that this is still a wide class of programs that includes all race-free functional programs, and since local variables do not need to be stored in mods, they can be written to multiple times so this framework can also support partially imperative programs. Note that there has been work done to remove the write-once restriction in the sequential setting [6], and in this project we will also explore it in the parallel setting.

1.1 Related Work

There has been considerable work on self-adjusting computation, parallel self-adjusting computation, and imperative self-adjusting computation. Acar [1] first presented the idea of self-adjusting computation in the sequential setting by using the idea of trace stability to bound the work done by change propagation for different algorithms: an idea that the work by [7] also uses. They also present several applications such as sorting algorithms like merge sort and quick sort, convex hull algorithms such as Graham's scan [18] and quick hull [8], and the tree contraction algorithms of Miller and Reif [21]. They discuss several language techniques in order to implement self-adjusting computation such as

- **Adaptive Functional Language (AFL)** [3]: the interface for this language is very similar to the one proposed by Anderson et al. [7]: they use mods and special functions for reading and writing to mods. Internally, the computation is stored in a directed acyclic graph where the vertices are evaluations of mods and the edges are evaluations of reads. This is called a dynamic dependence graph (DDG) and is used to prove bounds on change propagation. Note that this framework is built for a purely functional language.
- **Self-adjusting functional language (SLf)**: The above work was extended to the SLf framework, where the AFL framework was combined with the memoized functional language (MFL) framework [4] to achieve a better bound for change propagation. This model assumes access to the explicit management of memory, something that the PSAC framework also assumes. This framework is also applicable to a purely functional language.
- **Imperative self-adjusting computation** [6]: this framework builds on the SLf framework and finally allows multiple writes on the same mods. This framework helps us go beyond the scope of purely functional programs.

Other work on self-adjusting computation has been done such as CEAL [4]: a self-adjusting language for a low-level language such as C. A survey of self-adjusting computation (in the sequential setting) is presented by Acar [2].

Since then, a lot of work has been done on parallel self-adjusting computation as well:

- **Parallel Adaptive Language (PAL)**: Hammer et al [19] proposed PAL: a language for parallel self-adjusting computation (which is not fully implemented). It uses a similar SP-tree like structure for maintaining computational dependencies. This model, however, assumes the existence of data structures that we do not have, such as a concurrent fully dynamic lowest common ancestor (LCA) data structure. Their evaluation consists of a simulation of work that would be performed by the said data structure if it did exist, rather than a real benchmark. They also treat the mods imperatively: allowing multiple writes on the same mod, acknowledging the merits of having imperative mods for efficient parallelization. However, for the proof of correctness of change propagation, they assume that the mods are written to at most once.
- **Incoop**: Bhatotia et al. [11] presented a framework for parallel self-adjusting computation in the map-reduce framework. The model is specifically made for map-reduce-like computations in a distributed system. However, there are no theoretical guarantees of change propagation.
- **Two for the Price of One**: Burckhardt et al. [15] were the first to propose a general-purpose framework for parallel self-adjusting computation. They extend the concurrent revisions model with self-adjusting primitives, which enables them to create a DDG and record and re-execute certain computations. Only the computations that necessarily need to be re-executed on an input change are re-executed. They however, do not provide any theoretical guarantees for their framework.
- **iThreads**: Bhatotia et al [12] proposed a parallel self-adjusting framework which acts a drop-in replacement for pthreads. This way, the user does not need to use special primitives to make their algorithm dynamic, the dynamization is completely automatic. However, since the units of computations that can be re-executed on detecting a change are entire pthread computations, it makes the dynamization restrictive and coarse-grained. They also do not provide any theoretical guarantees of the runtime of updates.

Therefore the PSAC framework proposed by Anderson et al. [7] is the first one that provides solid theoretical guarantees of the work and span required by change propagation, while also providing a number of applications such as raytracing, Rabin-Karp fingerprinting [20], dynamic trees, filter on binary search trees etc.

1.2 Framework

The PSAC framework proposed by Anderson et al. [7] has the following primitives:

- **write**($\mathsf{dest} : \alpha$ **mod**, $\mathsf{value} : \alpha$). dest is the modifiable that the operation writes to and the value written is value.
- **alloc_mod**(T : **type**) : T **mod**. This operation is used for dynamically allocating mods within the algorithm.
- **read**($\mathsf{m} : (\alpha_1$ **mod**$, ..., \alpha_k$ **mod**$), \mathsf{r} : (\alpha_1, ..., \alpha_k) \mapsto ())$. m is the set of mods that is read by this reader, and r is the associated closure. Note that for the sake of simplicity the closure does not have a return value.

- **psac_par**(left_f : () $\mapsto$ (), right_f : () $\mapsto$ ()). left_f and right_f are the two computations that are executed in parallel.
- **run**(f : () $\mapsto$ ()) : S. The operation used run the computation f and to create the initial RSP tree out of the self-adjusting computation. Returns a *computation* object which can later be used to propagate changes.
- **propagate**(root : S). The operation used to propagate changes using the *computation* object obtained from using **run**. If some **write** operations occur after the **run** operation then there will some marked nodes in the RSP tree as described in the previous section which would need to be resolved using this process.

1.3 Stability and Computation Distance

Before we dive into the more technical details of the change propagation algorithm of Anderson et al. [7] and the algorithms that we can write using them, we need to learn what stability and computation distance are.

Definition 1.3.1. *[5] Given an algorithm A and an input I on which A is executed, we define $A_T(I)$ as the **trace** of the algorithm: the trace essentially captures the operations performed by the algorithm. We say that A is an $O(f(n))$**-stable** algorithm for a class of input changes, if $max_{(I,I')\in\Delta_n}\delta(A_T(I), A_T(I')) \in O(f(n))$, where Δ_n is the relation describing the set of inputs of size n obtained after an input change, and δ is some measure of distance between traces. For example, the trace can be the function-call tree of the execution, and the trace distance δ can be defined as the sum of the work of the function calls that differ in two traces $A_T(I)$ and $A_T(I')$.*

The general idea behind stability is that $O(f(n))$-stable algorithms, where $f(n)$ is a slow growing function such as $\log n$, are ideal candidates for self-adjusting computation. Since their traces do no change a lot, if the change propagation algorithm is good at identifying these small changes then updates will be very efficient.

Say we have an algorithm programmed in the framework, creating an RSP tree. Say there is a change in the value stored in a mod: a change made either to the input data by the user or to an internal mod by the self-adjusting function. We can identify all the read nodes which now read a different value than they did before change propagation. These are called *affected readers*. Now, instead of executing the closures of all the affected readers immediately, we do it lazily: we mark the affected readers and their ancestors in the RSP tree. Once all the appropriate nodes are marked, we can traverse down the RSP tree and rerun the closures of the affected readers in the correct order: if there is an S node, and an affected reader α is in its left subtree and another affected reader β is in its right subtree, then the closure of α must be run before that of β, since reruning that closure might change something about the computation (for example, the closure of α might change the value read by β, or it might lead to the deletion of β entirely, so they need to be executed in order). Similarly if α and β are in the left and right subtrees of a P node then their closures need to be run in parallel.

A more robust definition of affected readers is as follows:

Definition 1.3.2. *Consider an algorithm in the PSAC framework, and two inputs I, I' to the algorithm, and let T, T' be the RSP trees generated by the executions of the algorithm on these inputs. A reader is uniquely identified by its closure, and the value of mods it reads. A reader node is said to be subsumed by another read node if the first one is in the scope of the second one (the first one is created while executing the second one's closure). Two read nodes $v \in T$ and $v' \in T'$ are cognates if the paths in the two trees to the nodes are identical, that is, they comprise of the same chain of S and P nodes. A pair of cognate nodes are called **affected** if they are cognates and are not subsumed by another such pair of read nodes.*

Using this notion of affected readers, we can finally define what computation distance is:

Definition 1.3.3. *The cost of an affected reader is the work performed in its closure, assuming that all self-adjusting computation primitives are $O(1)$. The **computation distance** between two executions of the same algorithm intuively is the work done by one execution but not the other. In other words, if T and T' are the RSP trees produced by inputs I and I', then the computation distance between the computations on the inputs I and I' is the sum of the cost of the affected readers of T and T'.*

With this definition, we can present the key result of Anderson et al. [7] with respect to the efficiency of change propagation:

Theorem 1.3.4. *Consider an algorithm in the PSAC framework and two inputs to the algorithm. Let W_Δ be the computation distance between the two computations, R_Δ be the number of affected readers, and s is the span of the entire computation. The change propagation algorithm can update the first algorithm to yield the second one in $O(W_\Delta + R_\Delta \cdot h)$ work and $O(s(h + \log r))$ span with high probability where h is the maximum height of the RSP tree and r is the maximum number of reads for any mod.*

In the above theorem, a cost of $O(f(n))$ with high probability refers to a cost of $O(c \cdot f(n))$ with probability at least $1 - n^{-c}$ where $c \geq 1$. Note that the overhead for work is very small, and if the number of reads per mod is constant then there is only an additive overhead of $O(R_\Delta \cdot h)$. This quantity is also very small for good parallel algorithms, since h should be logarithmic in terms of the total work. The overhead for the span is also small due to the same reasons.

1.4 Subarray Sums

A great motivating example for self-adjusting computation is as follows: given an input array A of integers of length n, we want to support subarray sum queries. Without any updates, these queries can be supported in $O(1)$ time by storing prefix sums S. In order to compute the subarray sum $A[l : r]$ (right index exclusive), we simply return $S[r] - S[l]$. However, if we wish to support an update operation while maintaining these prefix sums, then each update requires $O(n)$ time since updating $A[0]$ changes all n prefix sums.

The classical way of solving this problem is to use a segment tree, which supports both queries and updates in $O(\log n)$ time. The key advantage of using a self-adjusting computation framework is that if the user writes the query function in the same way as it is written for a

segment tree, then the user does not need to write the update function, and they can get $O(\log n)$ time queries and updates.

Algorithm 1 Building segment tree

1: **function** BUILD(A : **int array**, a : **int**, b : **int**, S : **int array**, nd : **int**):
2: **if** $a + 1 = b$ **then**
3: $T[nd] \leftarrow A[a]$
4: **else**
5: **local** $m \leftarrow a + (b - a)/2$
6: BUILD($A, a, m, S, 2 \times nd$)
7: BUILD($A, m, b, S, 2 \times nd + 1$)
8: $S[nd] \leftarrow S[2 \times nd] + S[2 \times nd + 1]$
9: **end if**

Algorithm 2 Query function for subarray sum: obtain sum of $A[l : r]$ by calling QUERY($A, l, r, 0, n, S, 1$)

1: **function** QUERY(A : **int array**, l : **int**, r : **int**, a : **int**, b : **int**, S : **int array**, nd : **int**):
2: **if** $r \leq a$ **or** $b \leq l$ **then**
3: **return** 0
4: **end if**
5: **if** $l \leq a$ **and** $b \leq r$ **then**
6: **return** $S[nd]$
7: **end if**
8: **local** $m \leftarrow a + (b - a)/2$
9: **local** $x \leftarrow$ QUERY($A, l, r, a, m, S, 2 \times nd$)
10: **local** $y \leftarrow$ QUERY($A, l, r, m, b, S, 2 \times nd + 1$)
11: **return** $x + y$

Algorithm 3 Update function for subarray sum: set $A[i]$ to v by calling UPDATE$(A, i, v, 0, n, S, 1)$

```
1: function UPDATE(A : int array, i : int, v : int, a : int, b : int, S : int array, nd : int):
2: if b ≤ i ∨ i < a then
3:     return
4: end if
5: if a = b ∧ a = i then
6:     A[i] ← v
7:     S[nd] ← v
8: end if
9: local m ← a + (b − a)/2
10: UPDATE(A, i, v, a, m, S, 2 × nd)
11: UPDATE(A, i, v, m, b, S, 2 × nd + 1)
12: S[nd] ← S[2 × nd] + S[2 × nd + 1]
```

The segment tree is first created by the BUILD function, and the subarray sums of length $n, n/2, n/4...$ are stored in the S array. The nd variable keeps track of the nodes in this array: the left child of nd is $2 \times nd$ and the right child is $2 \times nd + 1$. The QUERY function identifies the appropriate $O(\log n)$ nodes of this segment tree such that their sum is the required sum. The UPDATE function updates all the nodes on the path from the updated leaf to the root of the segment tree by recalculating their sum. Note that this function also only takes $O(\log n)$ time and only $O(\log n)$ values in S are changed due to this update. Recalling the notion of stability, we can therefore infer that this algorithm is $O(\log n)$-stable, and is therefore a great candidate for self-adjusting computation. Moreover, since the build function recursively builds the left and right subtrees independent of each other, it is also a great candidate for parallelism, and can be written in the PSAC framework as follows:

Algorithm 4 Building segment tree in the PSAC framework

```
1: psac_function BUILD(A : int mod array, a : int, b : int, T : int mod array, nd : int):
2: if a + 1 = b then
3:     with read (A[a]) as x do
4:         write (T[nd], x)
5: else
6:     local m ← a + (b − a)/2
7:     local T[2 × nd] ← alloc_mod(int)
8:     local T[2 × nd + 1] ← alloc_mod(int)
9:     psac_par(BUILD(A, a, m, T, 2 × nd) ;
10:         BUILD(A, m, b, T, 2 × nd + 1))
11:     with read (T[2 × nd], T[2 × nd + 1]) as x, y do
12:         write (T[nd], x + y)
13: end if
```

The query function can be written in almost the same way as well:

Algorithm 5 Query function for subarray sum in the PSAC framework: obtain sum of $A[l:r]$ by calling QUERY$(A, l, r, 0, n, S, 1, res)$

, where res is a mod allocated outside the function.

1: **psac_function** QUERY(A : **int array**, l : **int**, r : **int**, a : **int**, b : **int**, S : **int array**, nd : **int**, res : **int mod**):
2: **if** $r \leq a$ **or** $b \leq l$ **then**
3:     **write** $(res, 0)$
4: **end if**
5: **if** $l \leq a$ **and** $b \leq r$ **then**
6:     **write** $(res, S[nd])$
7: **end if**
8: **local** $m \leftarrow a + (b-a)/2$
9: **local** $lres \leftarrow$ **alloc_mod**(**int**)
10: **local** $rres \leftarrow$ **alloc_mod**(**int**)
11: QUERY$(A, l, r, a, m, S, 2 \times nd, lres)$
12: QUERY$(A, l, r, m, b, S, 2 \times nd + 1, rres)$
13: **with read** $(lres, rres)$ **as** x, y **do**
14:     **write** $(res, x + y)$

Note that the PSAC framework only allows void-returning functions, so in order to compute these queries we must use destination-passing style as used above. Now there is no need for us to write the update function, since those will be handled by the PSAC framework automatically.

Now that we have the prerequisites out of the way, we can finally move on to more involved applications of this framework.

Chapter 2

Binary Search Trees

One key application of the PSAC framework is an implementation of binary search trees (BSTs). BSTs are a fundamental data structures which are synonymous with ordered dictionaries, and have other applications such as ordering data for databases [10]. They are also a great example for showing the power of the framework since they require dynamic allocation of mods inside the computation and also take great advantage of parallelism through operations such as map, filter, and reduce.

2.1 Structure of a Node

For the sake of simplicity, we shall assume for now that there is no granularity control, and that there is no self-balancing involved. Note that the data structure can be made balanced with high probability by adding a priority field to all nodes and appropriately changing the join function [13]. Data is stored at both the leaves and the internal nodes of the BST. Such a BST in a non-self-adjusting setting would only require 3 fields at any node: a value val, and pointers to the left and right children left and right. In order to make this structure self-adjusting there are two candidates (T is some arbitrary type for the value):

```
struct Node {
    val : T mod;
    left : Node* mod;
    right : Node* mod;
}
```

```
struct Node {
    val : T mod;
    left : Node mod*;
    right : Node mod*;
}
```

Since the PSAC framework is implemented in C++, the above mentioned pointers are C-style pointers: the null pointer is $0x0$ and equality can be tested using simple integer equality. Given this, should the left and right fields be mods of pointers to the children, or should they be pointers to mods which store nodes of children? Let's discuss the second structure: say the value of the left field changes for a node, initially it was a null pointer, but now it points to a mod storing some value. This is a change to the structure of the tree which needs to be propagated to the parent, but if we are using the second structure then this will not be possible. We can only read the mod stored in left if it is not a null pointer, so the read node responsible for the left child will

not exist in the first place. This problem can be fixed by using 'default mods' to represent null pointers so that the read already exists, but this is not ideal for two reasons:

1. It adds complexity to every BST function due to the logic for checking whether a mod represents a null pointer or not.
2. It increases the number of nodes allocated by approximately a factor of 2, increasing the amount of memory used as well as the time used.

On the other hand, with the first structure we can create leaves such that their left and right fields are mods of null pointers. If the left field now changes its value to a pointer to a node, then the change is immediately propagated since the reader now reads a different pointer. Therefore, for our implementation of self-adjusting BSTs we will use the first structure.

2.2 Basic operations: Split and Join

On a high level, given two trees L and R, and a value v such that v is greater than all values in L and less than all values in R, the JOIN operation returns the tree such that the value of its root is v, and its left and right subtrees are L and R respectively. Similarly, the JOIN2 operation takes two trees L and R such that all values in L are less than all values in R, and returns one tree T which contains exactly the values as the combined set of values of L and R. Note that this differs from JOIN because it does accept a middle value v, but still needs to assign a value to the root.

Given a tree T and a value v, the SPLIT operation splits the tree T at v: it returns whether v is present in T, and two trees L and R such that all values of L are less than v and all values in R are greater than v. The combined set of values in L and R is exactly same as the set of values in T if v is not present and also includes v if it was present.

JOIN is a primitive operation on a BST: any other operation, such as JOIN2, SPLIT, union, filter, reduce etc. can be written as an application of the JOIN function, without any knowledge of the balancing scheme used [14]. This is great because the self-adjusting join can be swapped out for other implementations using various self-balancing methods: no changes are required to the JOIN2, SPLIT, and other functions we present later when using different self-balancing methods. For simplicity, we use a trivial JOIN function in this section (with no self-balancing method).

Since the framework only allows us to write functions that have no return values, all functions need to written in destination-passing style. Therefore for the split function we need to define a new structure called SplitNode for storing the result of a split:

```
struct SplitNode {
    found : bool mod;
    left : Node* mod;
    right : Node* mod;
}
```

This helps us write the following join and split functions, for which the pseudocode is given below:

Algorithm 6 Join function for parallel self-adjusting BST

```
1: psac_function JOIN(l : Node*, r : Node*, v : T, res : Node*):
2: write (res.left, l)
3: write (res.right, r)
4: write (res.val, v)
```

Algorithm 7 Join2 function for parallel self-adjusting BST

```
1: psac_function JOIN2(l : Node*, r : Node*, res : Node*):
2: if l = ⊥ then
3:     if r = ⊥ then
4:         res ← ⊥
5:     else
6:         with read (r.left, r.right, r.val) as rl, rr, rv do
7:             JOIN(rl, rr, rv, res)
8:     end if
9: else
10:    local rres ← alloc(Node)
11:    with read (l.right) as lr do
12:        JOIN2(lr, r, rres)
13:    with read (l.left, l.val) as ll, lv do
14:        JOIN(ll, lv, rres, res)
15: end if
```

First note that JOIN is simply a $O(1)$ time function without any self-balancing (it is $O(\log n)$ if it is implemented with the self-balancing schemes of AVL trees or Red-Black trees for example, and $O(\log n)$ with high probability for treaps). For JOIN2, we recursively go down the right spine of left tree: if the length of the spine is r then the function does $O(r)$ work. For SPLIT, given that the height of the input tree is h, it can be easily seen that it does $O(h)$ work.

Note that the code should be written in such a way that the reads are as delayed as possible. For example, instead of reading all 3 value of the node in line 3 of SPLIT, we only read the val. This way we avoid doing some unnecessary work in the event that the left or right field of the node change. In particular, if the $v < val$ case is true, and only right changes, then we do not need to make the recursive call to SPLIT. Similarly, in JOIN2, we only read l.right on line 4 while calling JOIN2 recursively, and we defer the read for l.left and l.val for the $O(1)$ function JOIN.

Note that all of these functions are purely sequential and do not use parallelism in any way. But now that we have them out of the way, we can discuss filter and mapreduce, functions that do benefit from parallelism.

Algorithm 8 Split function for parallel self-adjusting BST

```
1:  psac_function SPLIT(v : T, nd : Node*, res : SplitNode):
2:  if nd = ⊥ then
3:      write (res.found, false)
4:      write (res.left, ⊥)
5:      write (res.right, ⊥)
6:  else
7:      with read (nd.val) as val do
8:          if val = v then
9:              write (res.found, true)
10:             with read (nd.left) as l do
11:                 write (res.left, l)
12:             with read (nd.right) as r do
13:                 write (res.right, r)
14:         else if v < val then
15:             local s ← alloc(SplitNode)
16:             with read (nd.left) as l do
17:                 SPLIT(v, l, s)
18:             with read (s.left, s.right, s.found, nd.right) as cl, cr, f, r do
19:                 write (res.left, cl)
20:                 write (res.found, f)
21:                 JOIN(cr, r, val, res.right)
22:         else
23:             local s ← alloc(SplitNode)
24:             with read (nd.right) as r do
25:                 SPLIT(v, r, s)
26:             with read (s.left, s.right, s.found, nd.left) as cl, cr, f, l do
27:                 write (res.right, cr)
28:                 write (res.found, f)
29:                 JOIN(l, cl, val, res.left)
30:         end if
31: end if
```

2.3 Filter and Mapreduce

Filter and Mapreduce are common applications of binary search trees. Filter lets the user specify a predicate function and obtain a new BST which only contains the elements of the input BST that satisfy the predicate. For example, the values can be all integers, and the filter function only keeps the ones that are even. Mapreduce lets the user specify two functions: a map function and a reduce function. The map function is any function that is applied to all the elements of the BST, the reduce function then combines all mapped values returning one value. For example, the values can be integers, the mapping function can obtain their last two digits, and then the reduce function can return the maximum value. Mapreduce is a very common tool in big data [16], and therefore there is a need of having an efficient parallel self-adjusting mapreduce.

2.3.1 Filter Implementation

Algorithm 9 Filter function for parallel self-adjusting BST

```
1: psac_function FILTER(nd : Node*, res : Node* mod, f : T ↦ bool):
2: if nd = ⊥ then
3:     write (res, ⊥)
4: else
5:     local lres ← alloc_mod(Node*)
6:     local rres ← alloc_mod(Node*)
7:     psac_par(with read (nd.left) as l do FILTER(l, lres, f);
8:         with read (nd.right) as r do FILTER(r, rres, f))
9:     local y ← alloc(Node)
10:    with read (lres, rres, nd.val) as l, r, v do
11:        if f(v) then
12:            JOIN(l, r, v, y)
13:        else
14:            JOIN2(l, r, y)
15:        end if
16:    write (res, y)
17: end if
```

In the above implementation, since y is a pointer to a Node, the write to res does not need to be within the read and does not need to be updated during change propagation. By simply updating the fields of y itself using JOIN and JOIN2 we can update the values in res.

Say that the input tree has height h and the total number of nodes is n. Assume for the sake of simplicity that the tree is perfectly balanced, that is it is complete binary tree and we have $h = O(\log n)$. Also, assuming that the function f can be computed in $O(1)$ time for any input, the total work and span at any single call to FILTER is $O(\log n)$. Therefore the work and span

recurrences of the algorithm are

$$W(n) = 2W(n/2) + O(\log n)$$

$$S(n) = S(n/2) + O(\log n)$$

The work recurrence is leaf-dominated, therefore the total asymptotic work is the total work at the leaves, which is $O(n)$. The span recurrence on the other hand, is balanced, and the total span is $O(\log^2 n)$.

2.3.2 Mapreduce Implementation

Note that in the parallel setting we pose a key restriction on the reduce function: it must be *associative*. That is, if the reduce function is $f : U \times U \mapsto U$ and $x, y, z : U$ are any inputs then we must have

$$f(f(x, y), z) = f(x, f(y, z))$$

In other words, the order in which we apply the function f to the inputs must not change the final value obtained. Moreover, we also need an identity $b : U$ such that for any $x : U$ we have

$$f(x, b) = f(b, x) = x$$

Furthermore, instead of just maintaining the overall reduced value of the entire tree, we can also store reduced values of subtrees. Due to the destination-passing style of all functions, just like a SplitNode we must also introduce a ReduceNode which is defined as following:

```
struct ReduceNode {
    val : U;
    left : ReduceNode*;
    right : ReduceNode*;
}
```

Note that the ReduceNode does not need to have fields with mods because it is only used for storing the result, and therefore changes to it do not need to be propagated.

Algorithm 10 Mapreduce function for parallel self-adjusting BST

```
1: psac_function MAPREDUCE(nd : Node*, res : ReduceNode*, m : T ↦ U, r : U × U ↦
   U, b : U):
2: if nd = ⊥ then
3:     write (res.val, b)
4:     write (res.left, ⊥)
5:     write (res.right, ⊥)
6: else
7:     local lres ← alloc(ReduceNode)
8:     local rres ← alloc(ReduceNode)
9:     psac_par(with read (nd.left) as l do MAPREDUCE(l, lres, m, r, b);
10:        with read (nd.right) as r do MAPREDUCE(r, rres, m, r, b))
11:    write (res.left, lres)
12:    write (res.right, rres)
13:    with read (lres.val, rres.val, nd.val) as lv, rv, v do
14:        local nv ← r(r(lv, m(v)), rv)
15:        write (res.val, nv)
16: end if
```

Similar to the implementation of FILTER, the above implementation also declares local variables $lres$ and $rres$, which are written to result outside the read. The pointers stored at res.left and res.right are constant but the Nodes that they point to are updated through change propagation.

For calculating the work and span of Mapreduce we will make the same assumptions as Filter: the input tree has height h and the total number of nodes is n, the tree is perfectly balanced, and both functions m, r can be computed in $O(1)$ time for any input. The total work and span at any single call to MAPREDUCE is $O(1)$. Therefore the work and span recurrences of the algorithm are

$$W(n) = 2W(n/2) + O(1)$$
$$S(n) = S(n/2) + O(1)$$

The work recurrence is leaf-dominated, therefore the total asymptotic work is the total work at the leaves, which is $O(n)$. The span recurrence on the other hand, is balanced, and the total span is $O(\log n)$.

2.4 Computation Distance for Batch Inserts

Batch inserts allow the user to specify an array of values, and insert them all into the BST in one go. Inserting values to the BST one by one does not lend itself to any parallelism, and therefore batch inserts are essential in order to demonstrate the benefits of parallel self-adjusting computation.

For the sake of simplicity, we will assume that inserts are made in such a way that the tree remains balanced with high probability (this holds, for instance, if the inserts are made uniformly randomly). Moreover, all inserts are made at the leaves. We will also assume that the initial tree has a height of $O(\log n)$ and the mapping, filtering, and reducing functions all have $O(1)$ cost. The pseudocode for batch inserts is given below for clarity about its implementation. Note that this function is not written in the self-adjusting framework: it is not a PSAC function since it is the external update to the self-adjusting data structure and therefore does not need any of the PSAC primitives other than **write**. We still need to use **write** since it not only updates the value stored in the mod, but it also marks the appropriate readers for change propagation. After making this external change, we need to call the **propagate** primitive in order to update the data structures and the queries. We do not need **read** however, and the values at mods can be accessed by accessing their value field.

Algorithm 11 Batch Inserts for parallel self-adjusting BSTs

```
1:  psac_function BATCH_INSERT(nd : Node mod, vals : T Vector, l : int, r : int):
2:  if l = r then
3:      return
4:  end if
5:  if nd.value = ⊥ then
6:      MAKETREE(nd, vals, l, r)
7:  else
8:      local m1 ← FINDCROSSOVERPOINT(nd.value.val, vals, l, r)
9:      local m2 ← nd.value.val = vals[m1]?m1 + 1 : m1
10:     par(BATCH_INSERT(nd.value.left, vals, l, m1);
11:         BATCH_INSERT(nd.value.right, vals, m2, r))
12: end if
```

Note that the above pseudocode assumes that $vals$ are sorted. The MAKETREE function creates a tree from scratch on the values of $vals[l : r]$ and outputs it to nd. The FINDCROSSOVERPOINT function is responsible for finding the smallest integer in $vals$ which is greater than or equal to nd.value.val. If the value begin inserted is already present in the tree at nd.value.val, then it is not inserted again and is skipped.

Theorem 2.4.1. *Consider a run of Algorithm 9: Filter on an input of size of n, and a batch update of k uniformly random inserts. The computation distance induced by such an update is $O\left(k \log n \log\left(1 + \frac{n}{k}\right)\right)$.*

Proof. Let's first define an **affected filter call**: after an update is made, an affected filter call is a call to the FILTER function where there exists an affected read node. This includes the affected read nodes in the call made to JOIN or JOIN2 by that FILTER call. Also, note that JOIN2 has $O(\log n)$ work in the worst case, and if JOIN were implemented with a self-balancing scheme such as the one for AVL trees, it would have $O(\log n)$ work as well.

We can count the number of affected filter calls in the first $\log k$ layers of the BST, separately from the affected filter calls in the other layers. Since every node has at most 2 children, the total number of calls to JOIN2 in the first $\log k$ layers is $O(2^{\log k}) = O(k)$. Note that this is an upper bound for the affected calls of JOIN2 in the first $\log k$ layers.

Now, for the remaining $O(\log(n/k))$ layers of the tree: first assume there is only one element inserted in the tree, and say it gets inserted as the left child C of current leaf L. Every call to filter made on the path from C to the root is affected, because the calls to JOIN and JOIN2 are going to have an affected reader. For example, JOIN2 as described in Algorithm 7, recursively joins together its two trees. Since the value of the mod containing the pointer to the left child of L now holds C instead of a null pointer, this might affect all of these calls to JOIN2. Moreover, note that a node that does not lie on the path from C to the root cannot possibly have an affected reader, since every function call to FILTER is agnostic to the tree outside the subtree rooted at nd. Therefore for 1 inserted value, in the bottom $O(\log(n/k))$ layers of the tree, there are $O(\log(n/k))$ affected filter calls. If there are k elements inserted then the total number of affected filter calls is $O(k \log(n/k))$.

Therefore the total number of affected filter calls is $O(k + k\log(n/k))$, and each of these calls can do $O(\log n)$ work in the worst case. Therefore, the total computation distance for a batch insert of k elements is $O(k \log n(1 + \log(n/k))) = O(k \log n(\log(1 + n/k)))$. □

We also have a similar proof for the computation distance of MAPREDUCE:

Theorem 2.4.2. *Consider a run of Algorithm 11: Mapreduce on an input of size of n, and a batch update of k uniformly random inserts. The computation distance induced by such an update is $O\left(k \log\left(1 + \frac{n}{k}\right)\right)$.*

Proof. Analogous to the previous proof, we define an **affected mapreduce call**: after an update is made, an affected mapreduce call is a call to the MAPREDUCE function where there exists an affected read node.

We can count the number of affected reads of MAPREDUCE in the first $\log k$ layers of the BST, separately from the affected reads in the other layers. Since every node has at most 2 children, the total number of reads in the first $\log k$ layers is $O(2^{\log k}) = O(k)$. Note that this is an upper bound for the affected reads of MAPREDUCE in the first $\log k$ layers.

Now, for the remaining $O(\log(n/k))$ layers of the tree: first assume there is only one element inserted in the tree, and say it gets inserted as the left child C of current leaf L. Every call to mapreduce made on the path from C to the root is affected, because if the val field of a node's child is modified, then that node is going to have an affected reader (line 13 in 11). Moreover, similar to the proof for FILTER we note that MAPREDUCE is agnostic to the tree outside the subtree rooted at nd, and therefore there cannot be any affected readers outside the filter calls on the path from C to the root. Therefore for 1 inserted value, in the bottom $O(\log(n/k))$ layers of the tree, there are $O(\log(n/k))$ affected mapreduce calls. If there are k elements inserted then

the total number of affected mapreduce calls is $O(k \log(n/k))$.

Each affected mapreduce call does $O(1)$ work, so the total computation distance for a batch insert of k elements is $O(k(1 + \log(n/k))) = O(k \log(1 + n/k))$. □

2.5 Granularity Control

In order to make the above implementation of a parallel self-adjusting BST more practically efficient, we need to add granularity control. Instead of only storing one value at any leaf, we can store a vector of values. This way, most of the values in the BST are stored in the leaves where we can utilize caching in order to write more efficient implementations of the algorithms we have discussed.

Adding granularity control adds a lot of implementation complexity for algorithms such as SPLIT, FILTER etc. We will not be going through detailed implementations in this report, but they can be found in the Github repository [1].

In this section, we will discuss the structure of the node in this implementation. There are two types of nodes now: internal nodes and leaves. Internal nodes need to store a single value, and two pointers to the left and right children. Leaves only need to a vector of values that are stored there. Let's say the internal nodes are represented by the structure INode and the leaves are represented by the structure Leaf. We also introduce a wrapper around these two structures NodePtr, which holds either a pointer to an INode or a pointer to a Leaf. These structures are defined as follows:

```
struct INode {
    left : NodePtr mod;
    val : T mod;
    right : NodePtr mod;
};

struct Leaf {
    arr : T Vector mod;
};

struct NodePtr {
    union {
        leaf : Leaf*;
        inode : INode*;
    }
};
```

A vector in the above implementation refers to the Vector in the Standard Template Library of C++ [25]: a variable-sized array-like container with random access. Note that in order to fully utilize caching, instead of storing a vector of mods, we store a single mod of the entire vector: even if one value is changed, the entire vector is replaced and change propagation kicks in. Therefore, not only does this method enforce granularity control for parallelism, but also for

[1] https://github.com/cmuparlay/psac/include/psac/examples/bst.hpp

change propagation.

In order to differentiate between a pointer to a leaf versus a pointer to an internal node, we can steal the last bit of the pointer: since the last 3 bits of a valid pointer value are always 0, we can make the last bit 1 if and only if it is a leaf to indicate that it is a leaf. This is a useful technique since it allows us to keep track of the type of pointer stored in a NodePtr without using any extra space.

2.6 Benchmarks

n	k	Seq	Parallel Static				PSAC Compute				PSAC Update					
			1	20	20ht	SU	1	20	20ht	SU	1	20	20ht	SU	WS	T
10^6	10^0	19ms	16ms	1.54ms	1.29ms	12.77	20ms	2.34ms	1.85ms	11.22	9us	27us	26us	0.35	2.19k	760.0
10^6	10^1	-	-	-	-	-	-	-	-	-	92us	54us	53us	1.74	215.1	374.6
10^6	10^2	-	-	-	-	-	-	-	-	-	751us	261us	181us	4.15	26.46	109.8
10^6	10^3	-	-	-	-	-	-	-	-	-	5.58ms	712us	587us	9.50	3.56	33.86
10^6	10^4	-	-	-	-	-	-	-	-	-	34ms	4.30ms	3.38ms	10.31	0.57	5.88
10^6	10^5	-	-	-	-	-	-	-	-	-	78ms	23ms	19ms	4.06	0.25	1.03
10^6	10^6	-	-	-	-	-	-	-	-	-	282ms	43ms	55ms	6.46	0.07	0.45

Table 2.1: Filter Benchmark

n	k	Seq	Parallel Static				PSAC Compute				PSAC Update					
			1	20	20ht	SU	1	20	20ht	SU	1	20	20ht	SU	WS	T
10^5	10^0	59ms	42ms	4.50ms	2.67ms	16.12	43ms	4.21ms	2.69ms	16.14	4us	15us	17us	0.31	11.97k	3.74k
10^5	10^1	-	-	-	-	-	-	-	-	-	42us	32us	49us	1.31	1.40k	1.84k
10^5	10^2	-	-	-	-	-	-	-	-	-	1.34ms	187us	160us	8.36	44.13	368.9
10^5	10^3	-	-	-	-	-	-	-	-	-	13ms	1.49ms	896us	15.35	4.30	66.07
10^5	10^4	-	-	-	-	-	-	-	-	-	57ms	13ms	8.14ms	7.04	1.03	7.28
10^5	10^5	-	-	-	-	-	-	-	-	-	246ms	48ms	45ms	5.43	0.24	1.30

Table 2.2: Mapreduce Benchmark

The above tables represent numerical benchmark results for Filter and Mapreduce:

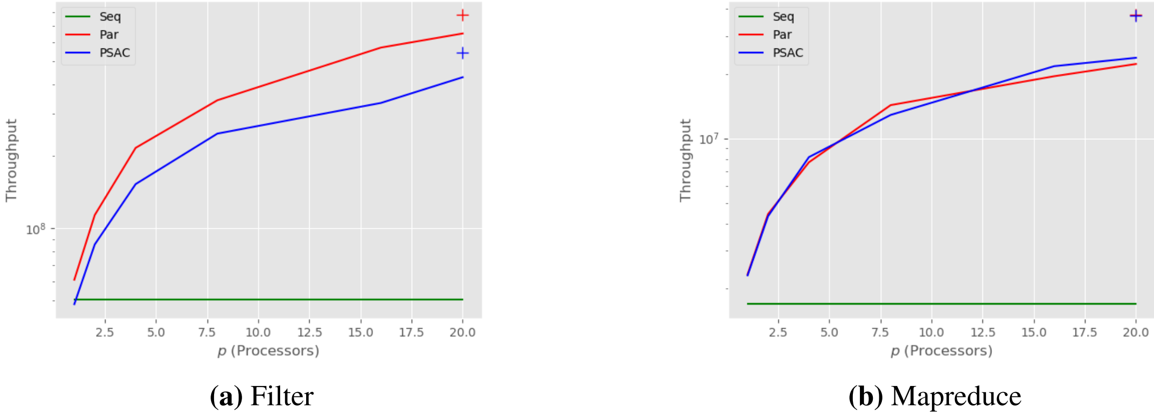
- The Filter benchmark is executed on an initial BST of size 10^6 with an $O(1)$ filter function which checks if the value is even or odd.
- The Mapreduce benchmark is executed on an initial BST of size 10^5: the mapping function maps the value of the value (a string) to its edit distance from a fixed string, and the reducing function obtains the minimum. The mapping function takes $O(L^2)$ time where L is the length of the strings. $L = 10$ is used for the benchmark.

A granularity control of 128 was used for both benchmarks.

SU is the relative speedup of the 20ht benchmark compared to the corresponding 1 core benchmark. WS denotes the work-savings, i.e. the relative speedup of the 1 core dynamic benchmark compared to the static sequential benchmark. Total is the total speedup, i.e. the relative speedup of the 20ht benchmark compared to the static sequential algorithm (equivalently, the product of SU and WS). The benchmarks were executed on the CMU Parallel cluster with

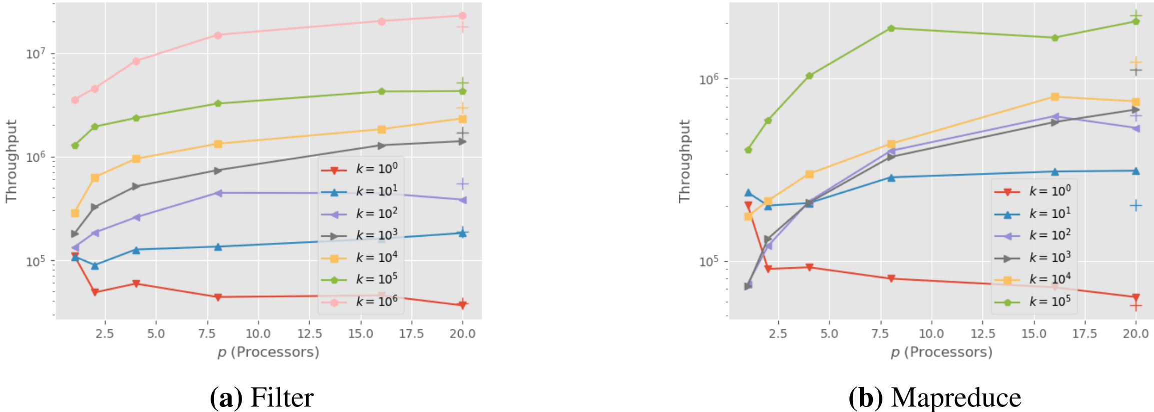
20 cores (and 40 threads with hyperthreading). We used the Google Benchmark C++ library to measure the speed (in real/wall time) of each benchmark. We run each benchmark ten times and take the average running time.

The benchmarks can be better summarized with the following graphs (the + symbols indicate the performance of hyperthreading):



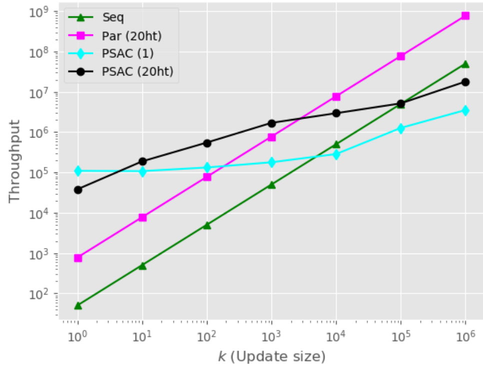
(a) Filter **(b)** Mapreduce

Figure 2.1: Absolute throughput (input updates per second) vs number of cores (initial computation)

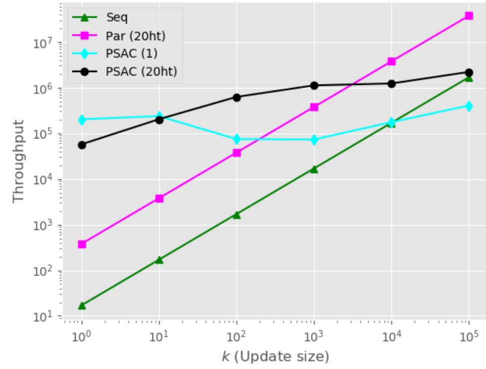


(a) Filter **(b)** Mapreduce

Figure 2.2: Absolute throughput (input updates per second) vs number of cores (updates)

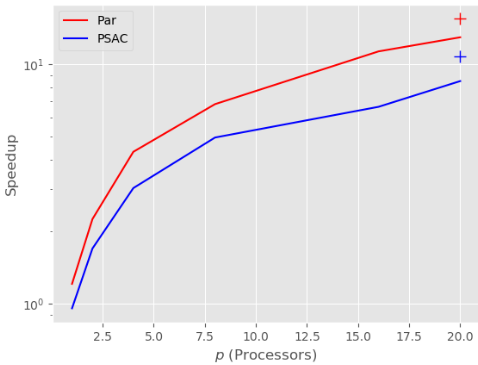


(a) Filter

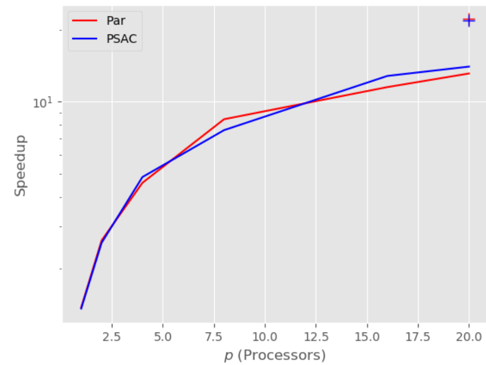


(b) Mapreduce

Figure 2.3: Absolute throughput (input updates per second) vs update size

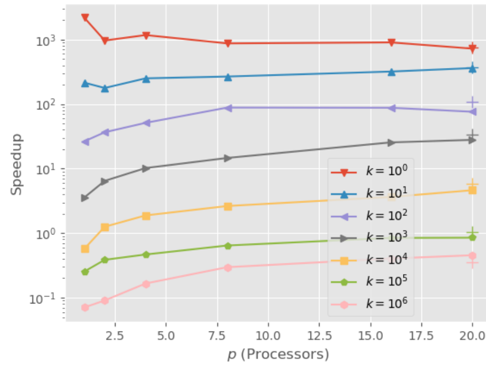


(a) Filter

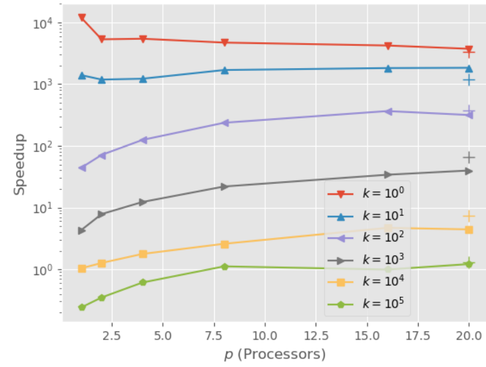


(b) Mapreduce

Figure 2.4: Relative Speedup with respect to sequential baseline vs number of cores (initial computation)

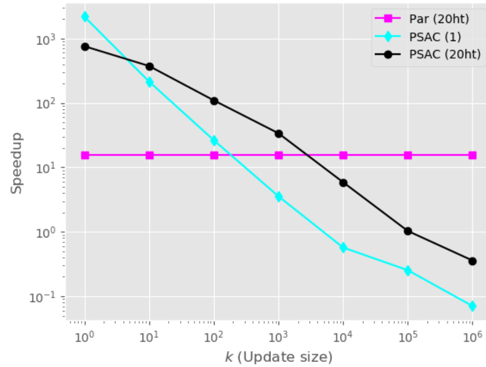


(a) Filter

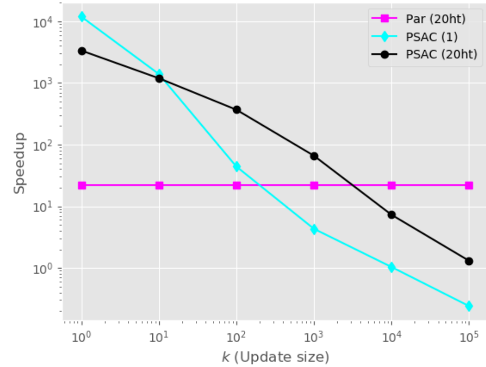


(b) Mapreduce

Figure 2.5: Relative Speedup with respect to sequential baseline vs number of cores (updates)



(a) Filter



(b) Mapreduce

Figure 2.6: Relative Speedup with respect to sequential baseline vs update size

Figure 2.6 shows that PSAC provides a significant benefit over non-self-adjusting implementations practically when it comes to smaller updates: the cross over point for filter and mapreduce occur when the update size is between 10^3 and 10^4: for updates larger than that it is more efficient to just build the BST from scratch but for smaller updates PSAC is very effective.

Chapter 3

Dynamic Convex Hull

Given a set of points in $\mathbb{R}^d$, a **convex hull** is defined as the smallest convex set that contains all of those points. In $\mathbb{R}^2$, there are various efficient algorithms such as Graham scan [18], quick hull [8], and Overmars' algorithm [24]. Given n points in the 2D plane, these algorithms can find a convex hull with $O(n \log n)$ work. For this chapter, we will represent points in the 2D plane by the tuple (x, y).

In this chapter, we will explore an implementation of Overmars' algorithm [24]: a dynamic convex hull algorithm. Not only does the algorithm support creation of a convex hull in $O(n \log n)$ work, but it can also support insertions and deletions in $O(\log^2 n)$ work. Moreover, it is a divide and conquer algorithm and therefore lends itself to good parallelism and stability, making it an ideal candidate for parallel-self-adjusting computation. Note that we will not discuss the update functions of Overmars' algorithm: we won't be needing them as an implementation of the initial creation of the convex hull in the self-adjusting framework is enough to efficiently handle updates automatically.

3.1 Overview of Overmars' Algorithm [24]

The algorithm first makes the following key observation: instead of maintaining the entire convex hull as one set, we can divide it into two halves: a left convex hull (LC-hull) and a right convex hull (RC-hull). If the set of points is P, then the LC-hull of P is the convex hull of $P \cup \{(\infty, 0)\}$. Similarly, the RC-hull of P is the convex hull of $P \cup \{(-\infty, 0)\}$. That is, these convex hulls contain all the points in P but also go out to infinity along the positive and negative x-axis respectively. In order to create the actual convex hull, we can easily 'stitch' together these two hulls, so we only need to without loss of generality, we only need to worry about maintaining the LC-hull.

The broad idea of the algorithm is as follows:

1. Sort the points by their y-coordinate.
2. Divide the set of points into 2 halves with equal number of points based on their y-coordinate.

3. Recursively compute the LC-hulls of the 2 halves and maintain the following information:
   (a) The point with the smallest y-coordinate.
   (b) The left and right end points of the *bridge* (defined later).
4. Given the two LC-hulls L and R of the two halves, perform binary search (again, according to their y-coordinate) and find a point a in L and b in R such that adding a line segment between a and b, removing the hull in L above a and removing the hull in R below b, creates a new LC-hull for the entire set of points. This entire step can be thought of as finding a suitable common 'tangent' of the two LC-hulls. These two points a and b are the left and right end points for the *bridge* of this LC-hull, and these must be returned by every recursive call.

We will not go into the details of how the original algorithm implements insertions and deletions, since those will be automatically supported in our self-adjusting framework. However the reader is highly encouraged to read Section 4 of [24] to appreciate the beauty of the algorithm.

3.2 Using Binary Search Trees

Note that the most involved step of the above algorithm is computing the bridge, using the bridge points of the two children in the recursion tree. For the sake of simplicity, we will assume that all points have non-negative coordinates.

One reason why this is example is interesting is because we get to use our BST implementation from the previous chapter. Before we discuss the implementation, we need to define a few structures:

```
struct Point {
    x : int;       // x-coordinate of point
    y : int;       // y-coordinate of point
};
struct NodeInfo {
    p : Point;       // the point
    bl : Point;      // the left bridge point
    br : Point;      // the right bridge point
    low : Point;     // the point with the smallest y-coordinate
};
```

The algorithm can be construed as a Map operation on an input BST of Points stored in the val field, producing an output BST of NodeInfos stored in the val field. The input BST is ordered by the y-coordinate of the points, and the output BST is ordered by the y-coordinate of the field p. For notational convenience, we can define the nodes in the input tree as InputNodes and the nodes in the output tree as OutputNodes.

However, there is one small issue: the divide and conquer strategy of Overmars' algorithm divides the set of points into 2, but the BST also stores a points at internal nodes, not just at leaves. This can be resolved in two ways:

1. Move all the real values to the leaves, and have dummy points on internal nodes.
2. While computing the bridge at any internal node, we can treat the Point p at the internal node as a singleton tree: we first find the bridge between the points in the left subtree left and p resulting in a LC-hull of the points in left and p, and then combine that resulting LC-hull with the LC-hull of the points in right.

Both of the above approaches are valid, but we will use the latter. Also, note that we will present the version of the algorithm which uses the BST without granularity control. The implementation with granularity control can be found in the Github repository [1].

3.3 Implementation

We initialize the bridges by calling the below INITBRIDGE function on the root of the input tree:

Algorithm 12 Initialize Bridges Recursively for the LC-hulls

```
1:  psac_function InitBridge(nd : InputNode, res : OutputNode mod):
2:  if nd = ⊥ then
3:      write (res, ⊥)
4:  else if nd.left = ⊥ ∧ nd.right = ⊥ then
5:      local x ← alloc(OutputNode)
6:      with read (nd.val) as p do
7:          x.val.p, x.val.bl, x.val.br, x.val.low ← p
8:          write (res, x)
9:  else
10:     local lres ← alloc_mod(OutputNode)
11:     local rres ← alloc_mod(OutputNode)
12:     psac_par(with read (nd.left) as l do InitBridge(l, lres);
13:         with read (nd.right) as r do InitBridge(r, rres))
14:     with read (nd.val, lres, rres) as p, l, r do
15:         local s ← alloc(OutputNode)
16:         s.val.p, s.val.bl, s.val.br, s.val.low ← p
17:         if l = ⊥ then
18:             BridgeHelper(s, r, p, p, r.val.low, res)
19:         else if r = ⊥ then
20:             with read (l.val) as lv do
21:                 BridgeHelper(l, s, lv.low, p, s.val.low, res)
```

[1] https://github.com/cmuparlay/psac/include/psac/examples/convex-hull.hpp

```
22:         else
23:             local tmp ← alloc_mod(OutputNode)
24:             with read (l.val) as lv do
25:                 BridgeHelper(l, s, lv.low, p, s.val.low, tmp)
26:                 with read (tmp) as l2 do
27:                     BridgeHelper(l2, r, lv.low, p, r.val.low, res)
28:         end if
29: end if
```

Lines 2-8 in the above pseudocode deal with the base cases: if the node is a leaf and both of its children are null pointers and contains a point p, then both the left and right "bridge" points of the subtree rooted at the node are simply p. For the recursive case, we first obtain the bridge points of the subtrees rooted at the left and right children of the node in parallel. Lines 14-28 then deal with computing the bridge of the current node: as mentioned earlier if both the left and right subtrees are non-null, then we need to call the BRIDGEHELPER function twice: first to combine the singleton tree containing p with the left subtree l, and then combining the result with the right subtree r. The six arguments to BRIDGEHELPER are intuitively as follows:

1. l : OutputNode. The left tree on which binary search is performed.
2. r : OutputNode. The right tree on which binary search is performed.
3. low : Point. The point with the smallest y-coordinate in the subtree for which the bridge initially needed to be created.
4. p : Point. The point stored at the root of the subtree for which the bridge initially needed to be created.
5. y : **int**. The smallest y-coordinate of any point in the right subtree of the original call.
6. res : OutputNode **mod**. Stores the resulting bridge.

Before we discuss the implementation of BRIDGEHELPER, we also need to need a helper function cross: Point $\times$ Point $\times$ Point $\mapsto$ **int**, which computes the cross product between three points. The cross product is positive if the 3 points are oriented anti-clockwise, 0 if they are co-linear, and negative if the 3 points are oriented clockwise. Knowing the orientation of points is critical for determining how the binary search proceeds. The y : **int** argument in the BRIDGE-HELPER function is also useful for the same purpose.

Algorithm 13 Perform a simultaneous binary search on two LC-hulls to find the appropriate bridge

```
1: psac_function BridgeHelper(l : OutputNode, r : OutputNode, low : Point, p : Point, y :
   int, res : OutputNode mod):
2: with read (l.val, r.val) as lv, rv do
3:     local a ← lv.bl
4:     local b ← lv.br
5:     local c ← rv.bl
6:     local d ← rv.br
```

```
7:      if a = b ∧ c = d then
8:          local x ← alloc(OutputNode)
9:          x.p ← p
10:         x.bl ← a
11:         x.br ← c
12:         x.low ← low
13:         write (res, x)
14:     else if a ≠ b ∧ cross(a, b, c) > 0 then
15:         with read (l.left) as lcl do
16:             BridgeHelper(lcl, r, low, p, y, res)
17:     else if c ≠ d ∧ cross(b, c, d) > 0 then
18:         with read (r.right) as rcr do
19:             BridgeHelper(l, rcr, low, p, y, res)
20:     else if a = b then
21:         with read (r.left) as rcl do
22:             BridgeHelper(l, rcl, low, p, y, res)
23:     else if c = d then
24:         with read (l.right) as lcr do
25:             BridgeHelper(lcr, r, low, p, y, res)
26:     else
27:         local s1 ← cross(a, b, c)
28:         local s2 ← cross(b, a, d)
29:         if s1 + s2 = 0 ∨ s1 × d.y + s2 × c.y < y × (s1 + s2) then
30:             with read (l.right) as lcr do
31:                 BridgeHelper(lcr, r, low, p, y, res)
32:         else
33:             with read (r.left) as rcl do
34:                 BridgeHelper(l, rcl, low, p, y, res)
35:         end if
36:     end if
```

The above BridgeHelper method performs a simultaneous binary search on l and r until both of them reach a leaf node. If and only if the leaf node is reached will the left and right bridge points of a node be equal, so we can use that equality as a test of whether we are at a leaf. Also note that in the above implementation low, p, y are all unchanged through all recursive calls. The details of why we move down the left or right subtrees of the left or right tree in the above specific cases are discussed by Overmars and Van Leeuwen [24].

3.4 Computation Distance for Convex Hull

We will assume that the Input and Output BSTs produced by the convex hull algorithm are balanced and have a height of $O(\log n)$ where n is the total number of points. First note that Algorithm 13: BRIDGEHELPER has a work and span of $O(\log n)$. Just like the previous chapter on BSTs, we will assume that all inserts are made on the leaves, and are uniformly random so that the trees stay balanced.

Theorem 3.4.1. *Consider a run of Algorithm 12:* INITBRIDGE *on an input of size of n, and a batch update of k uniformly random inserts. The computation distance induced by such an update is $O\left(k \log n \log\left(1 + \frac{n}{k}\right)\right)$.*

Proof. Analogous to the proofs in the BST chapter, we define an **affected InitBridge call**: after an update is made, an affected InitBridge call is a call to the INITBRIDGE function where there exists an affected read node. This includes possible affected read nodes in the calls to BRIDGE-HELPER.

We can count the number of affected reads of INITBRIDGE in the first $\log k$ layers of the BST, separately from the affected reads in the other layers. Since every node has at most 2 children, the total number of reads in the first $\log k$ layers is $O(2^{\log k}) = O(k)$. Note that this is an upper bound for the number of affected reads of INITBRIDGE in the first $\log k$ layers.

Now, for the remaining $O(\log(n/k))$ layers of the tree: first assume there is only one element inserted in the tree, and say it gets inserted as the left child C of current leaf L. Every call to INITBRIDGE made on the path from C to the root is affected, because every one of those nodes calls BRIDGEHELPER, and they each have an affected reader since their binary search eventually needs to read the left child of L, which has now changed. Moreover, similar to the proof for FILTER we note that INITBRIDGE is agnostic to the tree outside the subtree rooted at nd, and therefore there cannot be any affected readers outside the INITBRIDGE calls on the path from C to the root. Therefore for 1 inserted value, in the bottom $O(\log(n/k))$ layers of the tree, there are $O(\log(n/k))$ affected INITBRIDGE calls. If there are k elements inserted then the total number of affected InitBridge calls is $O(k \log(n/k))$.

Each affected InitBridge call does $O(\log n)$ work, so the total computation distance for a batch insert of k elements is $O(k \log n(1 + \log(n/k))) = O(k \log n \log(1 + n/k))$. □

3.5 Granularity Control

In order to get good practical performance, we must implement granularity control in the above algorithm so that we can take full advantage of the cache. As stated in the previous chapter, binary search trees are implemented using vectors at leaves: instead of storing one single value at leaves we store multiple values in a mod of a vector. This requires us to change the implementation of the bridge finding algorithm above a bit, since we need to implement binary search on these vectors as well. This leads to quite a strange implementation since we are performing

binary search simultaneously on two trees, and we also need to account for the case where we are at an internal node in one of the trees, and at a leaf in the other. We will not discuss the entire implementation of the algorithm since it is quite long and tedious (and can be found in the Github repository), but Algorithm 14 discusses one of the cases of the bridge finding algorithm: when the $left$ node is a leaf and the $right$ node is an internal node (the other cases can be easily extrapolated). Note that the InputNode and OutputNode types are now analogues of the NodePtr structure discussed in section 2.5, and therefore are wrappers around an internal node structure (with a single value and two children), and a leaf structure (with just a vector of values).

Algorithm 14 One case of BridgeHelper (Algorithm 13) with granularity control: when $left$ is a leaf, and $right$ is an internal node.

1: **psac_function** BRIDGEHELPERGRANCONTROL($left$: OutputNode, $right$: OutputNode, ll : **int**, lr : **int**, rl : **int**, rr : **int**, low : Point, p : Point, y : **int**, res : OutputNode **mod**):
2: **if** $left$.is_leaf $\wedge$ **not** $right$.is_leaf **then**
3:     **with read** ($left$.arr, $right$.val) **as** $larr, rv$ **do**
4:         **local** $lmid \leftarrow ll + (lr - ll)/2$
5:         **local** $a \leftarrow larr[lmid]$.bl
6:         **local** $b \leftarrow larr[lmid]$.br
7:         **local** $c \leftarrow rv$.bl
8:         **local** $d \leftarrow rv$.br
9:         **if** $a = b \wedge c = d$ **then**
10:             **local** $x \leftarrow$ **alloc**(OutputNode)
11:             x.p $\leftarrow p$
12:             x.bl $\leftarrow a$
13:             x.br $\leftarrow c$
14:             x.low $\leftarrow low$
15:             **write** (res, x)
16:         **else if** $a \neq b \wedge$ cross$(a, b, c) > 0$ **then**
17:             BRIDGEHELPER($left, right, ll, lmid, rl, rr, low, p, y, res$)
18:         **else if** $c \neq d \wedge$ cross$(b, c, d) > 0$ **then**
19:             **with read** ($right$.right) **as** rcr **do**
20:                 BRIDGEHELPER($left, rcr, ll, lr, rl, rr, low, p, y, res$)
21:         **else if** $a = b$ **then**
22:             **with read** ($right$.left) **as** rcl **do**
23:                 BRIDGEHELPER($left, rcl, ll, lr, rl, rr, low, p, y, res$)
24:         **else if** $c = d$ **then**
25:             BRIDGEHELPER($left, right, lmid + 1, lr, rl, rr, low, p, y, res$)

```
26:         else
27:             local s1 ← cross(a, b, c)
28:             local s2 ← cross(b, a, d)
29:             if s1 + s2 = 0 ∨ s1 × d.y + s2 × c.y < y × (s1 + s2) then
30:                 BridgeHelper(left, right, lmid + 1, lr, rl, rr, low, p, y, res)
31:             else
32:                 with read (right.left) as rcl do
33:                     BridgeHelper(left, rcl, ll, lr, rl, rr, low, p, y, res)
34:             end if
35:         end if
36: end if
```

Note that there are other caveats of the implementation: the result res need not be an internal node, and we need a separate function for when it is a leaf. Moreover, the above implementation does not initialize the range values when we hit a leaf. For instance, in the $a = b$ case, we must case on whether rcl is a leaf or not, and if it is we need to read rcl.arr as a, and pass 0 and **len**(a) to the recursive call instead of rl and rr. This is omitted in the code for clarity.

3.6 Benchmarks

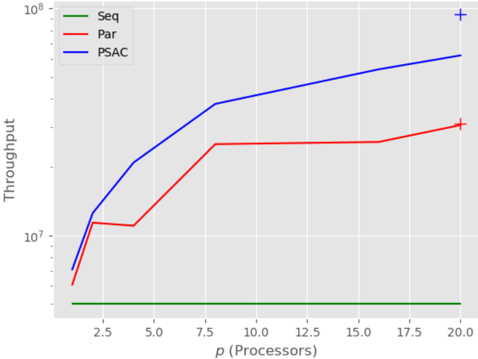
n	k	Seq	Parallel Static				PSAC Compute				PSAC Update					
			1	20	20ht	SU	1	20	20ht	SU	1	20	20ht	SU	WS	T
10^6	10^0	199ms	166ms	37ms	35ms	4.69	249ms	24ms	19ms	12.80	365us	344us	417us	1.06	547.2	579.7
10^6	10^1	-	-	-	-	-	-	-	-	-	2.00ms	1.12ms	1.20ms	1.78	100.0	178.4
10^6	10^2	-	-	-	-	-	-	-	-	-	13ms	3.14ms	3.07ms	4.53	14.33	65.00
10^6	10^3	-	-	-	-	-	-	-	-	-	78ms	12ms	11ms	7.03	2.53	17.79
10^6	10^4	-	-	-	-	-	-	-	-	-	255ms	40ms	31ms	8.10	0.78	6.34
10^6	10^5	-	-	-	-	-	-	-	-	-	498ms	82ms	60ms	8.19	0.40	3.28
10^6	10^6	-	-	-	-	-	-	-	-	-	699ms	284ms	272ms	2.56	0.29	0.73

Table 3.1: Convex Hull Benchmark

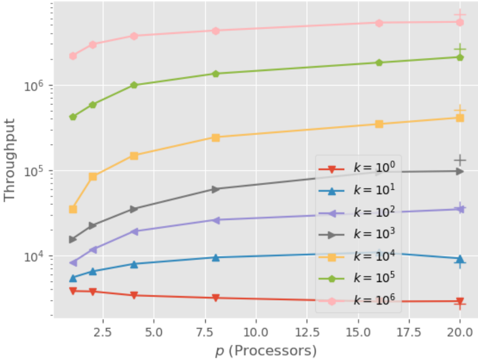
The above table represents numerical benchmark results for InitBridge. A granularity control of 256 was used for the benchmark.

SU is the relative speedup of the 20ht benchmark compared to the corresponding 1 core benchmark. WS denotes the work-savings, i.e. the relative speedup of the 1 core dynamic benchmark compared to the static sequential benchmark. Total is the total speedup, i.e. the relative speedup of the 20ht benchmark compared to the static sequential algorithm (equivalently, the product of SU and WS). The benchmarks were executed on the CMU Parallel cluster with 20 cores (and 40 threads with hyperthreading). We used the Google Benchmark C++ library to measure the speed (in real/wall time) of each benchmark. We run each benchmark ten times and take the average running time.

The benchmarks can be better summarized with the following graphs (the + symbols indicate the performance of hyperthreading):

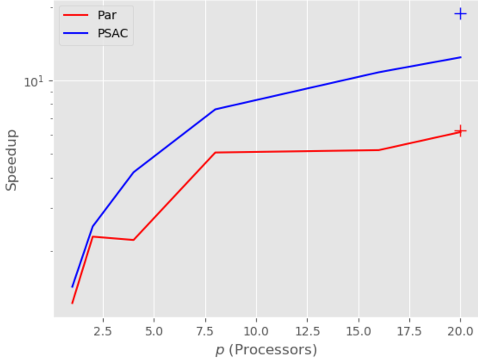


(a) Initial Computation

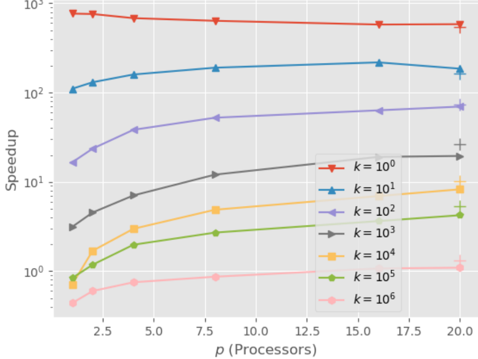


(b) Updates

Figure 3.1: Convex Hull: Absolute throughput (input updates per second) vs number of cores

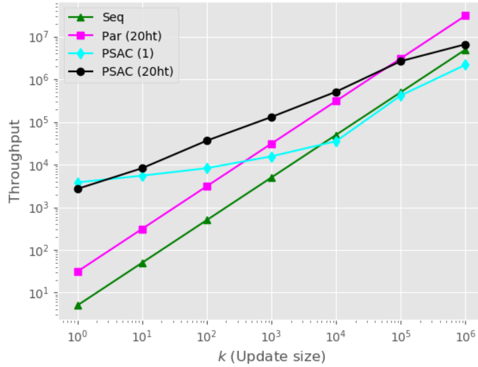


(a) Initial Computation

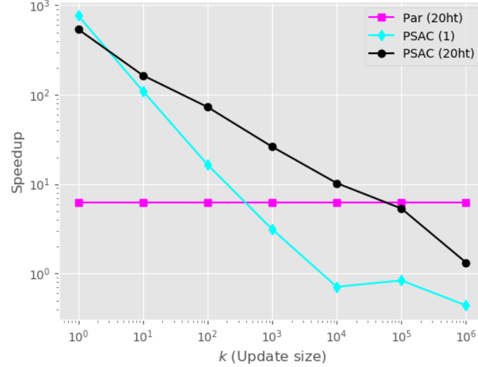


(b) Updates

Figure 3.2: Convex Hull: Speedup relative to sequential baseline vs number of cores



(a) Throughput vs update size



(b) Speedup vs update size

Figure 3.3: Convex Hull: throughput and speedup vs update size

Similar to what we observed in the BST benchmarks, Figure 3.3b shows that PSAC provides a significant benefit over non-self-adjusting implementations practically when it comes to smaller updates: the cross over point for filter and mapreduce occur when the update size is about 10^4: for updates larger than that it is more efficient to just build the data structure from scratch but for smaller updates PSAC is very effective.

We can now also measure how granularity control affects speedup:

Granularity	**Memory**	**Run** $p = 1$	**Update** $k = 1$	**Run** $p = 20\text{ht}$	**Update** $k = 10^4$
16	259.16 MB	670 ms	301 us	43.1 ms	50.3 ms
32	143.89 MB	367 ms	290 us	28.2 ms	42.1 ms
64	78.51 MB	244 ms	270 us	19.3 ms	32.1 ms
128	42.59 MB	164 ms	263 us	13.6 ms	27.8 ms
256	22.87 MB	141 ms	**261 us**	10.6 ms	19.5 ms
512	12.23 MB	121 ms	272 us	9.50 ms	15.9 ms
1024	6.51 MB	111 ms	281 us	10.1 ms	**14.4 ms**
2048	3.83 MB	104 ms	356 us	9.59 ms	17.2 ms

Table 3.2: Granularity Control Benchmarks for Convex Hull

We use different powers of 2 for benchmarking since cache lines are powers of 2, so this way we can optimize the reads on these cache lines. The memory column in the above table specifies the memory used by the WRSP tree for the initial computation. As expected, the memory use decreases as we increase granularity control since we decrease the size of the tree in terms of number of nodes, which in turn decreases the size of the WRSP tree.

The initial computation time decreases with increase in granularity control, although the benefits start to taper off as we approach large granularity controls. There is also a sweet spot for the best update times, which is achieved at 256 for 1 update, and at 1024 for updates of size 10^4 with 40 hyperthreads.

Chapter 4

Imperative Parallel Self-Adjusting Computation

In the framework we have discussed so far, there is one key restriction: we can write to a mod only once per computation. This is a reasonable restriction to make since it makes the implementation extremely clean and still lets us write a wide range of algorithms and data structures. However, if we allow multiple writes per computation then we run into a couple of problems:

1. While writing to a mod, not all readers should be notified: only the readers that are chronologically ahead of the the writer are affected and must be notified so that we avoid doing extra work.
2. Two readers reading the same mod might need to read different values based on the "history" of values of the mod.

In the following sections, we will take a look at how to resolve these two issues, but we must first establish some key assumptions about our model.

4.1 Previous Work and Assumptions

Acar et al. [6] first presented a framework for imperative self-adjusting computation in the sequential setting. The broad idea behind their framework is maintaining a global timeline of reads and writes using an order-maintenance data structure, so that they always know which readers will be affected by an inserted/deleted/updated writer. They also maintain a priority queue of all affected readers, which are prioritized by their timestamp in the global timeline. Using this implementation, they proved the following bound for change propagation:

Theorem 4.1.1. *[6] Let A be the set of affected readers of a mod m. We can partition A into A_d: the affected readers that are deleted, and A_e: the affected readers whose closures were executed again. For some re-executed reader $r \in A_e$, we can denote its work as $|r|$ assuming all self-adjusting primitives have a constant overhead. If n_t is the number of time stamps deleted during change propagation, n_q is the maximum size of the priority queue and n_{rw} is an upper bound on*

the number of readers and writers a mod can have, then change propagation takes

$$O\left(|A|\log n_q + |A|\log n_{rw} + n_t \log n_{rw} + \sum_{r\in A_e} |r|\log n_{rw}\right)$$

time. Note that if $n_{rw} = O(1)$ (constant number of reads and writes, then the above bound can be simplified to $O\left(|A|\log n_q + \sum_{r\in A_e} |r|\right)$.

Note that the above framework is only applicable in the sequential setting: in order to extend this framework to the parallel setting, we must also make a few assumptions about the readers and writers. Namely, we will assume that while readers can concurrently read from a mod, there are no concurrent reads and writes on the same mod. We also make an assumption about concurrent writes: a mod can only have more than one concurrent writer if all the writers write the same value in all possible traces of the algorithm (we will see an example of this later).

4.2 Write Nodes

One important step in building an imperative PSAC framework is the introduction of Write (W) nodes to the RSP tree (creating WRSP trees). On creation, these W nodes contain two fields: val, the value written by the W-node, and mod, a pointer to the mod being written to. Creating separate W-nodes instead of simply converting existing S-nodes and R-nodes into potential writers provides a cleaner implementation and reduces the amount of space used up by the tree.

Given the introduction of W-nodes, we can now modify read nodes to not just read the value of the given mod, but rather find the last W-node that wrote to the mod before the reader, and return its val. This requires us to store pointers to all the writers of a mod at the mod as well (and therefore indirectly store all the values written to the mod, i.e. its history).

We also introduce a new primitive: **write_internal**. Since every write in inside a self-adjusting computation must be associated with a node in the WRSP-tree, and external writes do not have such a node, this operation needs to be semantically different from the external **write**. This **write_internal** primitive is responsible for creating the new W-node as a left child of the current node of the tree, and then move the continuation to its right sibling, an S-node.

Having access to the set of writers in the mod itself (as well as the set of readers of the mod) also helps us selectively notify the appropriate readers when a writer changes the the value it is writing. However, in order to know which writers to modify we must have a notion of a global timeline and know in what order these nodes are in the timeline (similar to the idea of Acar et al. [6]). We also need to know this order for finding the last writer that occurs before a read for determining the correct value to be read from a mod. We want to maintain such a timeline in parallel as well, so we would need some extra tools.

4.3 Parallel Order Maintenance

In order to resolve the problems discussed earlier, we can take inspiration from the framework developed by Acar et al. [6]. They use an order-maintenance data structure to tell if a reader reads a value before or after a value is written to the mod, and we can try doing something similar in the parallel setting.

Fortunately, there is a concurrent version of the data structure that can help us insert values and answer order queries concurrently and in $O(1)$ time. A sequential version of the such a data structure was first proposed by Bender et al. [9]. Not only do they present an order maintenance data structure that can help maintain a total order on a set of values, but they propose an **SP-maintenance structure** called SP-order which can help maintain a partial order: if two values cannot be compared (such as two operations that occur on parallel threads), then the data structure can determine that as well. Such a set of relationships is best described with an Series-Parallel (SP) tree: similar to the one that we discussed in Chapter 1. They use a well-known technique of maintaining two orders - English and Hebrew [23] - in order to achieve this. They also proposed a parallel implementation of the data structure, although it has quite a bit of undesirable overhead.

Later, this data structure was modified by Utterback et al. [26] in such a way that inserts and queries still take $O(1)$ amortized work, but the data structure can be created in $O(W/P + S)$ time, where W is the number of number of values in the data structure, S is the height of the SP-tree, and P is the number of processors. This is asymptotically optimal and is achieved by modifying the job scheduler itself. The interface for the data structure is as follows:

- PRECEDES(x, y): returns true if and only if x is preceded by y in the total order.
- INSERT(x, y): given a pointer to an existing element x in the data structure, insert the element y immediately after the element x in the total order. This means that all existing successors of x will be successors of y as well, and all predecessors of x are predecessors of y.

We can now finally discuss the implementation of our imperative framework.

4.4 Implementation

For our purposes, we can maintain the above SP-maintenance data structure on the pre-order traversal of the nodes of the WRSP-tree, and therefore know at all times the temporal relationship between any read nodes and write nodes in $O(1)$ time. We can also store the set of writers of a mod (let's call them writers), in a concurrent binary search tree [22]: this way we can perform binary search on them to find the last writer present before a given reader. The set of readers (readers) can also be stored as a concurrent BST: we can determine the set of readers that are affected by the insertion/deletion/update of a writer by splitting the BST at the time stamp of the writer, and then we can mark all the readers that occur after it. Every single node in the WRSP tree also stores a pointer to its node in the order maintenance data structure, and is represented by the field om_node.

However, for most practical purposes the set of readers and writers is going to be small (about $O(1)$), and we can therefore forgo these concurrent BSTs for the real implementation. Instead, we can store readers as an unordered concurrent set and writers as a vector of pointers, ordered by their position in the order maintenance data structure. Therefore the get_value function for obtaining the value read by the reader, and the notify_readers function for marking the appropriate readers can be implemented as follows:

Algorithm 15 Obtaining the value read by a reader r for a mod m

```
 1: function GET_VALUE(m : T mod, r : RNode):
 2: i ← 0
 3: while i < len(m.writers) do
 4:     if PRECEDES(r.om_node, m.writers[i].om_node) then
 5:         if i = 0 then
 6:             return m.value
 7:         end if
 8:         return m.writers[i − 1].val
 9:     end if
10:     i ← i + 1
11: end while
12: return m.writers[i − 1].val
```

Algorithm 16 Notifying the appropriate readers of mod m when the writer w is updated

```
1: function NOTIFY_READERS(m : T mod, w : WNode):
2: if w = ⊥ then
3:     m.readers.For_All(r : RNode ↦ r.set_modified())
4: else
5:     m.readers.For_All(r : RNode ↦
6:         if PRECEDES(w.om_node, r.om_node) then
7:             r.set_modified()
8:         end if
       )
9: end if
```

Note that both of the above functions make a clear distinction between external writes and internal writes. In the function GET_VALUE, we return m.value (the value written by an external write) if there are no writer nodes present before the given reader node r. The function NOTIFY_READERS is called with $w = \perp$ if the external write changes its value: in this case all the readers of the mod must be notified irrespective of their position. The method set_modified() marks the reader for change propagation so that the readers are re-evaluated in the correct order later. Further details of the implementation can be found in the Github repository [1].

[1] https://github.com/cmuparlay/ipsac

Another key implementation detail has to do with deletion of nodes. Since the SP-maintenance data structure does not explicitly support a DELETE operation: we need to do the following:

- When deleting a node in the SP-maintenance data structure, do not explicitly delete the node immediately, only mark it as deleted and increment a global counter deletes.
- When deletes is at least half the total number of nodes in the data structure (for which we would maintain another global counter incremented when inserting a node), we create the entire data structure from scratch, with only the unmarked nodes.

It can be seen from the above construction that deletion of a node from the data structure requires $O(1)$ amortized time by a basic charging argument.

Another key hurdle has to do with the deletion of the nodes in the WRSP-tree itself. While re-executing the closure of a read node, the entire subtree of the R-node needs to be deleted. Therefore if there are any W-nodes in the subtree being deleted, those writers need to be deleted from the writers variable of their respective mods. This is achieved by simply traversing through the tree to find all the writers before releasing it for garbage collection.

4.5 Breadth First Search

Using Breadth First Search (BFS) on undirected graphs to find the shortest path lengths to different nodes is a good candidate for demonstrating imperative parallel self-adjusting computation. We keep track of the minimum distance to each vertex in D, a vector of mods, and each of these mods is written to twice: the first time to mark the node as unvisited, and the second time to write the shortest path distance and mark it as visited. The following is an implementation of one iteration of BFS (which takes in one frontier of vertices F and calls the next iteration with the next frontier F') which assumes an adjacency matrix representation of the input graph:

Algorithm 17 One iteration of parallel self-adjusting breadth first search.

1: **psac_function** BFS_ITER(A : **bool mod array array**, D : **int mod array**, F : **int array mod**, $iter$: **int**):
2: **local** $n \leftarrow$ **len**(D)
3: **local** $F' \leftarrow$ **alloc_mod_array**(**bool**, n)
4: **psac_parallel_for**(i : **int**, $0, n$) :
5:     **write_internal** ($F'[i]$, **false**)

6: **psac_parallel_for**(i : **int**, $0, n$) :
7: **with read** ($F[i]$) **as** f **do**
8: **if** f **then**
9: **psac_parallel_for**(j : **int**, $0, n$) :
10: **with read** ($A[i][j], D[j]$) **as** e, d **do**
11: **if** $e \wedge d = -1$ **then**
12: **write_internal** ($D[j], iter$)
13: **write_internal** ($F'[j]$, **true**)
14: **end if**
15: **end if**
16: **local** $T \leftarrow$ **alloc_mod**(**bool**)
17: IS_EMPTY(T, F')
18: **with read** (T) **as** $done$ **do**
19: **if not** $done$ **then**
20: BFS_ITER($A, D, F', iter + 1$)
21: **end if**

Note that in Algorithm 17 we might have concurrent writes on line 12-13. But since those writes are all writing the same value there are no races. However, note that even if these writes are all writing the same value: all of them need to be stored separately in the writer set of the mod: if one of the writers is deleted and there were 2 writers that wrote to that mod in parallel, then the other writer is still present, and therefore so is the written value.

Algorithm 17 is called from the following top-level function:

Algorithm 18 Top level function for self-adjusting breath first search

1: **psac_function** BFS(A : **bool mod array array**, D : **int mod array**):
2: **local** $n \leftarrow$ **len**(D)
3: **local** $F \leftarrow$ **alloc_mod_array**(**bool**, n)
4: **psac_parallel_for**(i : **int**, $1, n$) :
5: **write_internal** ($D[i], -1$)
6: **write_internal** ($F[i]$, **false**)
7: **write_internal** ($D[0], 0$)
8: **write_internal** ($F[0]$, **true**)
9: BFS_ITER($A, D, F, 1$)

In Algorithms 17 and 18 it can be clearly seen that every $D[i]$ is written with at most 2 different values (once in BFS, and once in BFSITER).

Bibliography

[1] Umut A Acar. *Self-adjusting computation*. Carnegie Mellon University, 2005. 1, 1.1

[2] Umut A Acar. Self-adjusting computation: (an overview). In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 1–6, 2009. 1.1

[3] Umut A Acar, Guy E Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–259, 2002. 1.1

[4] Umut A Acar, Guy E Blelloch, and Robert Harper. Selective memoization. *ACM SIGPLAN Notices*, 38(1):14–25, 2003. 1.1

[5] Umut A Acar, Guy E Blelloch, Robert Harper, Jorge L Vittes, and Shan Leung Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. 2004. 1, 1.3.1

[6] Umut A Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 309–322, 2008. 1, 1.1, 4.1, 4.1.1, 4.2, 4.3

[7] Daniel Anderson, Guy E Blelloch, Anubhav Baweja, and Umut A Acar. Efficient parallel self-adjusting computation. *arXiv preprint arXiv:2105.06712*, 2021. 1, 1.1, 1.2, 1.3, 1.3

[8] C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996. 1.1, 3

[9] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Charles E Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 133–144, 2004. 4.3

[10] Jon Louis Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, (4):333–340, 1979. 2

[11] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011. 1.1

[12] Pramod Bhatotia, Pedro Fonseca, Umut A Acar, Björn B Brandenburg, and Rodrigo Rodrigues. ithreads: A threading library for parallel incremental computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Lan-*

guages and Operating Systems, pages 645–659, 2015. 1.1

[13] Guy E Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 16–26, 1998. 2.1

[14] Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264, 2016. 2.2

[15] Sebastian Burckhardt, Daan Leijen, Caitlin Sadowski, Jaeheon Yi, and Thomas Ball. Two for the price of one: A model for parallel and incremental computation. *ACM SIGPLAN Notices*, 46(10):427–444, 2011. 1.1

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 2.3

[17] Mingdong Feng and Charles E Leiserson. Efficient detection of determinacy races in cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999. 1

[18] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Info. Pro. Lett.*, 1:132–133, 1972. 1.1, 3

[19] Matthew Hammer, Umut A Acar, Mohan Rajagopalan, and Anwar Ghuloum. A proposal for parallel self-adjusting computation. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 3–9, 2007. 1.1

[20] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, 31(2):249–260, 1987. 1.1

[21] Gary L Miller and John H Reif. Parallel tree contraction and its application. Technical report, HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB, 1985. 1.1

[22] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 317–328, 2014. 4.4

[23] Itzhak Nudler and Larry Rudolph. Tools for the efficient development of efficient parallel programs. In *Proceedings of the first Israeli conference on computer systems engineering*, pages 4–1, 1986. 4.3

[24] Mark H Overmars and Jan Van Leeuwen. Maintenance of configurations in the plane. *Journal of computer and System Sciences*, 23(2):166–204, 1981. (document), 3, 3.1, 3.1, 3.3

[25] Alexander Stepanov and Meng Lee. *The standard template library*, volume 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995. 2.5

[26] Robert Utterback, Kunal Agrawal, Jeremy T Fineman, and I-Ting Angelina Lee. Provably good and practically efficient parallel race detection for fork-join programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 83–94, 2016. 4.3