# Efficient Parallel and Parallel External Memory Algorithm for Geometric Query Problem using Segment Tree

***An M. Tech Project Report Submitted***
***in Partial Fulfillment of the Requirements***
***for the Degree of***

**Master of Technology**

*by*

**Nihar Shah**
(204101052)

*under the guidance of*

**Prof. G. Sajith**


**to the**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**
**INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI**
**GUWAHATI - 781039, ASSAM**

# CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Efficient Parallel and Parallel External Memory Algorithm for Geometric Query Problem using Segment Tree**" is a bonafide work of **Nihar Shah** (**Roll No. 204101052**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Prof. G. Sajith**

Department of Computer Science & Engineering,

Indian Institute of Technology Guwahati, Assam.

# Contents

# List of Figures

# Abstract

*This work provides an efficient parallel and parallel external memory (PEM) algorithm for stabbing query problem. The stabbing query problem is a geometric query problem which is an essential part of computational geometry that focuses on answering multiple queries efficiently. Geometric query problems have numerous applications in various fields like motion planning and visibility problems in robotics, geometric location and search or route planning in the geographic information systems, VLSI design, computer graphics, geometric modeling etc. First, we devise a parallel algorithm that takes $O(\log n + t)$ time to answer a query. Then we present the same algorithm to a PEM model that minimizes the I/O operations and takes $O(\log_B N + t/B)$ I/O operations for each query.*

# Chapter 1

# Introduction

Geometric query problems are an important part of computational geometry problems. It involves problems like range searching, point location, nearest neighbour, ray tracing etc. These problems have applications in diverse fields. For instance, a common problem in computer graphics is finding which area of the screen is clicked by a point. Here point represents a query, and the screen is a static structure which contains many geometric shapes. One more example is in defence systems or radar systems; the problem is to find whether an aircraft violated borders. Here, the country's border is treated as a geometric structure, and the aircraft represents a point that is the query. This paper looks at one famous geometric query problem known as the stabbing query problem. We devise a parallel and a parallel external memory (PEM) algorithm to solve this problem using segment trees. The segment tree is a critical and highly versatile data structure in computer science that helps us solve a wide range of problems and has use cases in diverse fields like computational geometry, computer graphics and machine learning. [ACG+85]. Segment trees are helpful when dealing with ranges or information about intervals. The segment tree does not have a fixed structure. The structure and the information a particular node of a segment tree contains depends on the type of problem we want to solve.

## 1.1 Models of Computation

### 1.1.1 Parallel Random Access Machine (PRAM)

Parallel random access machine (PRAM) is the most used model to write a parallel algorithm. It is parallel version of standard RAM model that allows parallel algorithm developers and designers to treat processing power as unlimited. PRAM algorithms are primarily theoretical but can be used to develop an efficient parallel algorithm for practical machines.

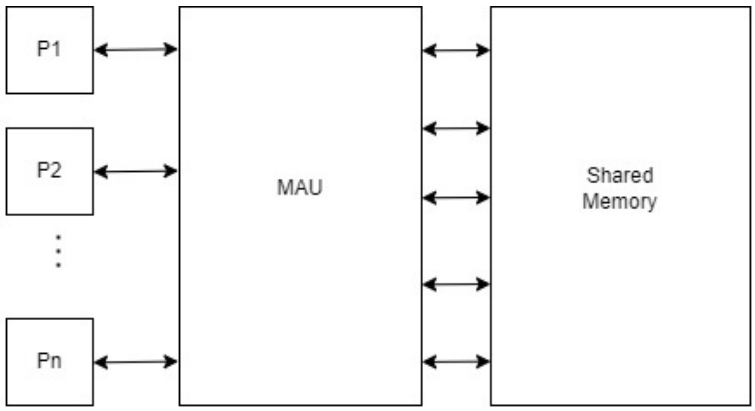


**Fig. 1.1**: PRAM Architecture

There are four versions of the PRAM model based on how they handle read and write conflict. The model used to build the parallel algorithm for the stabbing query problem is the CREW PRAM model, which means concurrent reads are allowed, but all the writes must be done exclusively. The devised algorithm uses this model and takes $O(\log n + t)$ time for answering each query, where $n$ is the input size, and $t$ is the size of the answer set.

### 1.1.2 Parallel External Memory (PEM)

With continuous scaling of the applications and the significant increase of data from the last few decades, the problem might be too large to fit in the main memory. If that is the case, the problem must be stored in the large secondary memory, which is slower than the main memory. The I/O operations between quick main memory and slower secondary

memory are bottlenecks in large scale applications. Our goal for this type of application is to reduce the number of I/O operations.

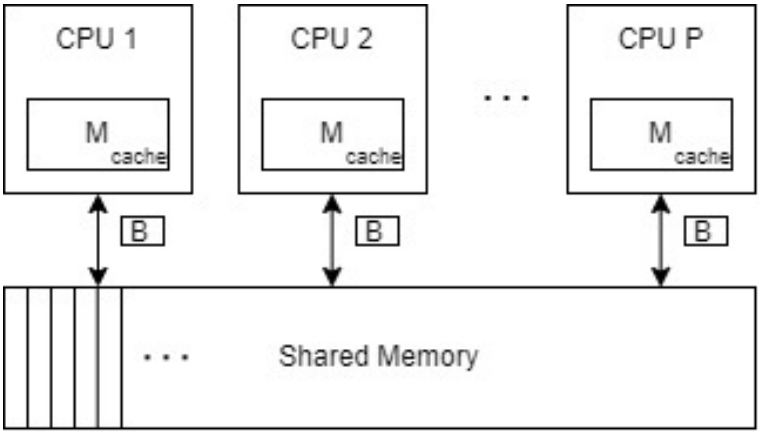


**Fig. 1.2**: PEM Architecture.

Figure 1.2 shows the parallel external memory (PEM) model. It is a parallel version of the external memory (EM) model where there are $P$ processors, each having its own cache of size $M$. Shared memory is of conceptually infinite size. Data is stored in the cache and shared memory in the blocks of size $B$. Each processor can only perform the computation on the data if it is present inside that processor's cache. Using one parallel I/O operation, all processors can transfer one block of size $B$ between cache and shared memory. The cost of an algorithm for this model is the total I/O operations it performs.

**The following parameters are used throughout the paper.**

$N$ = number of items in the problem.

$M$ = number of items that can fit in the processor's cache.

$B$ = number of items per disk block.

$K$ = number of queries in the problem.

$P$ = number of processors available.

$t$ = number of items in the solution.

# Chapter 2

# Problem Statement

## 2.1 1-$D$ stabbing query problem

Given a set $S$ of $n$ line segments and a set $Q$ of $K$ query points, a segment $s \in S$ is stabbed by a point $p \in Q$ iff that point is in the range of $s$. We need to find the stabbed segments for each point $p \in Q$. All the segments are on the real line, they are shown above each other just for the representation purposes.

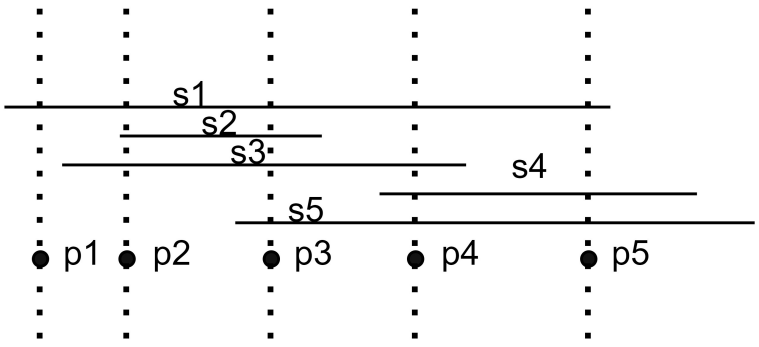


**Fig. 2.1**: 1-D stabbing query problem.

For the given example point $p1,\ p2,\ p3,\ p4\ and\ p5$ stabs $\{s1\},\ \{s1, s2, s3\},\ \{s1, s2, s3, s5\}$, $\{s1, s3, s4, s5\}$ *and* $\{s1, s4, s5\}$ respectively.

## 2.2 2-$D$ stabbing query problem

The stabbing query problem in 2-$D$ settings is similar to the 1-$D$ settings. Instead of a real line, now we have a plane which contains some line segments. a point $p$ stabs the segment $s$ iff the vertical line crossing the point $p$ crosses the segment $s$. Given a finite set of line segments $S$ and a set of Query points $Q$ we need to find the stabbed segments for each points in Q.

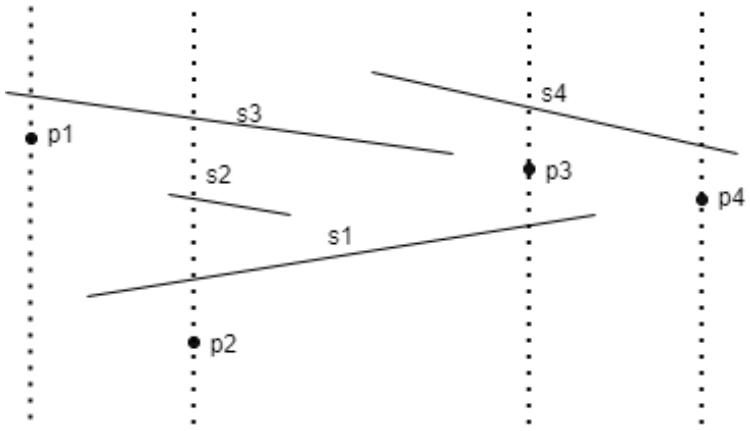


**Fig. 2.2**: 2-D stabbing query problem

For the given example point $p1,\ p2,\ p3$ *and* $p4$ *stabs* $\{s3\},\ \{s1, s2, s3\},\ \{s1, s4\}$ *and* $\{s4\}$ respectively.

One naive approach to find stabbed segments is for each point we go through whole set of segments and print all the segments containing query point. This approach is not optimal and for each query point it requires $O(n)$ time.

# Chapter 3

# Segment Tree Structure

We have seen the problem and naive solution to this problem, but of course, the naive algorithm is not optimal in terms of time, i.e. it takes $O(nK)$ time. We shall now see how we can improve the time using some extra space. To answer the query faster, we will build the segment tree over all the line segments. It will require $O(n \log n)$ space, and it can answer all the queries in $O(\log n + t)$ time. This segment tree implementation assumes that the input's scope remains constant till the end, *i.e.* we cannot insert the line segment once we have built the segment tree.

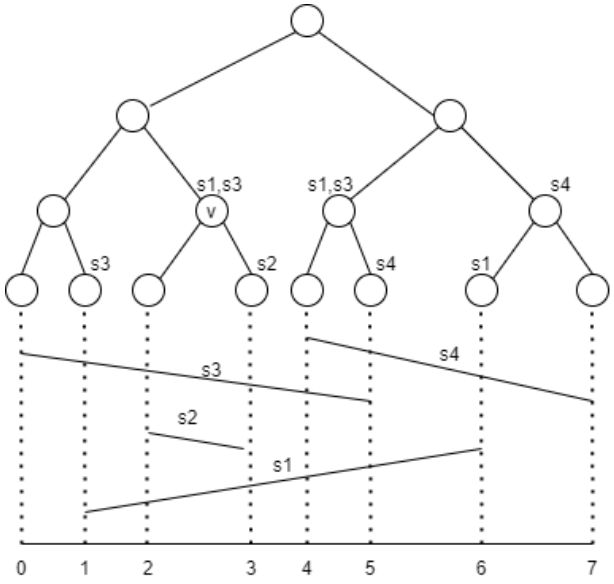


**Fig. 3.1**: Segment tree for interval stabbing problem.

## 3.1 Representation of Segment Tree

Figure 3.1 shows the constructed segment tree for the stabbing query problem. A node is assigned in the segment tree for each endpoint of the line segments. These nodes form the leaves of the segment tree. A perfect binary search tree is generated on the top of these leaves, that is the segment tree. Each node in the segment tree contains a few things: the range it covers, i.e. strip. And the number of segments crossing its strip but not its parent's strip, i.e. the cover set. Here, the term crossing means the segment should cover the entire range of a node's strip.

Let us see how we can build the strip of each node. The strip is nothing but the closed interval i.e. $[a, b]$. Let $S$ be the set of segments and $X$ be the set of endpoints of all the segments of $S$. To get the strip of the leaves we put infinitely long vertical lines at each point $x$ in $X$ (see Figure 3.1). The $strip(x)$ of the leaves is the range between the line crossing $x$ and the first line on the left of $x$. The strip of the leftmost leaf node is unusable but for the sake of completeness we take it as $(-\infty, 0]$. For each internal node $i$ the $strip(i)$ is the union of the strip of its two children. If a segment $s \in S$ crosses the node $u$ iff it covers the entire range of $strip(u)$. Now we define $cover(u)$ as follows. $cover(u) = \{s \in S \mid s \text{ crosses } strip(u) \text{ and not } strip(parent(u))\}$. For instance in Figure 3.1, $strip(v)$ is $[1, 3]$ and $cover(v) = \{s1,\ s3\}$

## 3.2 Query Processing using Segment Tree

For each query point, we do one root to leaf traversal so that the point always falls inside the strip of the current node. While doing the traversal, we keep track of the cover set of each node on the root to leaf path. Union of all this cover set forms the result of each point.

# Chapter 4

# Parallel Algorithm

## 4.1 Algorithm illustration

The segment tree defined in chapter 3 is generated using this algorithm. Build tree function builds our final segment tree. All the helper functions are defined in the algorithm. The algorithm begins by sorting the endpoints of all the segments and then building the base structure in a bottom-up fashion level by level. Then we insert each segment in the segment tree in its particular location to build a cover set of each node.

**Algorithm 1** Parallel Algorithm for building Segment Tree

**Input:** A set $S$ containing line segments and a set $X$ containing endpoints
**Output:** A segment tree will be generated

```
1: function BuildSkeletonTree(T, X, n)
2:     for all i ← n to 2 * n − 1 do in parallel
3:         strip(T_i) = [X[i − n − 1] : X[i − n]]        ▷ Generating strip for leaves
4:     end for
5:     while n ≠ 0 do
6:         if n/2 ≤ i ≤ n − 1 then
7:             strip(T_i) = strip(T_2i) ∪ strip(T_2i+1)
8:         end if
9:         n ← n/2
10:    end while
11: end function

12: function InsertSegment(T_i, s)
13:    if strip(T_i) ⊆ s then
14:        insert s in cover(T_i)
15:    else
16:        if strip(T_2i) ∩ s ≠ φ then
17:            InsertSegment(T_2i, s)
18:        end if
19:        if strip(T_2i+1) ∩ s ≠ φ then
20:            InsertSegment(T_2i+1, s)
21:        end if
22:    end if
23: end function

24: function BuildTree(S, X)
25:    Sort X using Cole's mergesort                  ▷ A parallel sorting algorithm
26:    BuildSkeletonTree(T, X, n)
27:    for all s ∈ S do in parallel
28:        InsertSegment(T_1, s)
29:    end for
30: end function
```

**Algorithm 2** Parallel Algorithm for query points

**Input:** A set $Q$ containing query points and segment tree $T$
**Output:** Answer for each point will be reported

```
1: function QueryOnTree(T_i, p)
2:     Report all intervals in cover(T_i)
3:     if T_i is not a leaf node then
4:         if p ∈ strip(T_2i) then
5:             QueryOnTree(T_2i, s)
6:         else
7:             QueryOnTree(T_2i+1, s)
8:         end if
9:     end if
10: end function

11: function Query(Q, T)
12:     for all p ∈ Q do in parallel
13:         QueryOnTree(T_1, p)
14:     end for
15: end function
```

## 4.2 Time Complexity

Cole's merge sort takes $O(\log n)$ time for sorting an array of $n$ elements using $O(n)$ processors. The function build skeleton tree takes $O(\log n)$ time because we find the strip of each level simultaneously in constant time, and there are $\log n$ such levels present in the segment tree. Inserting the segment takes $O(\log n)$ time because as the segment is a contiguous range, there can be at most two nodes per level containing a segment. So at each level, we can call the function at most two times, and there are $\log n$ such levels. For answering a query, we do one root to leaf traversal, and on the way we report all the segments in the cover sets of the nodes. We require extra $t$ time to report the segments where $t$ is the number of items in the answer set. When the query point is precisely on the top of an endpoint, the answer set can have some duplicate segments that can be removed if required in extra $t$ time. This leads the overall time complexity to be $O(\log n + t)$ with $O(n)$ processors.

## 4.3 Space Complexity

Each node in a segment tree contains a cover set that contains some segments. At most, there can be two nodes per level containing a segment, and there are $n$ such segments and $\log n$ such levels, leading the overall space complexity to $O(n \log n)$.

# Chapter 5

# Parallel External Memory Algorithm

## 5.1 Algorithm illustration

The time required to answer each query for the sequential setting was $O(\log n + t)$, where t is the total number of intervals reported for a query point. Here we will look at the Parallel External Memory solution, which requires $O(\log_B N + t/B)$ I/O's for each query point.

The segment tree structure remains the same for the Parallel External Memory solution, but how we are storing tree $T$ is essential. We will divide tree $T$ into subtrees of height $\log B$ and keep it in one disk block of size B. Since the tree's height is $\log N$ with these settings, a root to leaf path will only require $\log N / \log B = \log_B N$ I/O operations.

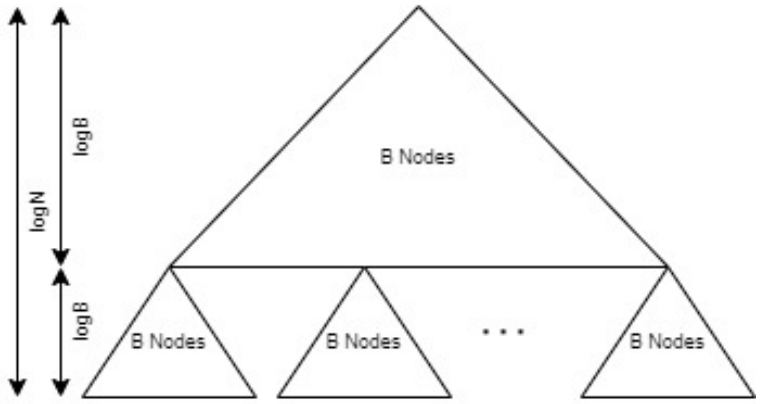


**Fig. 5.1**: External Segment Tree

To generate the skeleton tree or build each node's cover sets, we require only one root to leaf path for each processor. Hence building the segment tree requires only $\log_B N$ I/O operations. But still, for one root to leaf path, we need to do at least $\log N$ I/O, one for each node on this path, to get the cover set of each node. Hence total I/O operations required will be $O(\log N + t/B)$

## 5.2 Why we required more I/O's

If the number of segments in the cover set of a node $x$ on the root to leaf path is less than $B$, i.e. size of $cover(x) < B$, then the I/O operation to fetch the cover set of $x$ is wasteful I/O because we are bringing $B$ units of data but not using all. Contrary to the above, if the number of segments in the cover set of a node $x$ on the this path is greater than $B$, i.e. size of $cover(x) > B$, then I/O operations to fetch the cover set of $x$ are useful. If the number of wasteful I/O's are less than the number of useful I/O's, then the total I/O's required to report all the segments on the path $P$ will be $\leq 2t/B$, which means for each useful I/O, we can pair one wasteful I/O. Hence this will give us the desired I/O complexity of $O(\log_B N + t/B)$. Consider a node $x$ on the root to leaf path such that its cover set has at least $B$ segments. It is then easy to see that all but the last I/O at node $x$ will return $B$ segments. This implies that the number of wasteful I/O's at node $x$ is upper-bounded by the number of useful I/O's. Thus the problem nodes are only those that have fewer than $B$ intervals stored at them.

## 5.3 Optimization Path Caching

Instead of doing a direct I/O operation on these problem nodes, we will copy the cover set of such nodes and store them in the cache of the leaves. This way, instead of doing $\log N$ wasteful I/O's, we just perform at most one wasteful I/O.

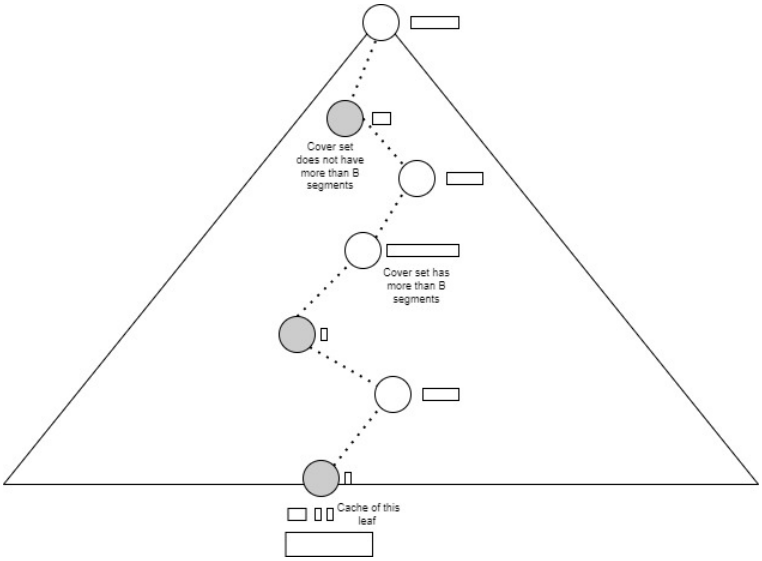


**Fig. 5.2**: Path Caching

**Optimized Approach**

- For each leaf $l$ locate the cover sets along the root to $l$ path, which has less than $B$ segments.
- Make copies of such cover sets and store them in the cache of that leaf.
- These caches are stored as blocks of size $B$ on the secondary memory.

## 5.4 I/O Complexity

This way number of wasteful I/O's can never be more than the useful I/O's, and hence the number of wasteful I/O's are upper-bounded by the number of useful I/O's. We have $K$ query points and $P$ processors available, which gives us the desired I/O complexity of $O(K/P(\log_B N + t/B))$. If $K$ is $O(P)$ then I/O complexity is $O(\log_B N + t/B)$. If $K$ is smaller than $P$ we still require $O(\log_B N + t/B)$ I/O operations.

## 5.5 Space Complexity

To store the tree $T$, we require $O(N/B)$ blocks. Total number of segments stored in cover sets is $O(N \log N)$; hence it will require $O(N/B \log N)$ blocks. Additionally, we are storing $O(B \log N)$ segments in the cache of the leaves. There are $2n$ such leaves (2n endpoints) which give us $O(NB \log N)$ total segments; hence space required is $O(N \log N)$ blocks.

# Chapter 6

# Conclusion

We saw an $NC(n)$ parallel algorithm to build and query on the segment tree that takes $O(\log n + t)$ time using $O(n)$ processors and takes $O(n \log n)$ space which is efficient compared to the standard sequential algorithm for the same problem. We converted the same parallel algorithm to parallel external memory algorithm which takes $O(\log_B N + t/B)$ I/O operations and requires $O(N \log N)$ blocks. Segment trees are prevalent due to their wide range of use cases, efficiency and ease of usage. The segment tree that is generated for the stabbing query problem in this paper can be used to answer many more problems like the first hit problem, where we find the first segment a point hits in direction d. This problem is similar to the stabbing problem, but we also need to maintain the order of segments concerning the y-axis. A whole new kind of problem that does not deal with query problems can also be solved by the segment tree, i.e. the line intersection problem in a plane where we have to find from a given set of line segments whether two lines intersect.

# References


[ACG+85] Alok Aggarwal, Bernard Chazelle, Leo Guibas, Colm Odunlaing, and Chee Yap. Parallel computational geometry. *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, 1985.

[ACG+88] A. Aggarwal, B. Chazelle, L. Guibas, C. Ódúnlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(1-4):293–327, 1988.

[AL93] Selim G. Akl and Kelly A. Lyons. *Parallel computational geometry.* Prentice Hall, 1993.

[Ber08] M. de. Berg. *Computational geometry: algorithms and applications.* Springer, 2008.

[CJ07] Xinuo Chen and Stephen A. Jarvis. Distributed arbitrary segment trees: Providing efficient range query support over public dht services. *2007 IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications*, 2007.

[DRC90] Frank Dehne and Andrew Rau-Chaplin. Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry. *Graph-Theoretic Concepts in Computer Science Lecture Notes in Computer Science*, page 316–329, 1990.

[Ger06] Alexandros V. Gerbessiotis. An architecture independent study of parallel segment trees. *Journal of Discrete Algorithms*, 4(1):1–24, 2006.

[SB19] Yihan Sun and Guy Blelloch. Implementing parallel and concurrent tree structures. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019.