Parallel Fenwick Tree

Tzu-Yen Tseng*
Jim Shao*
tzuyent@andrew.cmu.edu
chiatses@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Abstract

Parallel Fenwick Tree is an intricate data structure, and its unique access patterns make it hard to parallelize. We implemented several versions of Parallel Fenwick Trees with OpenMP and tested each of them on Carnegie Mellon University's GHC machines and Pitrsburgh Supercomputing Center's Bridges-2 machines. Our Pure Parallelism for general workload achieves 6x speedup and our Model-Parallel approach for BatchAdd optimizations reaches up to 11x speedup on PSC machines.

Keywords

Fenwick Tree, Parallelism, OpenMP

1 Introduction

The Fenwick tree was first proposed in 1994[4]. It is a data structure optimized for range queries. Peter Su et al.[7] explored the parallelism of this data structure when PRAM (Parallel Random Access Machine) first appeared and concluded that it is not an easily parallelizable data structure. As the NUMA architecture becomes more popular in multiprocessor computers these days, different tree parallelisms have been well studied. However, research on Fenwick trees remains rare due to its highly optimized data structure for a single thread, making parallelism nontrivial. Therefore, we would like to propose several potential optimizations for this data structure, either with workload balancing or with memory-balanced methods.

In Section 2, we discuss optimizations on general approaches to handle all kinds of workload for the parallel Fenwick Tree. We conclude that accumulating Adds has great potential for optimizations. In Section 3, we focus on the optimizations in BatchAdd. In Section 4, we pick out several designs with better potential for improvement and design some experiments. In Section 5, we demonstrate our experiment results and our observations. In Section 6, we conclude our methodologies and future directions.

2 General Approaches

A Fenwick Tree usually contains 2 operations, add() and sum(). This is equivalent to writes and reads, updates and queries. Therefore we use these terms interchangeably A common workload would be a series of writes with a few reads in the middle. We call this the "general workload" in this paper. In this section, we propose two main parallelism strategies to optimize the performance for handling general workloads with a simple glance at performance and potential trade-offs. All approaches throughout the research are implemented in C++.

2.1 Serialized Fenwick Tree

Let's first look at the original version of the Fenwick Tree. The serial version of the tree follows the implementation of the CP-Algorithm website [2]. This is the baseline of our measurement. The pseudocode is provided as follows.

```
class FenwickTree {
    int bits[N + 1];

    void add(int x, int val) {
        for (++x; x <= N; x += x & -x) {
            bits[x] += val;
        }
    }

    int sum(int x) {
        int total = 0;
        for (++x; x > 0; x -= x & -x) {
            total += bits[x];
        }
        return total;
    }
}
```

We can see from the pseudocode that the Fenwick Tree is already a very optimized data structure. Both the add() and sum() operations are well optimized with bit operations. For the rest of the approaches, we will propose different parallelism strategies to beat the baseline. To ensure the correctness of our approaches, we first define the correctness of the Parallel Fenwick Tree.

Definition 2.1. (Correctness) A Parallel Fenwick Tree is considered correct for all the query outputs; they should be equivalent to the outputs from a Serialized Fenwick Tree.

We mainly focus on task parallelism to achieve performance speed-up in this section.

2.2 Lazy Synchronization

A common approach mentioned in the paper on tree parallelism [1] is to delay all updates until a query is shown. The delay of execution can help us better utilize parallelism since each update can be done by only one thread. However, the query function requires all threads to be synchronized to get the most updated output. The pseudocode is described below:

```
// Fenwick Tree is an atomic array
// Utilize a sliding window to batch all writes before a read
left = 0;
right = 0;
```

*Both authors contributed equally to this research.

```
while (right < workload.size()) {
    if (workload[right] == operation.read) {
        #pragma omp parallel
        for (i = left; i < right; i++) {
            tree.add(workload[i]);
        }
        value = tree.query(workload[right]);
        left = right + 1;
    }
}
```

This approach can achieve up to 1.5x speedup when query operations are less than 0.1% in the overall workload. As query operations grow in workload, the limitations of the synchronization barrier start to overwhelm the benefits of distributing write tasks.

Query Percentage	Small Array	Medium Array	Large Array
0%	0.57x	1.07x	1.50x
0.1%	0.16x	1.01x	1.46x
1%	0.52x	0.28x	1.36x
5%	0.16x	0.64x	1.25x

Table 1: Lazy Synchronization Speedup (batch size: 65536, number of batches: 1024, thread count: 8)

2.3 Task Parallelism

The bottleneck of the Lazy Sync method lies in the synchronizations for each query. As the query frequency grows, the performance becomes extremely bad due to more synchronization barriers, limiting the chance of parallelizing tasks. To minimize the thread synchronizations, we eagerly look for a method which minimizes the number of synchronizations across threads.

Since we can parallelize the write workloads to all threads, can we also parallelize all the read workloads to all threads? If each thread receives the same order of update and query requests, the result will be equivalent to the sequential operation.

Definition 2.2. (Partial Execution) A thread in the Task Parallelism approach will receive the partial operations of the whole workload. The partial operations are in the same partial order as the original workload.

Theorem 2.3. *(Partial Ordering achieves Correctness) For each thread having its local Fenwick Tree and follows partial execution, correctness is achieved as long as all queries are distributed to all threads, and each update is handled by exactly one thread.*

That is, we need a way to distribute an update to any of the worker threads and broadcast each query to all worker threads to guarantee the correctness.

2.3.1 Centralized Scheduler. Inspired by the Morsel-driven parallelism paper [5], we decided to build a centralized task scheduler dedicated to distribute all operations. For a write operation, the scheduler will allocate the task to a thread in a round-robin manner. For a read request, the read is broadcasted to all worker threads. Each worker thread has its own tree to which they will update. The scheduler has a dedicated atomic array for each thread to write the partial query result. This is similar to a reduce function. With this implementation, we minimize thread synchronizations to only the atomic operations when writing the result.

```
class CentralizedScheduler {
    submit_add(index, value) {
        // round-robin
        worker_id = counter++ % worker_num;
        lock(worker_id);
        queues[worker_id].push(task);
        unlock(worker_id);
    }

    submit_query(index) {
        // broadcast
        for (worker_id : workers) {
            lock(worker_id);
            queues[worker_id].push(task);
            unlock(worker_id);
        }
    }

    worker_loop(worker_id) {
        while (True) {
            lock(worker_id);
            task = queue.pop();
            unlock(worker_id);
            if (task.op == "add") {
               local_tree.add(task.index, task.value);
            } else {
                res = local_tree.sum(task.index);
             global_res[task.position].fetch_add(res);
            }
        }
    }
}
```

This method achieves only a 0.4x speedup and becomes worse as the thread count grows. Since when implementing the task scheduler logic, we have dedicated queues for each thread, and each queue requires a lock. The time taken for locking/unlocking actions greatly exceeds the benefits of not synchronizing with others.

2.3.2 Lock-Free Queue Scheduler. We profiled the time the centralized scheduler spent distributing tasks, and it takes roughly 99% of the total computation time for a batch. Therefore, we assume that the locks on the queues are too heavy for this data structure. We changed locking queues to lock-free queues [3]. The performance boosts to 1.3x speedup in a 4 thread scenario but drops to only 1x speedup when the thread count reaches 8. Therefore, we still believe that this is not scalable enough.

2.3.3 Pure Parallelism. In the end, we decide to remove the centralized scheduler, let each thread go through the whole operation sequence, and decide if they are taking the update task in the round-robin manner. All threads have to take every query task. For writing back query results, there is a 2-dimensional array to store the partial query result from each worker thread. That is, each thread can directly write to the global result array without any locking mechanisms.

This achieves a 1.3x speedup in a 4 threads scenario and around 1.3x speedup in an 8 threads scenario. This method achieves similar speedup even if the query percentage is large, and is also more scalable than the Lock-Free Queue Scheduler.

	2	4	8
Centralized	0.39x	0.32x	0.12x
Lock-Free	0.73x	1.31x	1.02x
Pure Parallelism	0.96x	1.32x	1.31x

Table 2: Task Parallelism Speedup (batch size: 262144, array size: 16777215)

The Pure Parallelism design is very throughput-oriented. We do not guarantee that the results for the queries will be ready until all the work in the whole batch is finished. This is a trade-off between the query latency and the overall throughput.

From the observations in Table 1 and Table 2, we can see that there is a potential for optimization when the update functions are the majority. This is also a common case in real-world applications in Spatial Indexing[6]. Therefore, we decide to investigate more into the BatchAdd optimizations.

3 Batch Add Optimizations

When we accumulate multiple Adds, we can come up with more efficient algorithms to distribute Adds to worker nodes with some load balancing tweaks.

3.1 Fine-grained Lock

The most intuitive approach is to add a lock for each element in the internal array. The pseudocode is listed as follows:

```
class FenwickTree {
    // ...
    lock_t locks[N + 1];

    void add(int x, int val) {
        ++x;
        for (; x <= N; x += x & -x) {
            locks[x].lock();
            bits[x] += val;
            locks[x].unlock();
        }
    }

    void batchAdd(Op ops[B]) {
        #pragma omp parallel for
        for (auto &op : ops) {
            add(op.index, op.val);
        }
    }
    // ...
}
```

For this approach to work, we introduce a function `batchAdd` to the Fenwick tree, so we can process and parallelize a batch of inputs. Although performance degradation can be expected due to the locking overhead and cache coherency, this approach is still included for comparison. Testing on GHC shows roughly a 0.13x slowdown when running on one thread with default settings, indicating significant locking overhead. Also, the program does not show a performance gain when running with more threads.

3.2 Coarser-grained Lock

To lower the locking overhead, we implemented the add function with a coarser-grained lock mechanism as follows.

```
class FenwickTree {
    // ...
    lock_t locks[(N + L) / L)];

    void add(int x, int val) {
        ++x;
        // Update the tree
        locks[x / L].lock();
        int prev_x = x;
        for (; x <= N; x += x & -x) {
            if (prev_x / L != x / L) {
                locks[prev_x / L].unlock();
                locks[x / L].lock();
                prev_x = x;
            }
            bits[x] += val;
        }
        locks[prev_x / L].unlock();
    }

    // ...
}
```

However, after exploring various lock granularities, the program still has a slowdown of at least 0.18x on GHC machines when running with multiple threads. To identify the reasons for slowing down, we ran OpenMP on sequential version without any locking mechanism. Although this might provide invalid query results, it is the easiest way to observe the influence of cache coherency. This approach still has 0.75x slowdown compared to serial version, which means the locking approach can never work. Further analysis is conducted on GHC via `perf`.

Metrics	Serial	4 threads without lock
L1-dcache-loads	$3.23 \cdot 10^9$	$3.42 \cdot 10^9$
L1-dcache-load-misses	$8.66 \cdot 10^8$	$1.36 \cdot 10^9$
L1-dcache-stores	$2.75 \cdot 10^9$	$2.70 \cdot 10^9$
LLC-loads	$5.56 \cdot 10^8$	$6.76 \cdot 10^8$
LLC-load-misses	$6.89 \cdot 10^6$	$3.48 \cdot 10^6$
LLC-stores	$2.47 \cdot 10^8$	$8.47 \cdot 10^8$
LLC-store-misses	$6.28 \cdot 10^5$	$4.86 \cdot 10^5$

Table 3: Perf results of Serial and 4 threads without lock

Based on cache profiling, the L1-dcache miss rate of serial version and parallel version is 26.85% and 39.51%, respectively. In addition, the number of LLC stores in the parallel version is approximately three times higher than in the serial version. Thus, to reach an effective parallelism, it is important to remove or at least lower the memory contention among multiple threads toward the internal array.

3.3 Model Parallelism

3.3.1 Fixed-size partitioning. In contrast to the fine-grained lock approach that uses data parallelism, we attempted to implement model parallelism by partitioning the internal array into p roughly equal-sized subarrays, where p is the number of threads (Figure 1a). Each subarray is assigned to a thread, and each thread would only modify the assigned subarray. Similar to locking approaches, we focused on parallelizing `batchAdd` instead of `add` to lower the overhead of launching threads. The pseudocode of the function `batchAdd` to demonstrate the idea is provided as follows.

```
void batchAdd(Op ops[B]) {
    #pragma omp parallel
    {
        int t = get_thread_id();
        auto [lower, upper] = ranges[t];

        for (auto &op : ops) {
            int x = op.index + 1;
            int val = op.value;

            for (++x; x < upper; x += x & -x) {
                if (x >= lower)
                    bits[x] += val;
            }
        }
    }
}
```

By assigning each subarray to a single thread, there would only be some false sharing at the boundary of each subarray, highly reduce the cache contention across threads. In addition, running with multiple threads can utilize more L1 caches as the program is running on multiple cores. As shown in Table 4, the L1-dcache miss rate of the Model Parallelism approach is 15.67%, which is even lower than the serial version.

Metrics	Serial	Model Parallelism
L1-dcache-loads	$3.23 \cdot 10^9$	$3.78 \cdot 10^9$
L1-dcache-load-misses	$8.66 \cdot 10^8$	$5.93 \cdot 10^8$
L1-dcache-stores	$2.75 \cdot 10^9$	$2.69 \cdot 10^9$
LLC-loads	$5.56 \cdot 10^8$	$3.05 \cdot 10^8$
LLC-load-misses	$6.89 \cdot 10^6$	$1.49 \cdot 10^6$
LLC-stores	$2.47 \cdot 10^8$	$2.11 \cdot 10^7$
LLC-store-misses	$6.28 \cdot 10^5$	$1.43 \cdot 10^5$

Table 4: Perf results of model parallelism with 4 threads

As shown in Table 5, this approach (fixed-size without skipping) gained some speedup compared to the serial version, which takes roughly 1450 ms under the same settings. The purpose of Table 5 is to illustrate the intuition behind the development of our approaches. More robust experiments are presented in Sections 4 and 5.

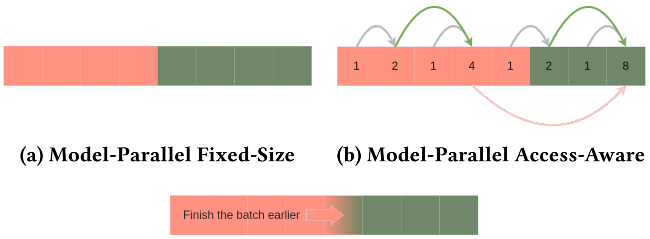
However, an issue is presented in this approach, which limits the speedup. Even though a range is assigned to each thread, each of them still follows `x += x & -x` until the index located in the range assigned is reached.

To address the issue, the following code is run before iterating through the loop. This series of bit operations computes the smallest index derived from `x` that lies within the target range in $O(1)$. After applying this optimization, the program has a decent speedup with 2 threads on GHC (Table 5). However, by profiling the runtime of each thread, we found out that the workload is not balanced among threads, which becomes more obvious when the number of threads increases (Figure 2a).

```
if (x < lower) {
    auto highest_diff_bit
        = (1ULL << 63) >> __builtin_clzll(x ^ lower);

    x |= highest_diff_bit;
    x &= ~(highest_diff_bit - 1);

    if (x < lower) {
        x += x & -x;
    }
}
```



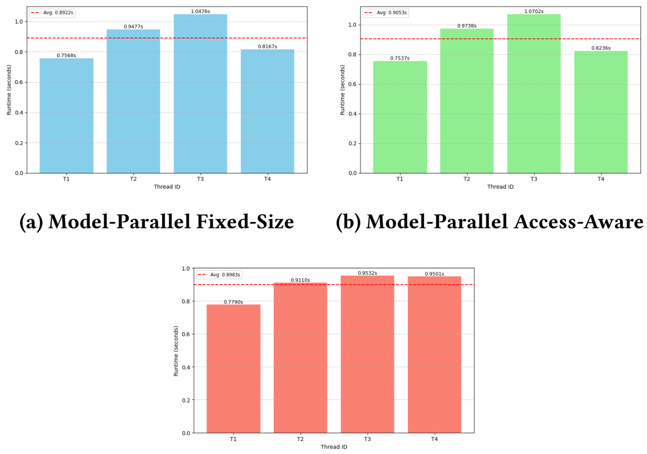
(a) Model-Parallel Fixed-Size **(b) Model-Parallel Access-Aware**

(c) Model-Parallel Semi-Static

Figure 1: Partitioning methods with 2 threads

3.3.2 Access-aware Partitioning. Our first attempt to address the workload imbalance is to analyze the access probability of each element based on the access patterns of the Fenwick tree, assuming uniformly random inputs. Then, the array is partitioned into p subarrays with roughly equal access probability, which would distribute the write operations equally across all threads. For example, in Figure 1b, the numbers indicate how many times each element would be accessed if each index is added once. In this case, the total access counts of the first partition and the second partition are 9 and 11, respectively. However, based on the profiling results, this approach does not lower the workload imbalance between threads (Figure 2b).

Therefore, the workload depends not only on the access counts of the subarray but also on how accesses are distributed within the subarray. For example, even if both subarrays have the same access counts, one might have accesses concentrated on a single element,



(a) Model-Parallel Fixed-Size **(b) Model-Parallel Access-Aware**

(c) Model-Parallel Semi-Static

Figure 2: Workload of Partitioning Methods

while the other's accesses are distributed across multiple elements. In this case, the first subarray may complete the write operations more quickly due to a more cache-friendly access pattern.

3.3.3 Semi-static Partitioning. It is hard to analyze the influence of cache and statically split the internal array based on that. Thus, instead of deciding the ranges of subarrays when initializing the Fenwick tree, we tune the ranges of subarrays after each batch. There are several ways to do so. One possible implementation is to calculate the execution time of each thread within a batch, allowing us to adjust the ranges accordingly. While this solution could estimate the workload on each thread precisely, it also introduces additional overhead from calling the timing function and adjusting the ranges after each batch.

Another solution is to make use of the OpenMP directives `#pragma omp single nowait` after the batch. The pseudocode is shown as follows.

```
void batchAdd(Op ops[N]) {
    #pragma omp parallel
    {
        int t = omp_get_thread_num();
        auto [lower, upper] = ranges[t];

        #pragma omp barrier
        for (auto &op : ops) {
            // operations same as previous
            // model-parallel approaches
        }

        #pragma omp single nowait
        {
            enlarge_range(ranges, t);
        }
    }
}
```

The block of the directive `single` would be run by the thread that first reaches that block. That is, the function `enlarge_range` would be run by the thread with less workload. The option `nowait` is used so that the first thread can proceed in parallel when other threads are still dealing with add operations, reducing the re-partitioning overhead. Note that `#pragma omp barrier` should be added to ensure the correctness when reading `ranges` and the accuracy when estimating the execution time. To minimize overhead, the function `enlarge_range` simply enlarges the assigned range and shrinks the neighboring ranges.

Based on profiling results, the semi-static approach highly reduced the difference in execution time among threads (Figure 2c). However, it still introduces some overhead, which might explain why the overall execution time doesn't improve a lot compared to fixed-size partitioning (Table 5). Additional results will be presented and discussed in later sections.

Approaches	p = 2	p = 4	p = 8
Fixed-size without skipping	1120	1052	1001
Fixed-size	741	559	474
Access-aware	731	581	520
Semi-static	725	541	462
Aggregate	898	659	519

Table 5: Execution time (milliseconds, with batch size: 65536, number of batches: 1024, and array size: 1048575)

3.4 Model Parallelism with Aggregation

So far, we have discussed fixed-size partitioning, access-aware partitioning, and semi-static partitioning for model-parallel approaches. However, all of those approaches focus on optimizing the partitioning algorithm. In this subsection, instead of processing each input in a batch one by one, we developed an approach where each thread would "mark" the smallest involved index for each input on a local array. Then, the values would be propagated to the full subarray all at once. The overall logic is presented in the following pseudocode.

```
int bits[N + 1];
int local_bits[N + 1];

void batchAdd(Op ops[B]) {
    #pragma omp parallel
    {
        for (auto &op : ops) {
            x = find_smallest_index(x, lower, upper);
            if (x < upper)
                local_bits[x] += val;
        }

        for (x = lower; x < upper; ++x) {
            int next_x = x + (x & -x);

            if (next_x < upper) {
                local_bits[next_x] += local_bits[x];
            }
            bits[x] += local_bits[x];
            local_bits[x] = 0;
        }
    }
}
```

Compared to previous approaches, this approach has a more predictable pattern. The time complexity of the algorithm is $O(B + N/P)$, where B, N, and P are batch size, array size, and the number of threads, respectively. The second loop primarily consists of sequential reads and writes, leading to better memory access efficiency. Although the approach does not have a great performance under current settings (Table 5), we expect it to have better parallelism with a higher thread count. More results will be discussed in the later sections.

4 Experiments

We have implemented several algorithms and we only pick a few ones with potentially good results to do the experiments.

4.1 Experimental Setup

The Fenwick tree and microbenchmarking code are written in C++, which is hosted on GitHub[1]. The microbenchmarking is implemented to test various batch sizes, array sizes, and numbers of batches. The number of batches is adjusted to target a runtime of roughly 10 seconds, reducing the impact of timing overhead and improving measurement accuracy.

All the experiments are run on Carnegie Mellon University's GHC machine and Pittsburgh Supercomputing Center's bridges-2. The specifications of both kinds of machines are listed below:

4.1.1 GHC Machines.

- **CPU:** Intel Core i7-9700
  - Cores: 8 cores
  - Threads: 8 threads (Note: This model does not have Hyper-Threading)
  - Clock Speed: 3.00 GHz base, up to 4.70 GHz boost
- **Cache:**
  - L1 Data (per core): 32 KB
  - L1 Instruction (per core): 32 KB
  - L2 (per core): 256 KB
  - L3 (shared): 12 MB (Intel Smart Cache)
- **RAM:** 16 GB
- **Operating System:** Ubuntu 22.04
- **Compiler:** GCC 11.4.0 with `-O3 -fopenmp`

4.1.2 Bridges-2 RM Nodes.

- **CPU:** AMD EPYC 7742
  - Cores: 64 cores
  - Threads: 128 threads (SMT enabled)
  - Clock Speed: 2.25 GHz base, up to 3.40 GHz max boost
- **Cache:**
  - L1 Data (per core): 32 KB
  - L1 Instruction (per core): 32 KB
  - L2 (per core): 512 KB
  - L3 (shared): 256 MB (Shared across cores within chiplets)
- **RAM:** 256 GB
- **Operating System:** Red Hat Linux
- **Compiler:** GCC 13.2.1 with `-O3 -fopenmp`

Note that the 256 MB of L3 cache is not a single shared shared across all cores. On Bridges-2, every 4 cores within the same Core Complex (CCX) share a 16 MB L3 cache. As there are 16 CCXs (64 cores) per node, the total L3 cache amounts to 256 MB.

4.2 Input Generator

An input generator generates random operations for our Fenwick tree implementations. The sampling frequency of `add` and `query` is based on the query frequency specified. For instance, if query frequency is set to 0%, only `add` operations are generated. The index of the input is uniformly sampled over all valid indices defined by the array size of the Fenwick tree. Since the value of an operation doesn't impact the speed, we simply sample a random integer value between 1 and 100.

5 Results

5.1 Impact of Query Frequency on General Approaches

From Fig. 3, we can observe that Pure Parallelism is more resilient to constant queries compared to the Lazy Sync method in all three different array sizes. This is because the Lazy Synchronization method accumulates all updates until a query is met, and constant queries limits the accumulation for parallelism. Meanwhile, queries in Pure Parallelism do not cause any synchronization barriers as each thread writes the result to independent cells in the global result array.

Also, we can observe that when the query frequency is close to 0, the performance of Lazy Scheduling mostly exceeds the Pure Parallelism method. This is also due to the fact that the parallelism is nearly linear to the number of threads as the query frequency is very low. Yet Pure Parallelism still suffers from the overhead of letting all threads read through all workloads. This observation encouraged us to further work on optimizing batched adds for this scenario.

5.2 Impact of Array Size on Speedup

Three sets of parameters are used to study the relationship between speedup and array size. The array sizes are selected based on the cache hierarchy of the GHC machines. The first array size, 4095, fits within the L1 cache; the second, 2,097,151, fits within the shared L3 cache; and the third, 16,777,215, fits only in RAM. Note that the internal array in our implementation uses the type `int`, so those three array sizes correspond to 16KB, 8MB, and 64MB, respectively. The batch size is fixed at 262,144. The results are presented in Figure 4 and 5.

5.2.1 General Optimizations. For general approaches, query frequency is set to 0 to provide a more fair comparison with the BatchAdd optimizations.

Pure Parallelism achieves great speedup in both small arrays that can fit in L1 cache entirely and large arrays that can only fit in RAM. For small arrays, Pure Parallelism performs better than the sequential one due to near-linear parallelism with low overhead

[1] https://github.com/EricTsengTy/ParallelFenwickTreeImpl

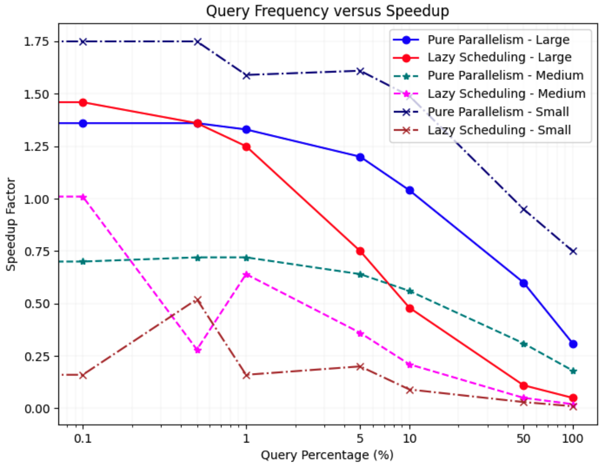


Figure 3: Speedup comparison versus Query Frequency

and avoiding excessive cache coherency traffic at the same time. For large arrays, the overhead of letting all threads iterate through all operations decreases as the total memory accesses increases, which therefore lead to at most 6x speedup with 128 threads.

Lazy Synchronization has a performance grow as the array size become larger. This is because as the array size grows, the number of memory access per operation grows and therefore the overhead of the OpenMP framework is relatively reduced. There is also a performance drop when the array size is small; this is due to the excessive cache contention and the overhead from the atomic array operations.

- GHC: Pure Parallelism suffers from a performance drop from 4 threads to 8 threads due to the fact that the memory consumption from all local Fenwick Trees exceeds the L3 cache; therefore, constantly evicting and loading cache lines from L3 and RAM significantly reduced the performance. The rest of the results are expected from the general approaches.
- Bridges-2: We can observe that there is usually a performance drop when thread count reaches 8 or 16, this might be accounted to the CPU design from the PSC machines. AMD EPYC 7742 groups 4 cores into 1 CCX, and communication between CCXs are slower than within a CCX. However, when the number of threads surpasses 32, the benefits of parallelism overwhelms the impact on this, and therefore the speedup grows again.

5.2.2 BatchAdd optimizations. Since the BatchAdd optimizations target a more limited use case, it generally performs better than the two general approaches. The Model-Parallel Aggregate approach has different characteristics compared to the other three, so they are discussed separately.

Model-Parallel Fixed-Size, Access-Aware, and Semi-Static approaches have a better speedup with medium arrays compared to the other two on GHC Machines. The poor speedup with small arrays is related to the limitation of the algorithms. Our approach benefits from multiple threads only when an update modifies multiple indices across different partitions, which becomes less likely when the array size is small. On the other hand, a different reason explains the poor speedup of these approaches when the array size is too large. When the array size exceeds the L3 cache, many reads and writes on the internal array must access RAM. As a result, the L3 cache miss rate grows from 3% to around 50% when the array size changes from 2,097,151 to 16,777,215. As the `BatchAdd` operation has low arithmetic intensity, the high miss rate makes RAM access a bottleneck. Thus, our approaches achieve poor speedup due to the limited bandwidth of RAM.

The main difference when running on Bridges-2 is that it still reaches great performance with the large array. This could be related to the larger L3 cache size on Bridges-2. It has at most 256 MB of L3 cache when fully utilized, which is enough to accommodate the entire array. Without being limited by the bandwidth, these approaches can reach at most 11x speedup on Bridges-2.

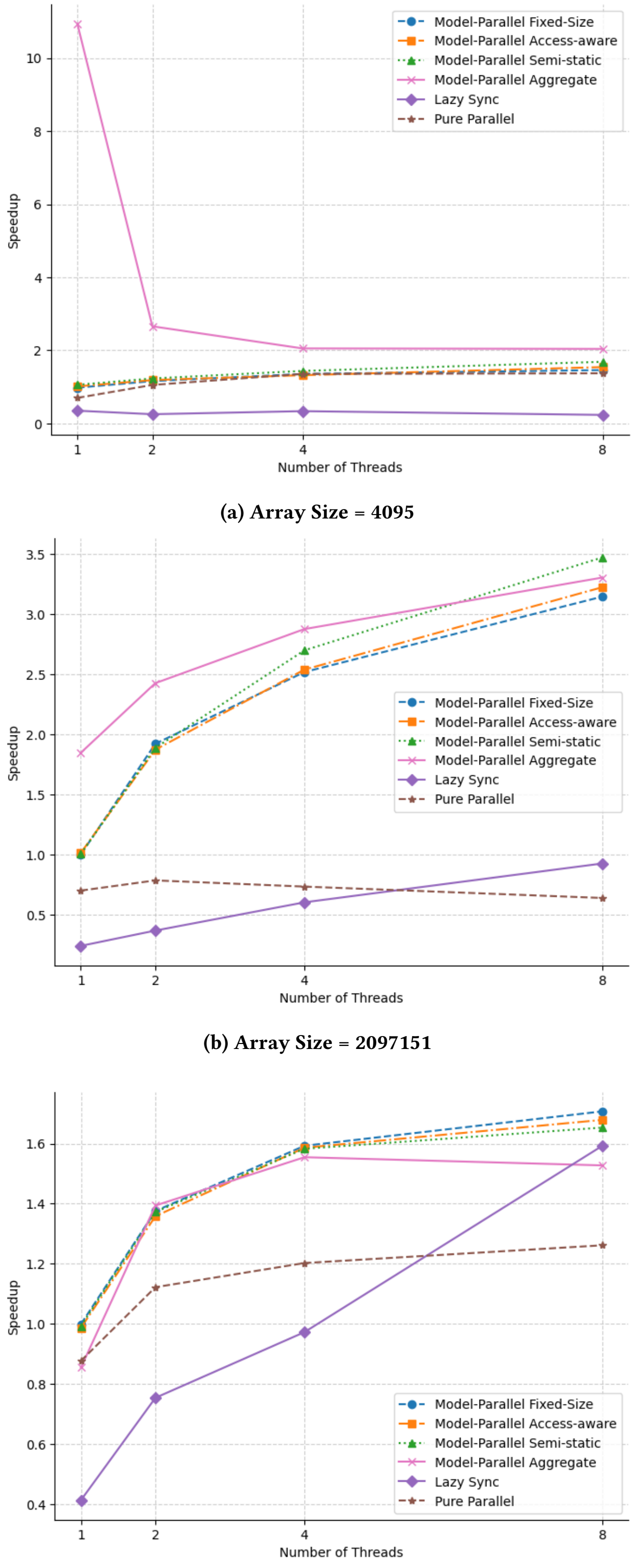
The Model-Parallel Aggregate approach has very different behaviors when the array size grows. With a small array, the approach cannot benefit from adding more threads, as the runtime is dominated by the batch size. In addition, adding more threads might even hurt performance by increasing overhead and communication costs. However, the approach achieves great speedup in other cases. It outperforms other approaches with medium and large arrays on the Bridges-2 machines. As mentioned in Section 3.4, the second loop of the approach consists almost entirely of sequential reads and writes, reducing memory overhead. This may explain its superior performance with larger arrays and higher thread counts. More analysis of the Model-Parallel Aggregate approach is provided in the next section.

5.3 Impact of Batch Size on Speedup of Batch-Add function

The `Batch-Add` function is introduced to lower the overhead of launching threads and eliminate the need to maintain consistency between updates and queries. To study the relationship between batch size and speedup, we run experiments on all Model-Parallel approaches with various batch sizes. In this set of experiments, the array size and number of threads is fixed at 2097151 and 4. In addition, all the speedup is computed relative to the serial Fenwick Tree mentioned in Section 2.1.

In Figure 6, the speedup of Model-Parallel Fixed-Size, Access-Aware, and Semi-Static approaches improves as the batch size grows and quickly converges when the batch size exceeds 1024. The diminishing effect is expected, as the OpenMP overhead accounts for a smaller portion of runtime as the batch size increases.

On the other hand, the Model-Parallel Aggregate approach shows poor speedup when the batch size is too small (Figure 6). This is because, compared to the serial version's time complexity of `Batch-Add`, which is $O(B \lg B)$, the Model-Parallel Aggregate approach has a time complexity of $O(B + N/P)$. When $N/P \gg B$, although the approach scales well when the number of threads increases, it still has poor speedup due to the algorithm's limitations. However, when $N/P \ll B$, although the Model-Parallel Aggregate approach runs significantly faster than the serial version, it does not scale well because the runtime is dominated by the batch size

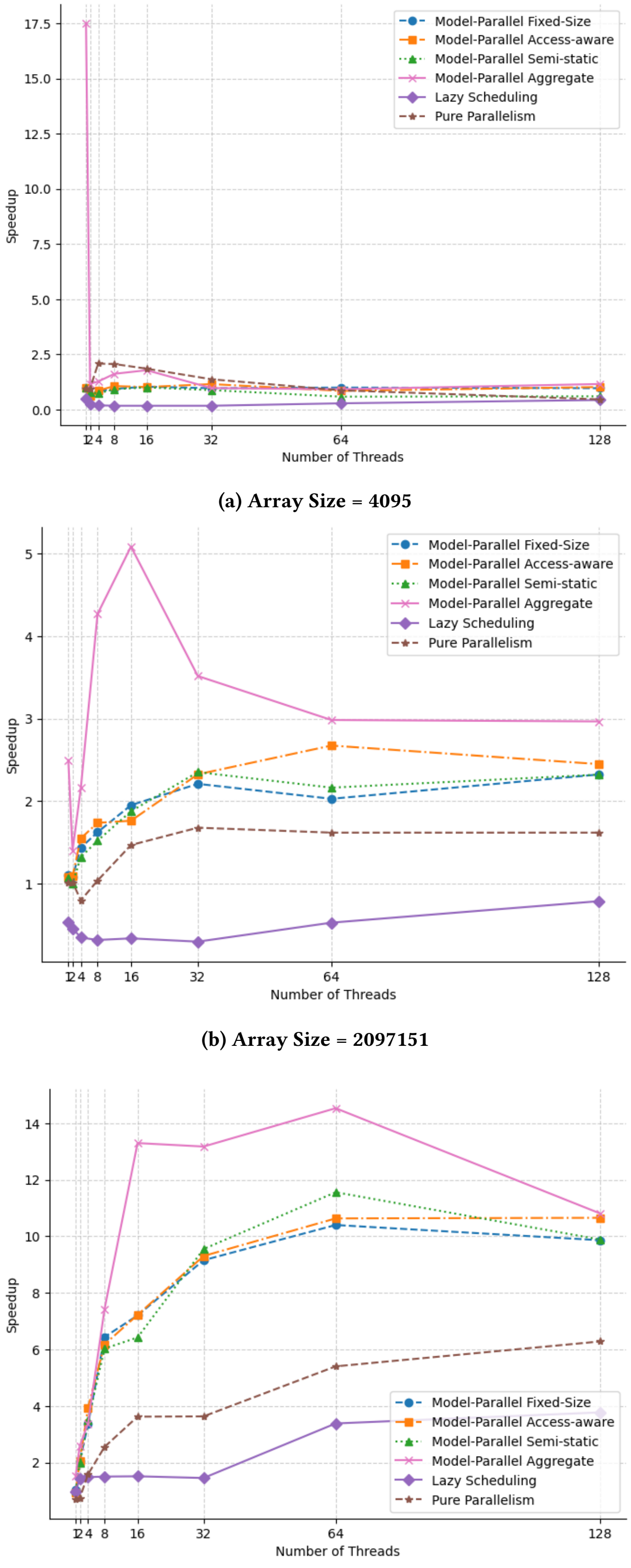


(a) Array Size = 4095

(b) Array Size = 2097151

(c) Array Size = 16777215

Figure 4: Speedup comparison for different approaches across threads for various array sizes on GHC.



(a) Array Size = 4095

(b) Array Size = 2097151

(c) Array Size = 16777215

Figure 5: Speedup comparison for different approaches across threads for various array sizes on Bridges-2.

B (Table 6). Thus, the approach scales well only when $N/P \approx B$, as observed in Figure 4b (with $N = 2^{21} - 1, B = 2^{18}, P = 2^3$) and Figure 5c (with $N = 2^{24} - 1, B = 2^{18}, P = 2^6$).

Approach	p = 1	p = 2	p = 4	p = 8
Serial	447.696	x	x	x
Model-Parallel Aggregate	51.262	69.824	85.919	95.536

Table 6: Execution Time (microseconds) of Model-Parallel Aggregate with array size = 2097151 and batch size = 16777216 on GHC

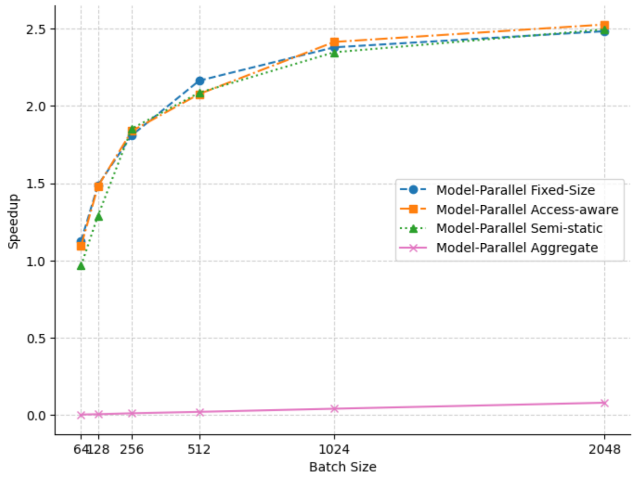


Figure 6: Speedup comparison for different approaches across threads for various batch sizes

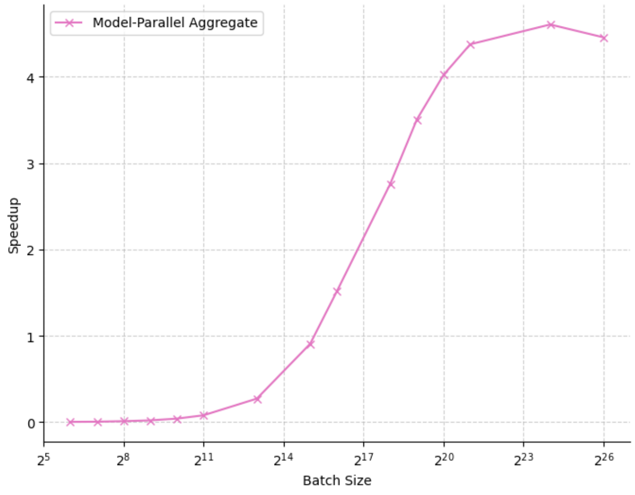


Figure 7: Speedup comparison for Model-Parallel Aggregate across threads for various batch sizes

6 Conclusion

The effectiveness of Parallel Fenwick Tree optimizations heavily depends on workload characteristics and access patterns—there is no one-size-fits-all solution. It is both workload-dependent and machine-dependent. One should design the benchmarks based on the required workload and run on their machines to really test out which algorithm works better.

For general workload algorithms, Pure Parallelism excels in scenarios with more randomly distributed update/query operations, offering better performance than Lazy Synchronization. Lazy Synchronization shows greater potential for optimization in workloads with mostly updates. Also, optimizations on general apporaches only work well when the array size is either small enough to fit in the L1 cache or big enough to only fit in RAM. They also demonstrate different performance characteristics in GHC and PSC machines. We conclude that machine speculations should also be considered when choosing the right algorithm.

When batched additions (BatchAdd) are feasible, Model Parallelism can achieve superior speedups, outperforming general approaches. However, no single approach consistently outperforms the others. A hybrid approach may be implemented in practice. When $B \approx N/P$, the Model-Parallel Aggregate achieves better speedup compared to the others. When the array size is much smaller than the batch size ($N \ll B$), the single-thread Model-Parallel Aggregate approach may be the fastest. In all other cases, Model-Parallel Semi-Static is likely the best choice due to its well-balanced workload.

For future work, we would like to combine the lazy Synchronization algorithm with Model-Parallelism. We believe that this can help Lazy Sync to achieve an even higher speedup.

References

[1] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. 2022. Joinable Parallel Balanced Binary Trees. *ACM Trans. Parallel Comput.* 9, 2, Article 7 (April 2022), 41 pages. doi:10.1145/3512769

[2] CP-Algorithms. 2024. Fenwick Tree (Binary Indexed Tree). https://cp-algorithms.com/data_structures/fenwick.html

[3] Cameron Desrochers. 2024. concurrentqueue: A fast multi-producer, multi-consumer lock-free concurrent queue for C++. https://github.com/cameron314/concurrentqueue.

[4] Peter M. Fenwick. 1994. A new data structure for cumulative frequency tables. *Softw. Pract. Exper.* 24, 3 (March 1994), 327–336. doi:10.1002/spe.4380240306

[5] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 743–754. doi:10.1145/2588555.2610507

[6] Jens Schneider and Peter Rautek. 2017. A Versatile and Efficient GPU Data Structure for Spatial Indexing. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 911–920. doi:10.1109/TVCG.2016.2599043

[7] Peter Su and Scot Drysdale. 1992. *Building Segment Trees in Parallel.* Technical Report PCS-TR92-184. Dartmouth College, Computer Science Technical Report. https://digitalcommons.dartmouth.edu/cs_tr/78