# Implementing Arbitrary/Common Concurrent Writes of CRCW PRAM

Fady Ghanim
Oak Ridge National Laboratory
Oak Ridge, United States
ghanimfa@ornl.gov

Wael R. ElWasif
Oak Ridge National Laboratory
Oak Ridge, United States
elwasifwr@ornl.gov

David E. Bernholdt
Oak Ridge National Laboratory
Oak Ridge, United States
bernholdtde@ornl.gov

**Abstract**

The Parallel Random Access Machines (PRAM) abstraction is the simplest and most elegant algorithmic model for the design and analysis of parallel algorithms. It consists of different models categorized based on the underlying memory access mode used, the most powerful of which is the Concurrent Read Concurrent Write (CRCW) model. A PRAM algorithm describes a series of rounds, each of which consists of a collection of operations that can be executed concurrently within the same time step. However, the lack of support for concurrent memory accesses and the prevalence of asynchronous programming models led to the belief that implementing CRCW PRAM algorithms is unattainable and prompted many to avoid this model except for theoretical studies of optimal performance.

In this work, we study the arbitrary and common concurrent writes in the CRCW PRAM model and explore implementation challenges on general-purpose systems. Moreover, we examine current practices for implementing common/arbitrary concurrent writes and propose a new efficient lightweight and thread-safe method to implement concurrent writes through leveraging atomic instructions. To demonstrate the efficacy of our method, we developed OpenMP kernels for classical CRCW PRAM algorithms and provide experimental results and comparisons based on run time performance measured over the x86 multicore architecture. Our results show a performance speedup compared to current practices up to 4.5x across all our benchmarks.

**Keywords**

CRCW PRAM, Parallel Algorithms, Arbitrary Concurrent Writes, Write-conflict resolution, Parallel Architectures

**Acknowledgements**

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

## Introduction

Parallel Random Access Machines (PRAM) is the most successful and well-studied parallel algorithmic model. The appeal of PRAM lies in its simplicity and abundance of techniques, making it the model of choice for designing and describing parallel algorithms. The Concurrent Read Con- current Write (CRCW) PRAM model provides a convenient and powerful parallel computational model to study the algorithmic complexity of parallel algorithms. However, the lack of underlying systems that support concurrent memory accesses, especially concurrent writes makes such a task extremely challenging. This work introduces methods lever- aging atomic instructions to implement arbitrary concurrent writes as specified in PRAM.

Many believe that CRCW PRAM algorithms are impossible to support on general-purpose parallel machines [5], due to PRAM's strict synchrony, its abstraction of the underlying hardware and the lack of support for concurrent memory writes in modern architectures. Parallel algorithms in PRAM are expressed as a sequence of time-step/rounds, each describing a collection of operations executed concurrently in lock-step over a shared memory, and in absence of any hardware constraints such as memory capacity, or the number of processors. Indeed, the current High-Performance Computing (HPC) landscape is dominated by asynchronous programming models that rely on programmers with intimate knowledge of the underlying heterogeneous architecture painstakingly fine-tuning applications to achieve higher performance computations, such as OpenMP, and MPI.

The trends above led to a decline in parallel algorithms research based on the PRAM model in general and the CRCW memory model specifically. General-purpose architectures follow an asynchronous Exclusive Read Exclusive Write (EREW) memory access model, or at best Concurrent Read Exclusive Write (CREW) memory access model. Hence, parallel algorithms research will either present a CRCW PRAM algorithm for best-case complexity analysis while also present another algorithm that follows the CREW to be implemented [20, 23], or will skip the PRAM model entirely in favor of an algorithm tailored towards a specific programming model and/or architecture.

In the last decade, recent research has refuted the claims of PRAM's infeasibility. An effort by Wu , et al. [22] introduced a new high-performance architecture based on the PRAM model called XMT. In later research on XMT, that group presented works using that processor showing higher performance than multicore and GPU processors [6, 7]. Ghanim, et al. [12] introduced a new parallel programming language called ICE and an accompanying compiler that takes a lock- step parallel program of a PRAM algorithm implemented as-is and translates it into a fork-join program that achieves comparable performance to hand-optimized parallel multi- threaded programs.

This work focuses on implementing arbitrary and common concurrent writes per the CRCW PRAM model and examines the possible hurdles towards realizing that on cur- rent asynchronous general purpose and exclusive memory accesses systems. We also examine how using atomic instructions can help in producing efficient implementation of arbitrary concurrent writes. Our goal in this work is to create an efficient implementation of concurrent writes that is usable by modern HPC hardware and applications, and that supports concurrent write for modern language data structures such as structure and class copies. It also needs to enable generic compiler approaches to translating high- level representations of concurrent writes in PRAM-based programming languages.

### *Contributions*

This work makes the following contributions:

- Define the criteria required by an efficient and safe concurrent write implementation to achieve common and arbitrary concurrent writes as specified in CRCW PRAM.
- Suggest a new efficient methodology to implement a generic arbitrary and common concurrent writes on general-purpose machines through leveraging atomic instructions strategically for a lightweight high-performance solution.
- Introduce a new generic wait-free API based on the suggested method and is compatible with the OpenMP model and x86 multicore architecture.

### *Overview*

Section 2 gives an overview of background information on the PRAM algorithmic model. Section 3 reviews related work that tackled the issue of implementing concurrent writes. Section 4 discusses the requirements of concurrent writes and lists the criteria for a proper solution. Section 5 details our implementation of concurrent write and compares it to currently available methods. The asymptotic bounds analysis is presented in Section 6. Section 7 presents the experimental environment and discuss our results. Conclusion and future work are presented in Section 8.

**Background on PRAM**

Parallel Random Access Machine (PRAM) is a parallel algorithmic abstraction for shared-memory systems and is the parallel analog to the random access machines. PRAM makes certain assumptions about the underlying architecture such as having unlimited shared memory and processors, and that any access to any part of the shared memory can be completed within unit time. PRAM describes an algorithm as a series of rounds/time-steps within which multiple operations are executed concurrently in lock-step. This allows PRAM to abstract several issues complicating algorithm design, such as; synchronization, communication, work-sharing. This made PRAM an attractive model for the study and design of parallel algorithms and consequently became the dominant parallel algorithmic theory for over two decades, producing a rich variety of algorithms spanning a wide range of applications.

A PRAM algorithm is characterized by two metrics; the Work and Depth of an algorithm. The work refers to the total number of operations performed by an algorithm, whereas depth refers to the number of time steps of an algorithm on the critical path. PRAM defines a variety of memory access modes regarding the exclusivity of memory access for reads or writes. The primary strategies are:

- **Exclusive Read Exclusive Write (EREW)** Only one processer can access a memory cell at a time whether it is for read or write.
- **Concurrent Read Exclusive Write (CREW)** Multiple processors can read a memory cell simultaneously, however, only one processor can write to it at a time.
- **Concurrent Read Concurrent Write (CRCW)** Multiple processors can read and write at the same time. If reads and writes coincide within the same time step, reads always happen before writes.

Based on the above categorization, general-purpose architectures either follow the EREW or the CREW memory access model. It is worth noting that in all these modes if a concurrent read/write is attempted in an exclusive read/write mode, the algorithm fails.

In CRCW PRAM, when multiple processors attempt to write to a memory cell concurrently, there are various conflict resolution strategies governing which processor gets to commit a write to a memory cell. These strategies have varying degrees of power determined by their ability to describe PRAM algorithms while providing better asymptotic complexities. Furthermore, a weaker strategy can be simulated by a more powerful one in O(1) time. The primary rules are (listed from strongest to weakest):

- **Priority CRCW** Only the processor with the highest priority can write, the rest fail. Processor priority is determined based on some attributes such as, processor with minimum processor rank has the highest priority, or processor writing the smallest value has the highest priority.
- **Arbitrary CRCW** An arbitrary processor succeeds in committing its write, the rest will fail. There is no specification which processor wins.
- **Common CRCW** All processors will try to write the same value to the same memory cell, concurrently.

**Related Work**

In this section, we describe the previous studies on the relationship among the different PRAM memory access models. We will pay close attention to simulations or implementations going from the concurrent write PRAM to exclusive write PRAM. To the best of our knowledge, we have not found any related work that enables safe and efficient implementation of concurrent writes on general-purpose asynchronous architectures.

The study of simulations between the different PRAM models has received immense attention in research. Generally, this has taken the form of studying comparisons of time complexity of simulating the execution of one step of a PRAM model using another. It is a well-known fact that simulating a concurrent write of Priority CRCW PRAM on any of the exclusive write PRAM requires time $T$ $(logP)$, where P is the number of processors required by the algorithm [16]. Being the most powerful of the CRCW PRAM models, this result was used as the upper limit for simulating all other concurrent write PRAM models on exclusive write PRAM. Further studies of simulations between different models of PRAM has largely been between models with similar memory access mode, but different write conflict resolution rules [2–4, 9–11, 13–15, 18]. It is worth noting, PRAM algorithms tend to assume a number of processors asymptotic to problem size while general-purpose architectures have a limited number of processors that can be aptly described as asymptotically constant.

While we are not aware of an official treatment of the topic at hand, there are two instances where concurrent writes were implemented or discussed casually; The first in- stance is as a part of the OpenMP implementation of Breadth-First Search in the Rodinia benchmark ver. 3.1 [17], where a common concurrent write was implemented naively by issuing the concurrent writes and relying on the underlying architecture to queue and commit all concurrent writes. The downside of this method is that it is prone to race conditions, especially in the case of arbitrary concurrent writes. Furthermore, it will cause a large number of cache updates and invalidations, and unnecessary bus congestion. The second instance [21] is as part of a discussion of the XMT programming model, where they discuss a method for implementing concurrent writes using a specialized prefix sum unit to perform the prefix sum operation over a gatekeeper variable and select one thread based on the result to perform the write. However, this method is limited in scope to concurrent writes to scalars. In the case of concurrent writes to multiple memory locations within the same step, they suggest performing prefix sum to a gatekeeper array of size equal to the number of memory locations to be written. However, this requires reinitialization of the gatekeeper variables before every re- write to the assigned memory location. We will discuss the advantages and limitations of both methods in detail in Section 5, and we will provide performance results based on both methods in Section 7.

**Arbitrary Concurrent Write Implementation Challenges**

In this work, we suggest strategies to perform efficient arbitrary/common Concurrent Writes (CW) while maintaining the correctness of the intended semantics of both strategies as specified in PRAM. In this section, we explore the design requirements to realizing this goal and investigate ways to achieve the intended semantics of concurrent write on the asynchronous threaded model.

The differences between the PRAM abstraction model and actual machines must be considered before a selection algorithm is designed. The PRAM model follows a lock-step execution model, hence all PRAM processors that need to write to memory will arrive at the same concurrent write step, pick a winner processor from amongst them to commit its write, and then commit the write, all synchronously within the same execution time-step/round. On the other hand, in the threaded model, each thread progresses at its own pace, which may lead to a scenario where one thread progresses to a read dependent on a concurrent write from a previous time-step long before the write has been performed. However, Ghanim , et al. [12] have shown that one can essentially use work sharing methods to distribute the work on the small number of physical processors available and disregard PRAM's lock-step semantics, as long as synchronization points are introduced strategically between inter-parallel context dependencies. This method also helps partially with any data dependence caused by the lack of synchronization. However, synchronization points will not help in resolving problems associated with race conditions resulting from multiple threads attempting to write to memory simultaneously. This is an important concern that must be addressed as part of any proposed concurrent write implementation.

Race conditions have different effects on the correctness of a write operation depending on the resolution mechanism in use, and the number of memory transactions required to complete a write (e.g., writing an integer vs copying a structure). Race conditions do not affect the correctness of a common CW since the same value is being written regardless of the thread that committed the write. In this case, it does not matter whether there is an overwrite, or different threads wrote different parts of a multi-transaction memory update. As such, naively allowing all writes to happen and relying on the underlying memory HW to resolve this should work, albeit with some performance degradation due to serialization as a result of queuing memory writes. However, this is not the case for arbitrary CW where different threads are trying to write different values. This is especially important when a concurrent write may require multiple memory transactions to complete. Otherwise, race conditions may produce a

structure that does not match any of the ones being written. One of our fundamental guiding principles in this work is for our solution to create an efficient implementation of concurrent writes that is usable by modern HPC hardware and applications, and that supports concurrent write for modern language data structures such as structure and class copies. A trivial but bad solution to this problem is to encapsulate the arbitrary CWs within a critical section, which will cause massive performance degradation.

A very important observation that is crucial for any high-performance implementation of CW is that only one write is ever observed by a dependent read, regardless of the number of concurrent writes attempted at a memory cell. This can be seen in the different conflict resolution strategies discussed in Section 2, where once a processor satisfies a specified condition, it can perform its write while other processors skip the write operation. For priority CW, the condition is the processor with the highest priority. However, in arbitrary and common CW, while any processor can write, only one processor succeeds. Consequently, the logical solution is to actively pick a winner thread to commit its write, while all other threads should skip the write operation for the current time step.

**Implementing Arbitrary Concurrent Writes**

In this work, our goal is to design a high-performance method for implementing concurrent writes on multi-core architectures. In this section, we describe how we achieve that goal while satisfying the requirements we established previously in Section 4, for a safe and efficient concurrent write implementation.

Our approach leverages atomic instructions to pick a thread to commit its write from among all the threads competing to perform a concurrent write to a specific shared memory address. The atomic operation enforces ordering between concurrent threads based on the order it is executed by each, and this can be used to pick a 'winner' thread to perform the concurrent write. The thread that successfully executes the atomic instruction is allowed to perform the write, while the remaining threads will skip it. This requires an additional single memory location to serve as the target of the atomic instruction per each target memory region of a concurrent write.

In our proposed method, we utilize the *compare-and-swap* atomic instruction to implement a *Compare-And-Swap-if-Less-Than (CAS-LT)* operation, which we use to perform the operation discussed above. Figure 1 illustrates our selection method, and Figure 3(a) illustrate an example showing how it is used. Our suggested method works as follows; any thread that is going to perform a concurrent write to a memory cell $M$ will call `canConWriteCASLT()` before attempting the concurrent write, and will only do so if the function returns **true**. `canConWriteCASLT()` requires two pieces of information to determine whether a thread can proceed or not: 1- 'lastRoundUpdated' which is the address of the auxiliary memory used as a target of the atomic instruction to coordinate between threads.' `lastRoundUpdated`' will contain the ID of the last round when a concurrent write was performed at the corresponding target memory $M$. And 2- 'round' which is a unique ID for a concurrent write execution step. Different concurrent write time steps should have different values of `round`, as such, round should be incremented before performing any new subsequent concurrent writes.

When there is a concurrent write, multiple writer threads compete to gain access to the same memory location, and all threads will check first if the current round's concurrent write has been performed (line 6). If 'lastRoundUpdated' is equal to round, then the check fails, and the thread will skip the execution of the atomic instruction, `canConWriteCASLT()` will return **false**, and this thread will skip the write. However, if 'lastRoundUpdated' has a value less than round, then the write has not been done yet for the current concurrent write execution step. All threads where the check succeeds now will compete to update 'lastRoundUpdated' to current round, to indicate to all threads that arrive later that the write should be skipped. Only one thread will be successful, while the rest will fail. The thread that succeeds is now allowed to perform the concurrent write, while all other threads will skip over it. It bears repeating that a synchronization point is required before any subsequent dependent read.

Now to compare our proposed method to the method based on using the prefix sums as suggested in [21], we look at Figure 3 which shows the BFS algorithm implementation using our proposed method (Figure 3(a)) and the prefix sum method (Figure 3(b)). We immediately notice two major differences: 1- On line 22 of both figures, we use `canConWriteCASLT()` for our proposed method, and `canConWriteAtomic()` for the prefix-sum method. While we

skip the atomic instruction once we have a winner thread in our proposed method as shown in Figure 1, in the prefix-sum method, shown in Figure 2, threads continue to execute the atomic instruction long after a winner thread has been determined. This will lead to serialization of parallel code if used by a multicore. However, this can be mitigated by skipping the atomic operation, once the gatekeeper variable is no longer equal to 0. 2- With every new instance of concurrent writes to the same memory cell, which in this example, happens every new iteration of the while

loop, the prefix sum method requires a re-initialization round of the `gatekeeper` array to indicate the start of a new concurrent write round (Figure 3(b) - line 34 - 35). However, in our proposed method we only need to increment round to achieve the same result (Figure 3(a) - line 33). Further- more, round could be substituted by the loop iteration, achieving the same result for free. It is worth noting that in `canConWriteAtomic()`, the atomic increment can be replaced by any other atomic operation including CAS, and we would still face the same problems discussed above, namely serialization due to atomic instruction, and the need to reinitialize the gatekeeper for every new concurrent write round.

**Theoretical Asymptotic Bounds**

In this section, we will examine the asymptotic bounds in terms of time and space requirements for the different methods for implementing concurrent writes. This examination will be limited to the time and space cost of the concur- rent write step, as compared to the costs per the arbitrary CRCW PRAM model. Throughout this section, the Work- Depth (WD) model will be used to express algorithmic complexity.

There is an important difference between the PRAM model and physical machines with respect to the number of processors. The PRAM model assumes a number of processors that are asymptotic to the problem size1, while the number of physical processors available on modern multicore machines is much smaller and can be aptly characterized as asymptotically constant. For the following discussion let $P_{PRAM}(N)$ denote the number of processors as specified by a PRAM algorithm and let $P_{Phy}$ denote the number of physical processors (cores) available in the system executing the algorithm.

Due to its importance to the following discussion we list Brent's Scheduling Theorem:

$$T(N) = D(N) + \frac{W(N)}{p}$$

Where $T(N)$ is the time bounds. $D(N)$ is the algorithm's depth, and $W(N)$ is total work. $p$ is number of processors used to execute the algorithm.

First, we examine the asymptotic bounds of an arbitrary concurrent write step as specified in CRCW PRAM. Let us consider a problem of size $N$ which has $P_{PRAM}(N)$ processors each performing a concurrent write step to one of $[M_i : 1 \leq i \leq K]$ potential shared memory targets. We consider the worst-case scenario, which is when all $P_{PRAM}(N)$ Processors are attempting to concurrently write to a single memory location $M$. According to arbitrary CRCW PRAM model, all $P_{PRAM}(N)$ processors will attempt and issue the write, and only one will succeed. The write step requires work $W(N) = P_{PRAM}$ processors, this can be completed in 1 time step or $T(N) = O(1)$. For a machine with $P_{Phy}$ processor, $T(N) = \frac{P_{PRAM}(N)}{P_{Phys}}$.

Now, we examine the same scenario on asynchronous threaded model using the naive method where we rely on the underlying architecture to resolve the writes. These architectures tend to queue the writes of all $P_{PRAM}(N)$ processors to the same memory location, effectively serializing the writes. Hence, according to Brent's theorem, this results in time $T(N) = P_{PRAM}(N)$, which is equivalent to performing the concurrent write serially.

Next, We examine the scenario above for the prefix sum method illustrated in Figure 2. This method relies on using a single auxiliary memory location per each $M_i$ memory location as a target for the atomic operation, to resolve which processor can perform the write. As such, the memory requirements for the scenario above is $O(K)$. Moreover, since this method relies on using atomics, it will result in serializing all atomic updates to the same

auxiliary memory location. Hence, similar to the naive approach, this will require a time $T(N) = P_{PRAM}(N)$. Furthermore, this method requires a reinitialization of the auxiliary array used for conflict resolution, with a W-D requirements of depth $D(N) = O(1)$ , and work $W(N) = O(N)$.

Finally, we examine the same scenario for the CAS-LT method. Similar to the prefix sum method, this method requires a single auxiliary memory location per target memory for resolving the winner processor, resulting in a memory requirement of $O(K)$. While the write is not complete, any processor can access the atomic operation on line 7 in Figure 1. However, this is bounded by the number of physical processors $P_{Phys}$ . Once any processor completes the write, the rest of the $P_{Phys}$ processors will fail the atomic CAS at line 7, and continue execution. Then, any subsequent attempt to write will concurrently perform the check at line 6 and fail, thus skipping the atomic update. This results in time $T(N) = O(P_{Phy}) = O(1)$ due to the serialized execution of the resolution part, and $T(N) = \frac{P_{PRAM}(N)}{P_{Phys}}$ for the entire step, which is consistent with the ideal PRAM time cost for a machine with $P_{Phy}$ processor, as discussed earlier. Finally, unlike the prefix-sum method, the proposed method does not require reinitialization of the auxiliary array used for conflict resolution.

**Evaluation**

In this section, we present the results of our experiments com- paring our proposed implementation of concurrent writes to other similar methods. We go briefly over our experimental setup, then we review the execution time experiments.

*Experimental Setup*

We implemented three classic CRCW PRAM algorithms to test the proposed approach. The first algorithm is the basic maximum algorithm with depth $D(1)$ and work $W(N^2)$ . A listing of this algorithm is provided in Figure 4. The second algorithm is the Rodinia benchmark's implementation of Breadth-First Search (BFS) which carries out its common concurrent write using the naive approach discussed in Section 4. It is essentially the same as the algorithm in Figure 3, with minor changes. The third algorithm is the Awerbuch-Shiloach Connected Components (CC) algorithm [1], a variant of the oach-Vishkin algorithm [19] with simplified hookings decisions.

We use the OpenMP programming model to implement our benchmarks above. For each benchmark, we wrote one implementation then took any necessary steps to ensure the correctness of the implementation such as insert synchronization points between concurrent writes and their corresponding dependent reads. Then, we created three versions that only differ in how the concurrent write itself is handled. These versions are:

(1) Allow all writes to happen naively with no selection method, and instead rely on the underlying architecture along with the added synchronization point. We will refer to this as the 'naive' method.
(2) Use an atomic postfix increment operation to select the writer thread in a similar fashion to the method discussed in [21]. To this end, we used OpenMP's atomic capture directive to implement the atomic prefix sum, as listed in Figure 2.
(3) Our method using CAS-LT, listed in Figure 1. For our method, since currently there is no compare-and-swap atomic implementation in OpenMP, we used gcc's built-in atomic CAS to implement CAS-LT.

We run our experiments using nodes from the Andes system at Oak Ridge Leadership Computing Facility (OLCF) [8]. Each node contains two 16-core x86 processors and 256GB of main memory. The benchmarks were compiled using GCC V10.1.0 with optimization level -O3 and executed on a compute node using the x86 processors. Furthermore, we set the OpenMP runtime environments to active wait policy and enabled thread affinity policy.

*Results*

In this section, we will look at the results of our experiments with respect to the execution time performance of our method as compared to the other implementations of CW for each benchmark. Since each of our experimentation algorithms has its unique profile, we will evaluate each individually, and compare different CW implementations based on problem sizes, and the number of threads used to run the experiment. Any provided measurement of

execution time excludes all time spent in initialization code and any serial code that is not part of the main algorithm.

**Constant-Time Basic Maximum Algorithm** We begin by examining our experimentation results for the basic maximum algorithm. Figure 4 present the naive OpenMP implementation. In this algorithm, we compare all pairs of elements in `List` to find the maximum value. Each comparison is assigned a single PRAM processor that will mark the smaller element as such. Only the maximum value will win.

While inefficient, as we examine the algorithm we notice a number of things that makes it a perfect test case to understand the characteristic of any concurrent write solution: 1- It is a small and embarrassingly parallel algorithm that executes the same number of operations independently of the input or execution path. 2- It is an extreme case of concurrency, where the entire algorithm largely revolves around a large number of common concurrent writes at line 10.

We begin examining the experimental results by looking at Figure 5 which shows the execution time for the basic maximum algorithm for all CW implementations with respect to problem size. We notice that our implementation achieves a higher performance when compared to alternatives. As the list size grows, the difference in performance increases. Furthermore, we achieve a maximum 2.5x speedup when compared to the naive method, and a geometric mean speedup of x1.98 over the naive approach across all problem sizes. Unsurprisingly, due to the massive serialization of numerous atomic prefix sum instructions, this approach resulted in a geometric mean 0.58x speedup (*i.e.*, 1.72x slower) when compared to the naive approach.

Next, we look at Figure 6 which shows the execution time for the maximum algorithm for all the CW implementations based on the number of execution threads. Our implementation achieves lower execution times compared to the alternatives. We notice that as the number of threads increases the execution time difference increases, reaching a speedup of 1.8x at 32 threads. As concurrency increases, the possibility of collisions between the threads increases. In our approach, we mitigate this by recording when the concurrent write happened and if it did, skip the execution of the redundant concurrent writes and CAS atomic instruction entirely.

**Breadth First Search Algorithm** Next, we look at the experimental results for the BFS benchmark. We started with the Rodinia benchmark's OpenMP implementation to create our benchmarks. The original code uses the naive method for handling CW, and was left unmodified. For the non-naive versions, we only added the respective CW resolution methods, similar to Figure 3.

We begin by looking at Figure 7 and Figure 8, which demonstrate the execution time for BFS with respect to the number of edges and nodes in a graph, respectively. We observe that in both cases, our proposed method provided significantly lower execution time compared to the alternatives, for all graph sizes. Additionally, when compared to Rodinia's implementation which uses the naive approach, we achieve a maximum speed up of 3.04x for graphs of different edge numbers, and a maximum speedup of 2.31x for graphs of different vertices number. For geometric mean, we achieve 2.12x and 1.86x speedups respectively when compared to Rodinia. For the prefix sum case, we notice that while the performance is generally better than the naive method, the difference in runtime is minor.

Finally, we discuss the effect the number of threads has on the execution time of the different methods. Figure 9 shows that our approach achieves the best runtime performance when compared to the alternatives. Furthermore, we notice that the difference in performance increases as the number of threads increases, reaching a speedup of 2.24x when compared to the native implementation of BFS in Rodinia.

**Connected Components Algorithm** Next, we look at our experimental results for the Connected Components (CC) kernel. For this, we implemented the Awerbuch-Shiloach algorithm [1], which is a variant of Shiloach-Vishkin (SV) algorithm with simplified hookings decisions. The SV algorithm (and its variants) are still considered one of the best algorithms for finding CC, and is the basis for many other graph algorithms. Since this algorithm requires the arbitrary CRCW PRAM model, we did not implement a naive approach for CC because this algorithm concurrently writes updates to multiple arrays during the hooking stage, rendering the naive method an unsafe approach due to potential race con- ditions as discussed in Section 5.

We begin by looking at Figure 10, which shows the execution time for CC with respect to the number of edges in a

graph. We notice that our method achieves superior performance when compared to the prefix-sum method. When compared to prefix sum we achieve a maximum speed up of 4.51x, and 4x geometric mean.

Moreover, we notice that as the number of edges increases, so does the performance gap. This is because the concurrent writes in the implemented algorithm are performed through parallelizing across all edges to perform the hooking step and mark the vertices pair as connected. So, by increasing the number of edges, the number of collisions between concur- rent writes increases, which in turn increases serialization for the prefix sum method. We can verify this by checking Figure 11, which shows the effect the number of vertices has on the execution time for the prefix sum and CAS-LT methods. We observe that for our method, the execution time trends very slightly upwards. However, the execution time for prefix sum is decreasing at a larger rate. This is because as the number of nodes increase, the "density" of edges per node decreases. This in turn means there are fewer sources of write collisions, which in turn benefits the prefix sum approach.

Figure 12 shows the relationship between the number of threads and execution time. We notice that our method is superior to the prefix sum method for different numbers of execution threads. Furthermore, we notice that as the number of threads increase, our execution time decreases.

**Conclusion**

In this work, we investigated the implementation challenges of CRCW PRAM arbitrary and common concurrent writes on general-purpose multicore architectures and presented a new generic, efficient and thread-safe method for implementing arbitrary concurrent writes using atomic instructions. To demonstrate the viability of our method, we developed OpenMP kernels based on classical CRCW PRAM algorithms and compared the run-time performance of the different methods as measured on the x86 multicore architecture. We show that our proposed solution achieved superior performance when compared to other methods for implementing concurrent writes with performance speedup between 2x to 4.5x across all our benchmarks.

This work represents an important step towards resolving an obstacle in bringing PRAM-based implementations to- ward general-purpose machines. As such, this work presents new research opportunities for implementing CRCW PRAM on general-purpose architectures, such as extending this work through exploring the implementation of concurrent writes on GPUs using atomic and shuffle instructions. Another possibility is to study the performance comparisons of EREW or CREW PRAM algorithms-based implementations currently in use, against relevant implementations of CRCW PRAM algorithms with better Work-Depth asymptotic complexities.

```
1  lastRoundUpdated : Last round the associated cell was updated
2  round: Current write round
3  atomic_cas(memory, old, new): Compare-exchange; return true if successful

4  inline bool canConWriteCASLT (unsigned &lastRoundUpdated, unsigned round) {
5      bool x = false;
6      If ((unsigned current = lastRoundUpdated) < round)
7          x = atomic_cas (lastRoundUpdated, current, round)
8      return x;
9  }
```

**Figure 1: Concurrent write check using CAS-LT.**

```
1  gatekeeper = 0: Auxiliary memory to track if concurrent write completed

2  inline bool canConWriteAtomic (unsigned &gatekeeper) {
3      unsigned x = 0;
4      #pragma omp atomic capture
5      {
6          x = gatekeeper;
7          gatekeeper++;
8      }
9      return x == 0;
10 }
```

**Figure 2: Concurrent write check using atomic instructions.**

```
1  #define M num_edges
2  #define N num_vertexes

3  unsigned V[N]; // index of first edge in E
4  unsigned E[M]; // ID of the destination vertex
5  unsigned Parent[N] = {-1}; //Parent of current node
6  unsigned sel_edge[N] = {-1}; //Index of the corresponding edge
7  unsigned  RoundWritten[N] = {0};

8  Void BFS () {
9      unsigned L = 0;
10     Level[source] = 0;
11     bool done = false;
12     while (!done) {
13         done = 1;
14         #pragma omp parallel for
15         for (unsigned v = 0; v < N; v++) {
16             if (level[v] == L){
17                 unsigned begin = V[v];
18                 unsigned end = V[v+1];
19                 for (unsigned j = begin; j < end ; j++) {
20                     unsigned u = E[j];
21                     if (!visited[u]) {
22                         if (canConWriteCASLT(&RoundWritten[u], L+1) {
23                             Parent[u] = v;
24                             Sel_edge[u] = j;
25                             Visited[u] = 1;
26                             Level[u] = L+1;
27                             Done = false;
28                         }
29                     }
30                 }
31             }
32         }
33         L++; //update round ID
34     }
35 }
```

(a) BFS algorithm implemented using Atomic CAS

```
1  #define M num_edges
2  #define N num_vertexes

3  unsigned V[N]; // index of first edge in E
4  unsigned E[M]; // ID of the destination vertex
5  unsigned Parent[N] = {-1}; //Parent of current node
6  unsigned sel_edge[N] = {-1}; //Index of the corresponding edge
7  unsigned gatekeeper [N] = {0};

8  Void BFS () {
9      unsigned L = 0;
10     Level[source] = 0;
11     bool done = false;
12     while (!done) {
13         done = 1;
14         #pragma omp parallel for
15         for (unsigned v = 0; v < N; v++) {
16             if (level[v] == L){
17                 unsigned begin = V[v];
18                 unsigned end = V[v+1];
19                 for (unsigned j = begin; j < end ; j++) {
20                     unsigned u = E[j];
21                     if (!visited[u]) {
22                         if (canConWriteAtomic(&gatekeeper[u]) {
23                             Parent[u] = v;
24                             Sel_edge[u] = j;
25                             Visited[u] = 1;
26                             Level[u] = L+1;
27                             Done = false;
28                         }
29                     }
30                 }
31             }
32         }
33         L++
34         #pragma omp parallel for
35         for (unsigned i =0; i < N; i++)        gatekeeper[i] = 0;
36     }
37 }
```

(b) BFS algorithm implemented using Atomic check

**Figure 3: Breadth First Search Implementation (a) using CAS-LT. (b) using atomic Instructions.**

```
1   unsigned List[N]
2   bool isMax[N] = {true};

3   unsigned Max () {
4       unsigned Max;
5       #pragma omp for collapse(2)
6       for (unsigned i = 0; i < N; i++) {
7           for (unsigned j = 0; j < N; j++) {
8               If (i == j) continue;
9               unsigned index = (list[i] < list[j] || list[i] == list[j] && i < j)? i : j;
10              isMax[index] = false        // common concurrent write
11          }
12      }
13      for (unsigned j =0; j < N; j++)
14          If (isMax[j])     Max = j;
15      return Max;
16  }
```

**Figure 4: The Constant Time Maximum Algorithm.**

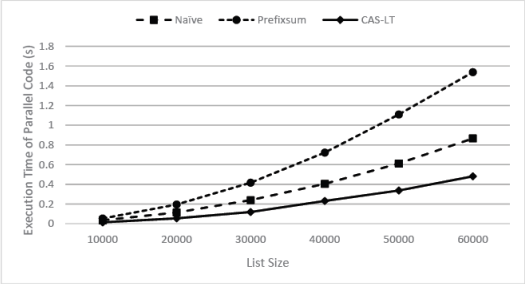


**Figure 5: Effect of list/array size on the execution time of constant-time maximum algorithm, executed by 32 threads.**

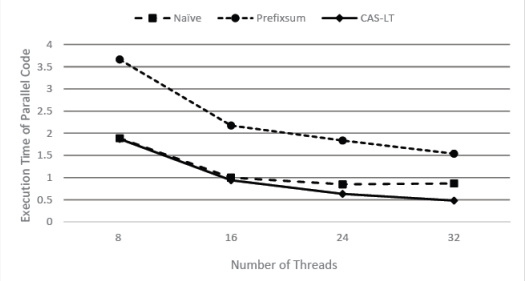


**Figure 6: Effect of the number of threads on execution time of constant-time maximum algorithm. These results are for list size of 60K Elements.**

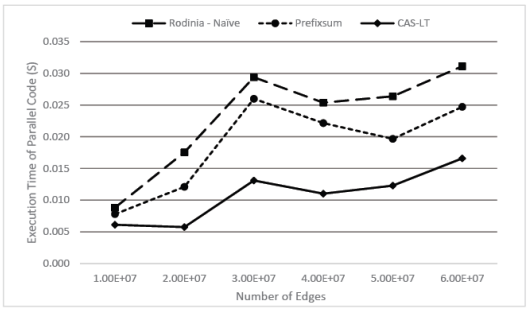


**Figure 7: Effect of number of graph edges on the Execution time of BFS. These results are for randomly-generated undirected graphs with 100K vertices each, executed by 32 threads.**

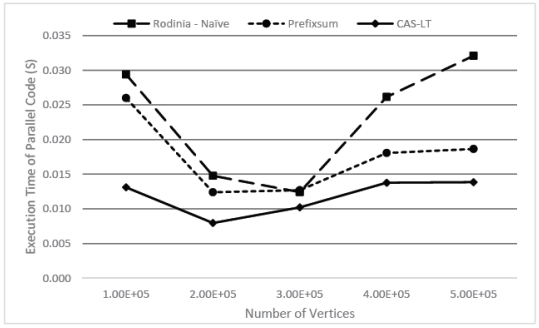


**Figure 8: Effect of the number of graph vertices on the Execution time relationship of BFS. These results are for randomly-generated undirected graphs with total of 30M edges, executed by 32 threads.**

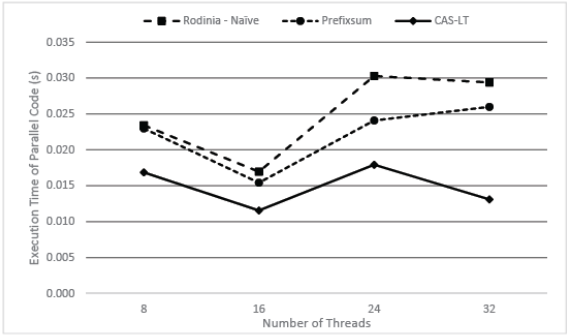


**Figure 9: Effect of number of execution threads on Execution time of BFS. These results are for randomly-generated undirected graphs with 100K vertices, and total of 30M edges.**

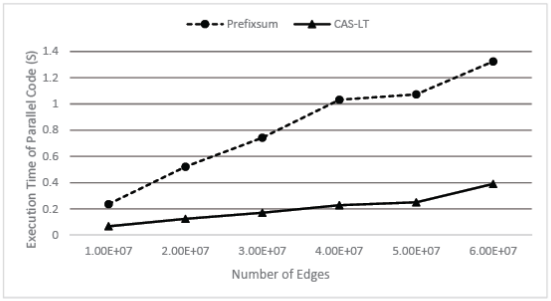


**Figure 10: Effect of number of graph edges on Execution time of CC. These results are for randomly-generated undirected graphs with 100K vertices each, executed by 32 threads.**

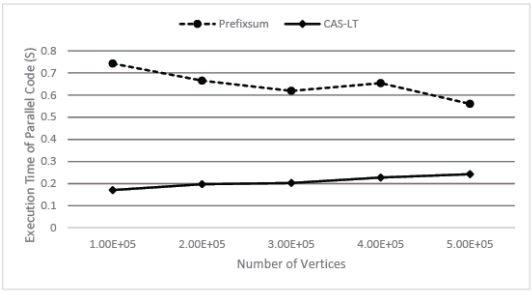


**Figure 11: Effect of number of graph vertices Execution time of CC. These results are for randomly-generated undirected graphs with total of 30 Million edges, executed by 32 threads.**

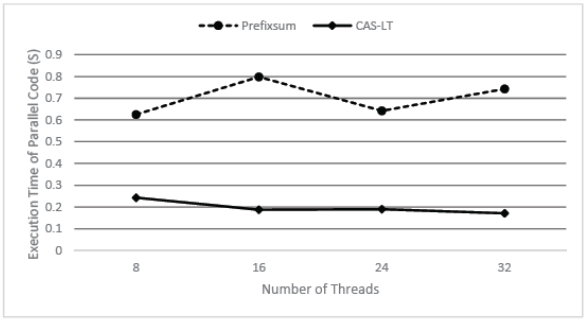


**Figure 12: Effect of the number of execution threads on Execution time of CC. These results are for randomly-generated undirected graphs with 100K vertices, and total of 30M edges.**

**References**


[1] Baruch Awerbuch and Yossi Shiloach. 1987. New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM. IEEE Trans. Comput. C-36, 10 (1987), 1258–1263.

[2] Ravi B. Boppana. 1989. Optimal Separations Between Concurrent- Write Parallel Machines. In Proc. of the 21st Ann. ACM Symp. on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA, David S. Johnson (Ed.). ACM.

[3] Stefan D. Bruda and Yuanqiao Zhang. 2010. Collapsing the Hierarchy of Parallel Computational Models. Int. J. Found. Comput. Sci. 21, 3 (2010).

[4] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. 1988. Efficient simulations between concurrent-read concurrent-write pram models. In Mathematical Foundations of Computer Science 1988, Michal P. Chytil, Václav Koubek, and Ladislav Janiga (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg.

[5] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Eric Schauser, Eunice Santos, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: toward a realistic model of parallel computation. SIGPLAN Not. 28, 7 (1993).

[6] James Edwards and Uzi Vishkin. 2012. Better Speedups Using Simpler Parallel Programming for Graph Connectivity and Biconnectivity. In Proc. of the 2012 Int. Workshop on Programming Models and Applications for Multicores and Manycores.

[7] James Edwards and Uzi Vishkin. 2013. Brief Announcement: Truly Par- allel Burrows-wheeler Compression and Decompression. In Proc. of the 25th Annu. ACM Symp. on Parallelism in Algorithms and Architectures.

[8] Oak Ridge Computing Leadership Facility. 2021. Andes cluster. https://www.olcf.ornl.gov/olcf-resources/compute-systems/andes/

[9] Faith E. Fich, Russell Impagliazzo, Bruce Kapron, Valerie King, and Miroslaw Kutylowski. 1993. Limits on the power of parallel random access machines with weak forms of write conflict resolution. In STACS 93, P. Enjalbert, A. Finkel, and K. W. Wagner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg.

[10] Faith E. Fich, Prabhakar Ragde, and Avi Wigderson. 1988. Simulations among Concurrent-Write PRAMs. Algorithmica 3, 1–4 (nov 1988).

[11] Toshiyuki Fujiwara, Kazuo Iwama, and Chuzo Iwamoto. 2004. Partially effective randomization in simulations between ARBITRARY and COMMON PRAMs. J. Parallel and Distrib. Comput. 64, 3 (2004).

[12] F. Ghanim, U. Vishkin, and R. Barua. 2018. Easy PRAM-Based High- Performance Parallel Programming with ICE. IEEE Transactions on Parallel and Distributed Systems 29, 2 (2018).

[13] J. Gil and Y. Matias. 1994. Fast and Efficient Simulations among CRCW PRAMs. J. Parallel and Distrib. Comput. 23, 2 (1994).

[14] Torben Hagerup. 1992. Fast and optimal simulations between CRCW PRAMs. In STACS 92, Alain Finkel and Matthias Jantzen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg.

[15] Torben Hagerup and T. Radzik. 1990. Every Robust CRCW PRAM Can Efficiently Simulate a PRIORITY PRAM. In Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures (Island of Crete, Greece) (SPAA '90). Association for Computing Machinery, New York, NY, USA.

[16] J. JaJa. 1992. An Introduction to Parallel Algorithms. Addison-Wesley Publishing Company.

[17] Rodinia Project. 2021. Rodinia Benchmark Suite. http://www.cs.virginia.edu/rodinia/doku.php?id=start

[18] Prabhakar Radge. 1992. Processor-Time Tradeoffs in PRAM Simula- tions. J. Comput. Syst. Sci. 44, 1 (Feb. 1992).

[19] Yossi Shiloach and Uzi Vishkin. 1982. An O(logn) parallel connectivity algorithm. Journal of Algorithms 3, 1 (1982). https://www.sciencedirect. com/science/article/pii/0196677482900086

[20] Julian Shun and Guy E. Blelloch. 2014. A Simple Parallel Cartesian Tree Algorithm and Its Application to Parallel Suffix Tree Constr. 1, 1 (Oct. 2014).

[21] U. Vishkin, G. Caragea, and B. Lee. 2008. Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Plat- form. In Handbook on Parallel Computing (Editors: S. Rajasekaran, J. Reif). Chapman and Hall/CRC Press.

[22] X. Wen and U. Vishkin. 2008. FPGA-based prototype of a PRAM-on- chip processor. In Proc. ACM Computing Frontiers.

[23] S. B. Yang, S. K. Dhall, and S. Lakshmivarahan. 1991. Simple randomized parallel algorithms for finding a maximal matching in an ted graph. In IEEE Proceedings of the SOUTHEASTCON '91. 9–581 vol.1.