



Optimized Sound and Complete Data Race Detection in Structured Parallel Programs

Kyle Storey, Jacob Powell, Ben Ogles, Joshua Hooker, Peter Aldous ,
and Eric Mercer  

Brigham Young University, Provo, UT 84601, USA
kyle.r.storey@gmail.com, s.jacob.powell@gmail.com,
benjaminogles@gmail.com, joshua.d.hooker@gmail.com,
{aldous, egm}@cs.byu.edu

Abstract. Task parallel programs that are free of data race are guaranteed to be deterministic, serializable, and free of deadlock. Techniques for verification of data race freedom vary in both accuracy and asymptotic complexity. One work is particularly well suited to task parallel programs with isolation and lightweight threads. It uses the Java Pathfinder model checker to reason about different schedules and proves the presence or absence of data race in a program on a fixed input. However, it uses a direct and inefficient transitive closure on the happens-before relation to reason about data race. This paper presents Zipper, an alternative to this naïve algorithm, which identifies the presence or absence of data race in asymptotically superior time. Zipper is optimized for lightweight threads and, in the presence of many threads, has superior time complexity to leading vector clock algorithms. This paper includes an empirical study of Zipper and a comparison against the naïve computation graph algorithm, demonstrating the superior performance it achieves.

1 Introduction

Correctness in task parallel programs is only guaranteed if the programs are free of data race. A data race is a pair of concurrent uses of a shared memory location when at least one use writes to the location. The order in which these uses occur can change the outcome of the program, creating nondeterminism.

Structured parallelism sometimes takes the form of lightweight tasks. Languages such as Habanero Java, OpenMP, and Erlang encourage the use of new tasks for operations that can be done independently of other tasks. As a result, many programs written in this family of languages use large numbers of threads. In some cases, the number of threads cannot be statically bounded.

Data race is usually undesirable and there is much work to automatically and efficiently detect data race statically. However, static techniques often report too many false positives to be effective tools in practice. Precise data race detection

The research presented here is supported in part by the NSF under grant 1302524.

for a single input can be achieved dynamically. Many dynamic techniques use access histories (shadow memory) to track accesses to shared memory locations.

Vector clocks [12, 22] are an efficient implementation of shadow memory. One analysis based on vector clocks is capable of reasoning about multiple schedules from a single trace [17]. Its complexity is linear if the number of threads and locks used is constant. Vector clocks have been extended to more efficient representations for recursively parallel programs [1, 6] that yield improved empirical results. In all of these cases, the complexity of vector clock algorithms is sensitive to the number of threads used.

When programs are restricted to structured parallelism, shadow memory can reference a computation graph that encodes which events are concurrent. This allows the size of shadow memory to be independent from the number of threads in the program.

The SP-bags algorithm [11], which has been extended to task parallel languages with futures [26], detects data race by executing a program in a depth first fashion and tracking concurrent tasks. Other extensions enable locks [5] and mutual exclusion [25], but can produce false positives.

As an alternative to shadow memory, each task can maintain sets of shared memory locations they have accessed. Lu et al. [20] created TARDIS, a tool that detects data race in a computation graph by intersecting the access sets of concurrent nodes. TARDIS is more efficient for programs with many sequential events or where many accesses are made to the same shared memory location. However, TARDIS does not reason about mutual exclusion.

The computation graph analysis by Nakade et al. [24] for recursively task parallel programs can reason precisely about mutual exclusion [24]. By model checking, it is capable of proving or disproving the presence of data race over all possible schedules. Its algorithm for detecting data races in a single trace is direct, albeit naïve; it computes the transitive closure of the computation graph. This quadratic algorithm admits significant improvement.

This paper presents such an improvement in the form of Zipper, a new algorithm for detecting data races on computation graphs. Zipper maintains precision while utilizing mutual exclusion to improve the efficiency of the computation graph analysis. This algorithm is superior in asymptotic time complexity to that of vector clock implementations when the number of threads is large. It also presents an implementation of the algorithm and a comparison with the naïve computation graph algorithm. The implementation is an addition to the code base published by Nakade et al., which allows for a direct comparison of the two.

In summary, the contributions of this paper are:

- An algorithm for identifying data races in the framework of Nakade et al.,
- A discussion of the relative merits of vector clocks and this algorithm, and
- An empirical study of both the naïve computation graph algorithm and the optimized Zipper algorithm.

The structure of this paper is as follows: Sect. 2 discusses computation graphs and SP-bags. Section 3 presents the Zipper algorithm, and demonstrates the algorithm on a small graph. Section 4 contains an empirical study that compares

Zipper to the original computation graph algorithm. Section 5 details work related to this paper. Section 6 concludes.

2 Background

2.1 Programming Model

The surface syntax for the task parallel language used in this paper is based on the language used by Nakade et al. [24] and is given in Fig. 1. A program \mathbf{P} is a sequence of procedures, each of which takes a single parameter l of type L . The body of a procedure is inductively defined by s . The expression language, e , is elided.

$$\begin{aligned} \mathbf{P} &::= (\mathbf{proc} \ p \ (\mathbf{var} \ l : L) \ s)^* \\ s &::= s; s \mid l := e \mid \mathbf{skip} \mid [\mathbf{if} \ e \ \mathbf{then} \ s \ \mathbf{else} \ s] \\ &\quad \mid [\mathbf{while} \ e \ \mathbf{do} \ s] \mid \mathbf{call} \ l := p \ e \mid \mathbf{return} \ e \\ &\quad \mid \mathbf{async} \ p \ e \mid [\mathbf{finish} \ s] \mid [\mathbf{isolated} \ s] \end{aligned}$$

Fig. 1. The surface syntax for task parallel programs.

The **async**, **finish**, and **isolated** have interprocedural effects that influence the shape of the computation graph. The remaining statements have their usual sequential meaning. The **async**-statement calls a procedure p asynchronously with argument e . The **finish** statement waits until all tasks initiated within its dynamic scope terminate.

This programming model disallows task passing and therefore does not capture some concurrent language constructs like futures. Futures can result in non-strict computation graphs that Zipper is not currently able to analyze. Related work for structured parallelism can reason about programming models that include futures [20, 26] but cannot reason about isolation. ESP-bags can reason about isolated regions, but only when they commute [25], as discussed in Sect. 2.3.

Restrictions on task passing are not unique to this programming model. Task parallel languages usually restrict task passing in some way in order to ensure deadlock freedom. For example, Habanero Java [4] restricts futures to be declared final. Deterministic Parallel Ruby [20] requires futures to be completely independent and to deep copy their arguments. Extending Zipper to treat task passing is the subject of further research. Despite this restriction, the collection of concurrent constructs treated in this paper is sufficient to simulate a wide range of functionality common in modern task parallel languages.

2.2 Computation Graph

A Computation Graph for a task parallel program is a directed acyclic graph representing the concurrent structure of one program execution [7]. The edges in the graph encode the happens before relation [19] over the set of nodes: $\prec \subset N \times N$. There is a data race in the graph if and only if there are two nodes, n_i and n_j , such that the nodes are concurrent, $(n_i \parallel_{\prec} n_j \equiv n_i \not\prec n_j \wedge n_j \not\prec n_i)$, and the two nodes conflict:

$$\begin{aligned} \text{conflict}(n_i, n_j) = & \rho(n_i) \cap \omega(n_j) \neq \emptyset \vee \\ & \rho(n_j) \cap \omega(n_i) \neq \emptyset \vee \\ & \omega(n_i) \cap \omega(n_j) \neq \emptyset, \end{aligned} \quad (1)$$

where $\rho(n)$ and $\omega(n)$ are the sets of read and write accesses recorded in n .

In order to prove or disprove the presence of data race in a program that uses mutual exclusion, a model checker must be used to enumerate all reorderings of critical sections [24]. For each reordering, a different computation graph is generated that must be checked for data race. The main contribution of this paper is an algorithm that can efficiently check a computation graph for data race without reporting false positives even in the presence of mutual exclusion.

2.3 The SP-Bags Algorithm

The SP-bags algorithm can check a computation graph for data race with a single depth first traversal. ESP-bags [25] generalizes SP-bags to task parallel models similar to the one in Fig. 1. However, it can report false positives when the isolated regions in a program do not commute. To demonstrate this limitation, an example is given where it reports a race when in fact there is none.

ESP-bags maintains shadow memory for each shared memory location. The reader and writer shadow spaces record the last relevant task to read or write to the location. Similar shadow spaces exist for isolated regions. To check an access for data race, one must determine if the last task to access a location is executing concurrently with the current task. Tasks that are executing concurrently with the current task are stored in “P-bags”. Serialized tasks are stored in “S-bags”. Therefore, checking an access for data race reduces to checking whether the last task to access the location is in a P-bag.

When a task is created with **async** its S-bag is created containing itself. Its P-bag is empty. When it completes and returns to its parent, its S-bag and P-bag are emptied into its parent’s P-bag. When a **finish** block completes the contents of its S-bag and P-bag are emptied into its parent’s S-bag.

2.4 Example

The program contained in Fig. 2 represents a program with critical sections that do not commute and therefore cause ESP-bags and similar algorithms to report races when there are none. There are two isolated blocks in the program. If the isolated block in procedure *main* executes first then the shared variable x is

```

1      x := false // unwritten
2      a := false
3      proc main () {
4          finish {
5              async p
6                  isolated {
7                      a := true
8                  }
9              y := x
10         }
11     }
12     proc p () {
13         isolated {
14             if (!a)
15                 x := true // written
16         }
17     }

```

Fig. 2. A simple example of a task parallel program.

never written. Otherwise, it is written and the isolated block in p happens before the isolated block in $main$. Because the happens before relation is transitive, the read of x in $main$ becomes ordered with the write in p and there is no race.

Table 1 shows the state of the ESP-bags algorithm as it executes the program. Only rows that contain state changes are listed. The thread that executes procedure $main$ is labeled as T_1 and the thread that executes procedure p is labeled T_2 . The only finish block is labeled F_1 .

Table 1. ESP-bags state through Fig. 2

Line	T_1 S	F_1 P	T_2 S	x I-Writer
0	–	–	–	–
3	T_1	–	–	–
5	T_1	–	T_2	–
15	T_1	–	T_2	T_2
17	T_1	T_2	–	T_2

The first row shows the initial state of the algorithm. The next two rows show the correct initialization of T_1 and T_2 S-bags. On line fifteen, the shared variable x is written to because of the order in which ESP-bags executes the critical sections in the program. When T_2 completes and the algorithm returns to the **finish** block that spawned it, T_2 is placed in the P-bag of F_1 signifying that it will be in parallel with all subsequent statements in the **finish** block.

This is the state that is in play when x is read outside of an isolated block on line nine. Here ESP-bags reports a race because the last isolated writer is in a P-bag. This is a false positive.

ESP-bags is an efficient algorithm, but its imprecision makes it unsuitable when false positives are unacceptable. The goal of the computation graph analysis is to precisely prove or disprove the absence of data race. As such, a comparison of the efficiency of ESP-bags with Zipper is not given in this work.

3 The Zipper Algorithm

The algorithm presented by Nakade et al. [24] checks every node against every other node. While effective, this algorithm is inefficient. This paper presents the Zipper algorithm, which is more efficient but still sound. Zipper performs a depth-first search over non-isolation edges in the computation graph, capturing serialization imposed by isolation. Section 3.1 describes the variables and algorithmic semantics. Section 3.2 presents the algorithm in several procedures. Lastly, Sect. 3.3 shows an example execution of the algorithm.

3.1 Definitions

Integers have lowercase names and collection names (arrays and sets) are capitalized. Their types are as follows:

- $Z\downarrow, Z\uparrow$: Array of sets of node IDs; indices correspond to isolation nodes
- $slider\downarrow, slider\uparrow, next_branch_id, next_bag_id$: Integer
- $Z\downarrow_\lambda, Z\uparrow_\lambda, S, I$: Array of sets of node IDs; indices correspond to branch IDs
- C : Array of sets of pairs of node IDs; indices correspond to branch IDs
- R : Set of node IDs
- B_λ : Array of sets of branch IDs; indices correspond to bag IDs
- B_i : Array of sets of isolation indices; indices correspond to bag IDs

Zippers encode serialization. The isolation zipper Z captures serialization with respect to isolation nodes. The “lambda” zipper Z_λ captures nodes not in the isolation zipper in a particular task.

Traversal begins at the topmost **async** node. Anytime an **async** node is visited, each branch is given a new branch ID (tracked with $next_branch_id$) and are queued for evaluation in arbitrary order. As the algorithm traverses downward, the visited node IDs are added to the set at the index of the branch ID in the S array. This continues until an isolation node is visited that has an outgoing isolation edge. Then, contents of the S array set for the current branch ID are emptied into the set at the index of the isolation node in the down zipper $Z\downarrow$. Once a wait node is visited on the way down, all nodes in the S array set for the current branch ID are emptied into the set at the index of the current branch ID in the down lambda zipper $Z\downarrow_\lambda$. The process is then performed again on the way up, except isolation nodes that have incoming edges are used to trigger a dump from S into $Z\uparrow$. Additionally, when an async node is hit the S set for the current branch is emptied into the up lambda zipper $Z\uparrow_\lambda$.

When an isolated node is visited, its ID is also placed into the set at the index of the current branch ID in I , creating an easily-accessible mapping of branch ID to the isolated nodes on that branch. The ready set R is used to identify which async nodes have had all children traversed; therefore after the last branch of a async node is traversed, the async node ID is placed into the set R to signify that the algorithm can continue with the wait node that corresponds with the async node, since all children have been processed. Any time an isolation node is visited on the way down, $slider\downarrow$ is set to that isolation node's index; similarly, $slider\uparrow$ is set to the index of isolation nodes seen on the way up, restricting data race checks to the fewest possible nodes.

When returning to an async node, the set in I at the current branch ID is emptied into the B_i at the current bag ID. The current branch ID is also placed into the set at the current bag ID index in B_λ . The B_i and B_λ are used to indicate nodes that are concurrent and are not serialized by an isolation edge with the current node.

On the way down a branch, each time a node is visited the p_bag_id is used to index into the B_λ and B_i sets. Each of the indices in the B_λ set at the p_bag_id index is used to index into the $Z\downarrow_\lambda$ to obtain node IDs that are possibly in parallel with the current node. Each pair of possibly parallel nodes is placed into the set located in the C array at the current branch ID. A similar process is used with B_i and $Z\downarrow$; however, only indices larger than $slider\downarrow$ that are in the set in B_i at the p_bag_id index are paired and placed in C .

On the way up the same process is followed, except $Z\uparrow$ and $Z\uparrow_\lambda$ are used, and only indices smaller than $slider\uparrow$ are used when indexing into $Z\uparrow$. Also, node pairs that are possibly in parallel are not placed in C ; instead, node pairs are checked against C . A node pair discovered in both the upwards and downwards traversal is actually in parallel and is checked with *conflict*.

3.2 The Algorithm

The top level of the algorithm is the *recursive_analyze* function. Before it is invoked, several variables are declared and initialized:

- 1: $Z\downarrow = \square$
- 2: $Z\uparrow = \square$
- 3: $Z\downarrow_\lambda = \square$
- 4: $Z\uparrow_\lambda = \square$
- 5: $slider\downarrow = NULL$
- 6: $slider\uparrow = NULL$
- 7: $next_branch_id = 1$
- 8: $next_bag_id = 2$
- 9: $R = \{\}$
- 10: $S = \square$
- 11: $I = \square$
- 12: $C = \square$

```

13:  $B_\lambda = [\emptyset, \emptyset]$ 
14:  $B_i = [\emptyset, \emptyset]$ 
15: recursive_analyze (entry, 0, 0, 1)
16: procedure RECURSIVE_ANALYZE(n, branch_id, s_bag_id, p_bag_id)
17:   if async (n) then
18:     async_node (n, branch_id, s_bag_id, p_bag_id)
19:   end if
20:   if wait (n) then
21:     wait_node (n, branch_id, s_bag_id, p_bag_id)
22:   end if
23:   other_node (n, branch_id, s_bag_id, p_bag_id)
24: end procedure

```

recursive_analyze relies on three helpers. *async_node* analyzes nodes with **async**-statements, *wait_node* analyzes nodes that terminate **finish**-statements, and *other_node* analyzes other nodes:

```

1: procedure ASYNC_NODE(n, branch_id, s_bag_id, p_bag_id)
2:   thread_bag_id = next_bag_id
3:   next_bag_id = next_bag_id + 1
4:    $B_\lambda[\textit{thread\_bag\_id}] = B_\lambda[\textit{p\_bag\_id}]$ 
5:    $B_i[\textit{thread\_bag\_id}] = B_i[\textit{p\_bag\_id}]$ 
6:    $\textit{slider}\uparrow_0 = \textit{slider}\uparrow$ 
7:    $\textit{slider}\downarrow_0 = \textit{slider}\downarrow$ 
8:    $\lceil \textit{slider}\uparrow \rceil = \textit{slider}\uparrow$ 
9:    $\lfloor \textit{slider}\downarrow \rfloor = \textit{slider}\downarrow$ 
10:  for  $n' \in \textit{succs}(n)$  do
11:     $\textit{slider}\uparrow = \textit{slider}\uparrow_0$ 
12:     $\textit{slider}\downarrow = \textit{slider}\downarrow_0$ 
13:    id = next_branch_id
14:    next_branch_id = next_branch_id + 1
15:     $S[\textit{id}] = \emptyset$ 
16:     $C[\textit{id}] = \emptyset$ 
17:    new_bag_id = next_bag_id
18:    next_bag_id = next_bag_id + 1
19:     $B_\lambda[\textit{new\_bag\_id}] = \emptyset$ 
20:     $B_i[\textit{new\_bag\_id}] = \emptyset$ 
21:    recursive_analyze (n, next_branch_id, new_bag_id, thread_bag_id)
22:     $Z\uparrow_\lambda[\textit{id}] = S[\textit{id}]$ 
23:     $\lceil \textit{slider}\uparrow \rceil = \min(\lceil \textit{slider}\uparrow \rceil, \textit{slider}\uparrow)$ 
24:     $\lfloor \textit{slider}\downarrow \rfloor = \max(\lfloor \textit{slider}\downarrow \rfloor, \textit{slider}\downarrow)$ 
25:     $B_\lambda[\textit{s\_bag\_id}] = B_\lambda[\textit{s\_bag\_id}] \cup \{\textit{id}\}$ 
26:     $B_i[\textit{s\_bag\_id}] = B_i[\textit{s\_bag\_id}] \cup I[\textit{id}]$ 
27:     $B_\lambda[\textit{thread\_bag\_id}] = B_\lambda[\textit{thread\_bag\_id}] \cup \{\textit{id}\}$ 
28:     $B_i[\textit{thread\_bag\_id}] = B_i[\textit{thread\_bag\_id}] \cup I[\textit{id}]$ 
29:  end for
30:   $\textit{slider}\downarrow = \lfloor \textit{slider}\downarrow \rfloor$ 

```



```

31:    $R = R \cup \{n\}$ 
32:    $j = \text{get\_join}(n)$ 
33:    $\text{recursive\_analyze}(j, \text{branch\_id}, s\_bag\_id, p\_bag\_id)$ 
34:    $\text{slider}\uparrow = \lceil \text{slider}\uparrow \rceil$ 
35:   return
36: end procedure
1: procedure WAIT_NODE( $n, \text{branch\_id}, s\_bag\_id, p\_bag\_id$ )
2:    $a = \text{get\_async}(n)$ 
3:   if  $a \in R$  then
4:      $c = \text{get\_child}(n)$ 
5:      $\text{recursive\_analyze}(c, \text{branch\_id}, s\_bag\_id, p\_bag\_id)$ 
6:     return
7:   end if
8:    $Z\downarrow_\lambda[\text{branch\_id}] = Z\downarrow_\lambda[\text{branch\_id}] \cup S[\text{branch\_id}]$ 
9:    $S[\text{branch\_id}] = \emptyset$ 
10:  return
11: end procedure
1: procedure OTHER_NODE( $n, \text{branch\_id}, s\_bag\_id, p\_bag\_id$ )
2:    $S[\text{branch\_id}] = S[\text{branch\_id}] \cup \{n\}$ 
3:   if  $\text{isolated}(n)$  then
4:      $i = \text{isoindex}(n)$ 
5:      $\text{slider}\downarrow = i$ 
6:     if  $\text{hasOutgoingEdge}(n)$  then
7:        $Z\downarrow[i] = Z\downarrow[i] \cup S[\text{branch\_id}]$ 
8:        $S[\text{branch\_id}] = \emptyset$ 
9:        $I[\text{branch\_id}] = I[\text{branch\_id}] \cup \{i\}$ 
10:    end if
11:   end if
12:    $\text{checkDown}(n, \text{branch\_id}, p\_bag\_id)$ 
13:    $c = \text{get\_child}(n)$ 
14:    $\text{recursive\_analyze}(c, \text{branch\_id}, s\_bag\_id, p\_bag\_id)$ 
15:    $\text{checkUp}(n, \text{branch\_id}, p\_bag\_id)$ 
16:    $S[\text{branch\_id}] = S[\text{branch\_id}] \cup \{n\}$ 
17:   if  $\text{isolated}(n)$  then
18:      $i = \text{isoindex}(n)$ 
19:      $\text{slider}\uparrow = i$ 
20:     if  $\text{hasIncomingEdge}(n)$  then
21:        $Z\uparrow[i] = Z\uparrow[i] \cup S[\text{branch\_id}]$ 
22:        $S[\text{branch\_id}] = \emptyset$ 
23:        $I[\text{branch\_id}] = I[\text{branch\_id}] \cup \{i\}$ 
24:     end if
25:   end if
26: end procedure

```

Lastly, *checkDown* and *checkUp* are used for identifying data races:

```

1: procedure checkDown(n, branch_id, p_bag_id)
2:   for branch_id'  $\in B_\lambda$  [p_bag_id] do
3:     for n'  $\in Z_{\downarrow\lambda}$  [branch_id'] do
4:        $C[\textit{branch\_id}] = C[\textit{branch\_id}] \cup \{(n, n')\}$ 
5:     end for
6:   end for
7:   for i  $\in \textit{isoindex}(\textit{branch\_id})$  do
8:     if i  $\leq \textit{slider}\downarrow$  then continue
9:   end if
10:  for n'  $\in Z_{\downarrow}$  [i] do
11:     $C[\textit{branch\_id}] = C[\textit{branch\_id}] \cup \{(n, n')\}$ 
12:  end for
13: end for
14: end procedure

1: procedure checkUp(n, branch_id, p_bag_id)
2:   for branch_id'  $\in B_\lambda$  [p_bag_id] do
3:     for n'  $\in Z_{\uparrow\lambda}$  [branch_id'] do
4:       if  $((n, n') \in C[\textit{branch\_id}]) \wedge \textit{conflicts}(n, n')$  then reportRace()
5:     end if
6:   end for
7: end for
8:   for i  $\in \textit{isoindex}(\textit{branch\_id})$  do
9:     if i  $\leq \textit{slider}\uparrow$  then continue
10:  end if
11:  for n'  $\in Z_{\uparrow}$  [i] do
12:    if  $((n, n') \in C[\textit{branch\_id}]) \wedge \textit{conflicts}(n, n')$  then reportRace()
13:  end if
14:  end for
15: end for
16: end procedure

```

3.3 Zipper Example

Figure 3 is a computation graph that serves to illustrate the Zipper algorithm, each step of the algorithm is given in Table 2. The Node column in Table 2 represents the visited nodes, in traversal order. The other columns refer to the global variables and their values at each step. For brevity, empty sets in the Z_{\downarrow} , Z_{\uparrow} , $Z_{\downarrow\lambda}$, and $Z_{\uparrow\lambda}$ arrays are omitted and nonempty sets are preceded by their index. Additionally, the S column shows the set at the current branch ID rather than the entire S array.

At node a in Fig. 3 all variables are initialized to empty; `ASYNC_NODE` is then called, which calls `RECURSIVE_ANALYZE` on line 21. Node b is then visited in `OTHER_NODE`, and added to the S set at the current branch ID. Then, node b is checked for conflicts at line 12 in `OTHER_NODE`, however, the B_i and B_λ

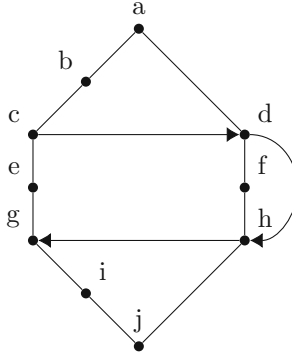


Fig. 3. An example computation graph

at p_bag_id are empty, so no operation takes place. This is true for the entirety of the first branch; data race checks are performed while traversing the second branch. It then recursively visits its child node, c . Node c calls `OTHER_NODE` and is an isolated node, therefore $slider\downarrow$ is set to the index of c (0), on line 5. Node c is also added to S which is emptied into $Z\downarrow$ at index 0 on lines 7 and 8. Recursion continues until node g . The slider is set to the index of g , but S is not emptied because the isolation edge is not outgoing as shown on line 6 in `OTHER_NODE`. Execution continues to node i until the `WAIT_NODE` j is visited and `WAIT_NODE` is called.

Table 2. Step-by-step Zipper algorithm

Node	$Z\downarrow$	$Z\uparrow$	$Z\downarrow_\lambda$	$Z\uparrow_\lambda$	$slider\downarrow$	$slider\uparrow$	S
a	\emptyset	\emptyset	\emptyset	\emptyset	-	-	$\{\}$
b	\emptyset	\emptyset	\emptyset	\emptyset	-	-	$\{b\}$
c	$[0 : \{c, b\}]$	\emptyset	\emptyset	\emptyset	0	-	$\{b, c\}$
e	$[0 : \{c, b\}]$	\emptyset	\emptyset	\emptyset	0	-	$\{e\}$
g	$[0 : \{c, b\}]$	\emptyset	\emptyset	\emptyset	3	-	$\{e, g\}$
i	$[0 : \{c, b\}]$	\emptyset	\emptyset	\emptyset	3	-	$\{e, g, i\}$
j	$[0 : \{c, b\}]$	\emptyset	$[1 : \{e, g, i\}]$	\emptyset	3	-	$\{\}$
i	$[0 : \{c, b\}]$	\emptyset	$[1 : \{e, g, i\}]$	\emptyset	3	-	$\{i\}$
g	$[0 : \{c, b\}]$	$[3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	\emptyset	3	3	$\{i, g\}$
e	$[0 : \{c, b\}]$	$[3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	\emptyset	3	3	$\{e\}$
c	$[0 : \{c, b\}]$	$[3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	\emptyset	3	0	$\{e, c\}$
b	$[0 : \{c, b\}]$	$[3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	\emptyset	3	0	$\{e, c, b\}$
a	$[0 : \{c, b\}]$	$[3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	$[1 : \{e, c, b\}]$	-	-	$\{\}$
d	$[0 : \{c, b\}, 1 : \{d\}]$	$[3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	$[1 : \{e, c, b\}]$	1	-	$\{d\}$
f	$[0 : \{c, b\}, 1 : \{d\}]$	$[3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	$[1 : \{e, c, b\}]$	1	-	$\{f\}$
h	$[0 : \{c, b\}, 1 : \{d\}, 2 : \{f, h\}]$	$[3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	$[1 : \{e, c, b\}]$	2	-	$\{f, h\}$
j	$[0 : \{c, b\}, 1 : \{d\}, 2 : \{f, h\}]$	$[3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	$[1 : \{e, c, b\}]$	2	-	$\{\}$
h	$[0 : \{c, b\}, 1 : \{d\}, 2 : \{f, h\}]$	$[2 : \{h\}, 3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	$[1 : \{e, c, b\}]$	2	2	$\{h\}$
f	$[0 : \{c, b\}, 1 : \{d\}, 2 : \{f, h\}]$	$[2 : \{h\}, 3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	$[1 : \{e, c, b\}]$	2	2	$\{f\}$
d	$[0 : \{c, b\}, 1 : \{d\}, 2 : \{f, h\}]$	$[1 : \{f, d\}, 2 : \{h\}, 3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	$[1 : \{e, c, b\}]$	2	1	$\{f, d\}$
a	$[0 : \{c, b\}, 1 : \{d\}, 2 : \{f, h\}]$	$[1 : \{f, d\}, 2 : \{h\}, 3 : \{i, g\}]$	$[1 : \{e, g, i\}]$	$[1 : \{e, c, b\}]$	-	-	$\{\}$

In `WAIT_NODE` the S set is emptied into $Z \downarrow_\lambda$ at the current branch ID on lines 8 and 9. Node i is placed in S on line 2 in `OTHER_NODE`, then recursion returns to node g . The S set is then emptied into $Z \uparrow$ at index 3 (the index of g) on lines 21 and 22. Execution continues in a similar fashion until it arrives at a . Take note that since the execution is returning up the first branch, the S set is not emptied at node c , since it has an outgoing edge (S empties on the way down on isolation nodes with outgoing edges and on the way up with incoming edges). It is important to note that c and b are in $Z \uparrow_\lambda$ at the branch index. Then a recursive call is made to traverse the second branch in the same manner as the first branch, except data race checks will be performed. The data race checks are not in the table, but are shown in the algorithm in `CHECKDOWN` and `CHECKUP` and described in Sect. 3.

4 Implementation and Results

4.1 Methods

The implementation of Zipper is an addition to the implementation provided by Nakade et al. [24] in their paper. The original implementation is available at <https://jpf.byu.edu/jpf-hj/>. The benchmarks referenced in their paper, which are available as part of the same repository, provide a rich comparison of the two algorithms.

The results of the comparison of the two analyses are included in Table 3. In the table, the *Benchmark* column contains the name of the program used. The *Nodes* column contains the number of nodes in a computation graph. The *Isolation* and *Race* columns indicate, respectively, whether or not isolation and data race are present. The *CG (ms)* and *Zipper (ms)* columns contain the execution time for the respective analyses in milliseconds. Lastly, $\frac{Zipper}{CG}$ is the ratio of the two time measurements.

The experimental results measure the time taken to model check over all possible isolation schedules and reason about each resulting computation graph. All experiments were run on an Intel(R) Xeon(R) Gold 5120 CPU with 8 GB RAM.

4.2 Analysis

Zipper performed better on every benchmark (except for `DoAll2OrigNo`) whose computation graph had more than 37 nodes. For all of the benchmarks with 37 nodes or fewer, the Zipper analysis performed slower except for `IsolatedBlockNo` and `PrimeNumCounter`. In all of these smaller cases, the analyses' execution time was virtually identical. As expected, the degree to which Zipper outperforms the computation graph analysis grows with the number of nodes.

While the size of the computation graph is the strongest predictor of relative runtime performance between the two analyses, other factors contribute to performance. For example, `DoAll1OrigNo` and `DoAll2OrigNo` produce computation graphs with identical structure. However, they differ in both number and

Table 3. Comparison of the computation graph and Zipper analyses

Benchmark	Nodes	Isolation	Race	CG (ms)	Zipper (ms)	$\frac{\text{Zipper}}{\text{CG}}$
ClumpedAccess	11	Y	–	4	14	3.5
PrimitiveArrayNoRace	11	–	–	1	12	12
PrimitiveArrayRace	11	–	Y	1	8	8
SimpleSimpleSimple	11	–	Y	1	14	14
SimpleSimpleSimple2	11	–	Y	1	7	7
VectorAdd	11	–	–	5	18	3.6
DataRaceIsolateSimple	13	Y	Y	10	17	1.7
DoubleBranchExample	13	Y	–	1	7	7
DataRaceIsolateSimple1	15	Y	Y	5	22	4.4
ForallWithIterable	17	–	–	4	11	2.75
IsolatedBlockNo	19	Y	–	9	2	0.2222
Add	23	–	–	18	28	1.5555
PrimeNumCounter	23	Y	–	34	22	0.6470
PrimeNumCounterForAll	27	Y	–	39	44	1.1282
ScalarMultiply	27	–	–	37	41	1.108
TwoDimArrays	27	–	–	22	40	1.8181
ReciprocalArraySumFutures	37	–	–	32	72	2.25
IntegerCounterIsolated	43	Y	–	774	697	0.9005
PrimeNumCounterForAsync	43	Y	–	138	103	0.7463
Antidep1VarYes	45	–	Y	1271	54	0.0424
Antidep2OrigYes	45	–	Y	1234	58	0.0470
ReciprocalArraySum	53	–	–	1801	51	0.0283
PipelineWithFutures	62	–	–	61	53	0.8688
DoAll1OrigNo	207	–	–	650	372	0.5723
DoAll2OrigNo	207	–	–	47287	49534	1.0475
Antidep1OrigYes	2005	–	Y	247736	58215	0.2349
Antidep2VarYes	2005	–	Y	246561	7003	0.0284

placement of shared variable reads and writes in their respective nodes. As a result, the Zipper analysis executes in half the time that the computation graph analysis does on DoAll1OrigNo. On the other hand, the two analyses take about the same amount of time when analyzing DoAll2OrigNo.

The key difference between the Zipper algorithm and the CG algorithm is in the work done to identify the nodes that need to be checked for data race. The Zipper algorithm is able to identify the nodes that execute in parallel much more quickly than the CG algorithm. If there is a large number of reads and writes in proportion to the number of nodes, the Zipper algorithm performs comparably to the CG algorithm, since they both spend a majority of the time checking conflicting nodes for data race in the same way. Conversely, if there are relatively few reads and writes in proportion to the number of nodes, identifying the nodes

that need to be checked becomes much more significant in the analysis time. This makes Zipper more suitable for recursively parallel programs or any task parallel programs that utilize many light weight threads. The Zipper algorithm identifies the nodes that need to be checked much quicker than CG and therefore the overall time is reduced.

4.3 Comparison with TARDIS, SP-Bags and Vector Clocks

Like SP-bags and TARDIS, the Zipper algorithm operates as a depth first traversal of a computation graph that represents a single execution of the program. Zipper tracks reads and writes to shared memory locations in a set for each task and intersects these sets to check for race. However, Zipper does not union access sets and therefore performs more intersect operations than TARDIS. Unlike TARDIS and SP-bags, Zipper can reason about mutual exclusion and includes the scheduled order of isolated regions to reduce the number of intersect operations necessary to check a graph for race.

The vector clock algorithm by Kini et al. [17] checks a program execution for data race by comparing the vector clock of shared memory locations after they are accessed with the current thread's vector clock in order to ensure that the last thread to access the same location is not concurrent with the current thread. The vector clocks are updated after access events and synchronization events.

The vector clock algorithm takes $O(N(L+T^2))$ time to analyze a trace where N is the number of events in the trace, L is the number of locks used and T is the number of threads. In the programming model used in this paper L is always one. It is linear in the length of the trace for programs that use a small, bounded number of locks and threads.

It takes $O(M(T+I))$ time for Zipper to analyze a computation graph and compute the pairs of nodes that are parallel with each other. M is the number of nodes, T is the number of branches in the computation graph and I is the number of isolated regions. Zipper must take the intersection of the access sets for $O(M^2)$ pairs containing K events. This makes the total complexity of Zipper $O(M(T+I) + M^2K)$.

When a program repeats many accesses to the same shared memory locations M and K can be much smaller than N , as TARDIS shows. In this case, Zipper is more efficient than vector clocks and can scale to larger programs. In addition, it may be possible to apply the ideas of TARDIS to Zipper in order to achieve a linear number of intersect and union operations.

5 Related Work

This work is an improvement upon the computation graph analysis by Nakade et al. [24]. Lu et al. [20] implement a similar analysis based on access sets in their tool TARDIS. TARDIS only requires a linear number of intersect and union operations to detect data race in a computation graph but does not support mutual exclusion.

Feng and Leiserson’s SP-bags algorithm [11] is a sound and complete data race detection algorithm for a single program execution but it can only reason about a subset of task-parallel programs that do not use locks. Work has been done to apply SP-bags to other task-parallel models with the use of futures [26], async and finish constructs and isolation [25] with limitations discussed in Sect. 2. Defined in [5] the ALL-SETS and BRELLY algorithms extend SP-bags to handle locks and enforce lock disciplines but can also report false positives when the execution order of critical sections change the control flow of the program being verified. Other SP-bags implementations use parallelization to increase performance [2].

Mellor-Crummey [23] uses thread labels to determine whether two nodes in a graph are concurrent and gives a labeling scheme that bounds the length of labels to be proportional to the nesting level of parallel constructs. This work however, does not treat critical sections at all.

Many algorithms for detecting data race are based on vector clocks that map events to timestamps such that the partial order relation on the events is preserved over the set of timestamps. The complexity of vector clocks algorithm is sensitive to the number of threads used in a program. Fidge [12] modifies vector clocks to support dynamic creation and deletion of threads. Christiaens and Bosschere [6] developed vector clocks that grow and shrink dynamically as threads are created and destroyed. Flanagan et al. [13] replace vector clocks with more lightweight “epoch” structures where possible. Audenaert [1] presents clock trees that are also more suitable for programs with many threads. The time taken in a typical operation on a clock tree is linear with respect to the level of nested parallelism in the program. Kini et al. [17] present a vector clock algorithm that runs in linear time with respect to the number of events in the analyzed execution assuming the number of threads and the number of locks used is constant. This assumption also fails in programs that use large numbers of lightweight threads.

This work relies on structured parallelism to reduce the cost of precise dynamic analysis. Structured parallelism is strict in how threads are created and joined, for example, a locking protocol leads to static, dynamic, or hybrid lock-set analyses for data race detection that are often more efficient than approaches to unstructured parallelism [9, 10, 28]. Unstructured parallelism defines no protocol for when and where threads can be created or join together. Data race detection in unstructured parallelism typically relies on static analysis to approximate parallelism and memory accesses [16, 18, 27] and then improves precision with dynamic analysis [3, 8, 14]. Other approaches reason about threads individually [15, 21]. The work in this paper relies heavily on structured parallelism and it is hard to directly compare to these more general approaches.

6 Conclusion

The computation graph analysis presented by Nakade et al. [24] is well suited to task parallel programs with isolation and lightweight threads. However, its admittedly direct algorithm for identifying data races is inefficient. The Zipper algorithm achieves the same soundness and completeness as does the direct algorithm with significantly improved asymptotic time complexity and empirical performance. In programs with many threads, its time complexity is superior to that of vector clock implementations. This improved algorithm affords improved efficiency to the computation graph analysis, enabling it to prove the presence or absence of data race in larger and more complex task parallel programs.

References

1. Audenaert, K.: Clock trees: Logical clocks for programs with nested parallelism. *IEEE Trans. Softw. Eng.* **23**(10), 646–658 (1997)
2. Bender, M.A., Fineman, J.T., Gilbert, S., Leiserson, C.E.: On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In: *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2004*, pp. 133–144. ACM, New York (2004). <https://doi.org/10.1145/1007912.1007933>
3. Brat, G., Visser, W.: Combining static analysis and model checking for software analysis. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE Computer Society (2001). <https://doi.org/10.1109/ase.2001.989812>
4. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-Java: the new adventures of old X10, August 2011
5. Cheng, G.I., Feng, M., Leiserson, C.E., Randall, K.H., Stark, A.F.: Detecting data races in CILK programs that use locks. In: *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 1998*, pp. 298–309. ACM, New York (1998). <https://doi.org/10.1145/277651.277696>
6. Christiaens, M., De Bosschere, K.: Accordion clocks: logical clocks for data race detection. In: Sakellariou, R., Gurd, J., Freeman, L., Keane, J. (eds.) *Euro-Par 2001*. LNCS, vol. 2150, pp. 494–503. Springer, Heidelberg (2001). <https://doi.org/10.1007/3-540-44681-8-73>
7. Dennis, J.B., Gao, G.R., Sarkar, V.: Determinacy and repeatability of parallel program schemata. In: *Data-Flow Execution Models for Extreme Scale Computing (DFM)*, pp. 1–9. IEEE (2012)
8. Dimitrov, D., Raychev, V., Vechev, M., Koskinen, E.: Commutativity race detection. *SIGPLAN Not.* **49**(6), 305–315 (2014)
9. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: a race and transaction-aware Java runtime. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007*, pp. 245–255. ACM, New York (2007). <https://doi.org/10.1145/1250734.1250762>
10. Engler, D., Ashcraft, K.: RacerX: effective, static detection of race conditions and deadlocks. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP 2003* pp. 237–252. ACM, New York (2003). <https://doi.org/10.1145/945445.945468>

11. Feng, M., Leiserson, C.E.: Efficient detection of determinacy races in CILK programs. *Theory Comput. Syst.* **32**(3), 301–326 (1999). <https://doi.org/10.1007/s002240000120>
12. Fidge, C.J.: Partial orders for parallel debugging. In: *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, PADD 1988*, pp. 183–194. ACM, New York (1988). <https://doi.org/10.1145/68210.69233>
13. Flanagan, C., Freund, S.N.: FastTrack: efficient and precise dynamic race detection. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*, pp. 121–133. ACM, New York (2009). <https://doi.org/10.1145/1542476.1542490>
14. Godefroid, P.: Model checking for programming languages using VeriSoft. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 174–186 (1997)
15. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. *SIGPLAN Not.* **42**(6), 266–277 (2007)
16. Kahlon, V., Sinha, N., Kruus, E., Zhang, Y.: Static data race detection for concurrent programs with asynchronous calls. In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 13–22 (2009)
17. Kini, D., Mathur, U., Viswanathan, M.: Dynamic race prediction in linear time. In: *ACM SIGPLAN Notices*, vol. 52, pp. 157–170. ACM (2017)
18. Kulikov, S., Shafiei, N., Van Breugel, F., Visser, W.: Detecting data races with Java PathFinder (2010). <http://nastaran.ca/files/race.pdf>
19. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978)
20. Lu, L., Ji, W., Scott, M.L.: Dynamic enforcement of determinism in a parallel scripting language. *SIGPLAN Not.* **49**(6), 519–529 (2014). <https://doi.org/10.1145/2666356.2594300>
21. Malkis, A., Podelski, A., Rybalchenko, A.: Precise thread-modular verification. In: Nielson, H.R., Filé, G. (eds.) *SAS 2007*. LNCS, vol. 4634, pp. 218–232. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_14
22. Mattern, F., et al.: Virtual time and global states of distributed systems. *Parallel Distrib. Algorithms* **1**(23), 215–226 (1989)
23. Mellor-Crummey, J.: On-the-fly detection of data races for programs with nested fork-join parallelism. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing 1991*, pp. 24–33. ACM, New York (1991). <https://doi.org/10.1145/125826.125861>
24. Nakade, R., Mercer, E., Aldous, P., McCarthy, J.: Model-checking task parallel programs for data-race. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) *NFM 2018*. LNCS, vol. 10811, pp. 367–382. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_25
25. Raman, R., Zhao, J., Sarkar, V., Vechev, M., Yahav, E.: Efficient data race detection for async-finish parallelism. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) *RV 2010*. LNCS, vol. 6418, pp. 368–383. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_28
26. Surendran, R., Sarkar, V.: Dynamic determinacy race detection for task parallelism with futures. In: Falcone, Y., Sánchez, C. (eds.) *RV 2016*. LNCS, vol. 10012, pp. 368–385. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_23

27. Vechev, M., Yahav, E., Raman, R., Sarkar, V.: Automatic verification of determinism for structured parallel programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 455–471. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_28
28. Voung, J.W., Jhala, R., Lerner, S.: RELAY: static race detection on millions of lines of code. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE 2007, pp. 205–214. ACM, New York (2007). <https://doi.org/10.1145/1287624.1287654>