2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2) | 978-1-6654-1140-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/ESPM254806.2021.00009

Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Graphs

McKay Mower*§, Luke Majors*§, and Tsung-Wei Huang*
* Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT

***Abstract*—Taskflow is a general-purpose parallel and heterogeneous task graph programming system that enables in-graph control flow to express end-to-end parallelism. By integrating control-flow decisions into *condition tasks*, developers can efficiently overlap CPU-GPU dependent tasks both inside and outside control flow, largely enhancing the capability of task graph parallelism. Condition tasks are powerful but also mistake-prone. For large task graphs, users can easily encounter erroneous control-flow tasks that cannot be correctly scheduled by the Taskflow runtime. To overcome this challenge, this paper introduces a new instrumentation module, Taskflow-San, to assist users to detect erroneous control-flow tasks in Taskflow graphs.**

I. Introduction

Recent years have seen a great deal amount of *task-based computing systems* (TCSs), such as oneTBB [1], StarPU [2], TPL [3], Legion [4], Kokkos [5], PaRSEC [6], HPX [7], Fastflow [8], and Taskflow [9] that aim to streamline the building of parallel and heterogeneous applications [10]. Compared to existing TCSs that are limited to directed acyclic graph (DAG) models, *Taskflow* introduces a new *conditional tasking* model to enable end-to-end expression of dependent tasks along with control flow in a general task graph [9]. Programmers benefit from the ability to make *in-graph* control-flow decisions and describe end-to-end parallelism for complex parallel algorithms that frequently call for dynamic control flow, iterations, and non-deterministic blocks.

Conditional tasking is powerful but is prone to mistake. Specifically, given Taskflow's scheduling algorithm [9], [11], users are responsible for writing a correct task graph that does not introduce erroneous control flows, such as deadlock, infinite loop, and unreachable tasks. This manual inspection is not scalable and can become very tedious when the task graph is large. For instance, a large Taskflow-enabled timing analysis workload can spawn 1.7M CPU-GPU dependent tasks and over 2K condition tasks [12], [13], [14], [15], [16]. It is very common for users to mistakenly write a wrong task graph that is not properly conditioned. Therefore, we introduce in this paper *Taskflow-San* as an instrumentation tool to assist users to sanitize erroneous control-flow tasks and guide them toward correct use of condition tasks in Taskflow programs.

II. Taskflow Control-Flow Programming Model

Taskflow is motivated by our DARPA project to reduce the long design times of modern circuits [17]. The main research objective is to advance *computer-aided design* (CAD) tools with heterogeneous parallelism to achieve transformational performance and productivity milestones. Compared with existing TCSs, Taskflow introduces a very simple and expressive programming model to describe task graph parallelism using C++ lambda function objects. Listing 1 demonstrates a simple Taskflow program of four static tasks, where `A` runs before `B` and `C`, and `D` runs after `B` and `C`. The graph is run by an *executor* which schedules dependent tasks across worker threads. Overall, the code explains itself. More details can be found at [9], [18], [19], [20].

```
tf::Taskflow taskflow;
tf::Executor executor;
auto [A, B, C, D] = taskflow.emplace(
  [] () { std::cout << "Task A"; },
  [] () { std::cout << "Task B"; },
  [] () { std::cout << "Task C"; },
  [] () { std::cout << "Task D"; }
);
A.precede(B, C);  // A runs before B and C
D.succeed(B, C);  // D runs after  B and C
executor.run(tf).wait();
```

Listing 1: A task graph of four static tasks.

A. Control-flow Programming

Unlike existing TCSs that are limited to DAG models, Taskflow introduces a new *conditional tasking* model to enable developers to make *in-graph* control-flow that are integrated within task dependencies. Conditional tasks can be used to express control-flow blocks, such as iterations, if-else, and non-deterministic loops. A condition task is a callable that returns an integer index indicating the next successor task to execute. The index is defined with respect to the order of the successors preceded by the condition task. Figure 1 shows an example of if-else control flow, and Listing 2 gives its implementation. The code is self-explanatory. The condition task, `cond`, precedes two tasks, `yes` and `no`. With this order, if `cond` returns `0`, the execution moves on to `yes`, or `no` if `cond` returns `1`.

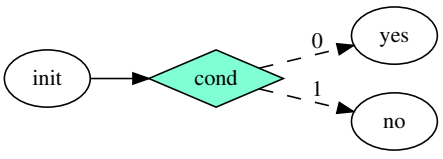

Fig. 1: An example task graph of if-else control flow using one condition task (in diamond).

§Equal contribution

978-1-6654-1140-0/21/$31.00 ©2021 IEEE
DOI 10.1109/ESPM254806.2021.00009

```
auto [init, cond, yes, no] = taskflow.emplace(
  [] () { std::cout << "init"; },
  [] () { std::cout << "cond"; return 0; },
  [] () { std::cout << "cond returns 0"; },
  [] () { std::cout << "cond returns 1"; }
);
cond.succeed(init)
    .precede(yes, no);
```

Listing 2: Taskflow program of Figure 1.

Our condition task supports iterative control flow by introducing a *cycle* in the graph. Figure 2 shows a task graph of *do-while* iterative control flow, implemented in Listing 3. The loop continuation condition is implemented by a single condition task, `cond`, that precedes two tasks, `body` and `done`. When `cond` returns `0`, the execution loops back to `body`. When `cond` returns `1`, the execution moves onto `done` and stops. In this example, we use only four tasks even though the control flow spans 100 iterations. Our model is more efficient and expressive than existing frameworks that count on dynamic tasking or recursive parallelism to execute condition on the fly [4], [6].

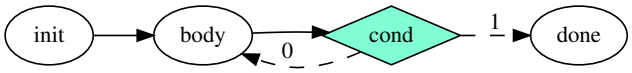

Fig. 2: A Taskflow graph of iterative control flow using one condition task.

```
int i;
auto [init, body, cond, done] = taskflow.emplace(
  [&](){ i=0; },
  [&](){ i++; },
  [&](){ return i<100 ? 0 : 1; },
  [&](){ std::cout << "done"; }
);
init.precede(body);
body.precede(cond);
cond.precede(body, done);
```

Listing 3: Taskflow program of Figure 2.

B. Task Scheduling

Taskflow separates the execution logic between condition tasks and other tasks using two dependency notations, *weak dependency* (out of condition tasks) and *strong dependency* (other else). For example, the two dashed arrows in Figure 1 are weak dependencies and the solid arrow `init`→`F1` is a strong dependency. Based on these notations, we design a simple and efficient algorithm for scheduling tasks, as depicted in Figure 3. When the scheduler receives an HTDG, it (1) starts with tasks of *zero* dependencies (both strong and weak) and continues executing tasks whenever *strong* remaining dependencies are met, or (2) skips this rule for weak dependency and directly jumps to the task indexed by the return of that condition task. By removing the scheduling part of weak dependency, our algorithm falls back to DAG scheduling (marked in gray).

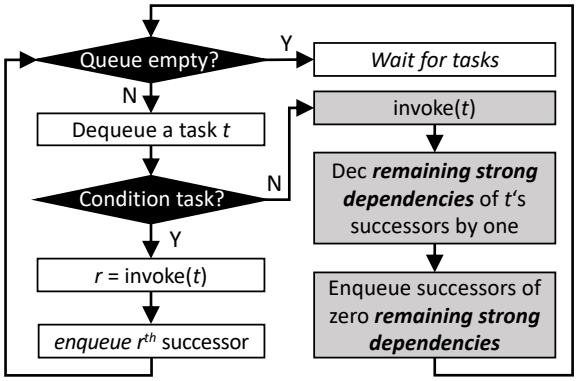

Fig. 3: Flowchart of our task scheduling.

Given this scheduling algorithm, users can infer whether their task graph defines correct control flow. However, this process becomes tedious when a task graph is large. It is desirable to automate this process by calling an instrumentation module to sanitize erroneous control-flow blocks in the given task graph. Based on our user experience, we identify three common control-flow errors, *infinite loop*, *deadlock*, and *unreachable tasks*, which we explain in the next following sections, respectively.

III. Infinite Loop

One problem with cyclic control flow is infinite loop. Infinite loop occurs when non-condition tasks form a cycle of strong dependencies that will execute continuously. Figure 4 shows an example of infinite loop. Execution begins at the condition task, `Start`. Assuming `Start` returns `0`, the scheduler will proceed to task `A`, then task `B`, followed by task `C`. Once task `C` is completed, the scheduler will return to task `A` and the cycle will continue forever. We designed Algorithm 1 to detect if a user-defined task graph will result in an infinite loop.

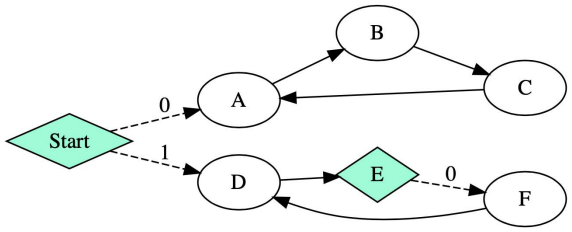

Fig. 4: An example task graph of infinite loop.

The first step of Algorithm 1 copies the original task graph into a custom graph data structure (line 1). During this process, we ignore condition tasks. Condition tasks can break execution if they return a value that does not correspond to another task. For instance, in the task graph shown in Figure 4, if `Start` returns `1`, the loop formed by tasks `D`, `E`, and `F` would begin executing. However, if task `E` returned something other than `0`, task `F` would not be scheduled and the loop would end.

Algorithm 1: Infinite_loop_sanitizer($source$)

Input: a Taskflow graph $source$
Output: a list of infinite loops L

```
1  G ← copy_graph(source);
2  SCCs ← SCC(G);
3  loop ← false;
4  foreach scc ∈ SCCs do
5      foreach t ∈ scc.get_tasks(l) do
6          if t.num_weak_dependencies > 0 then
7              g ← scc.remove_task(t);
8              new_scc_list ← SCC(g);
9              if new_scc_list.empty() then
10                 loop ← true;
11             end
12         end
13     end
14     if loop then
15         L.insert(scc)
16     end
17 end
18 return L;
```

For this reason, cycles formed with weak dependencies are not considered for infinite loops.

After we copy the original graph into the new graph structure, we use Tarjan's algorithm to find all strongly connected components (SCC) in the graph (line 2). Since no condition tasks are included in the graph, each SCC is composed of only strong dependencies making them candidates for infinite loops. We then check each SCC to determine if an infinite loop will occur (lines 4:17). To determine if an SCC forms an infinite loop, we check each task in the SCC for incoming weak dependencies (lines 5:13). Any task with an incoming weak dependency is a possible starting point for an infinite loop. Since an SCC is only formed with strong dependencies, any task in the SCC can only be scheduled if it is reached by a condition task. Otherwise, there will be strong dependencies for each task that will never be met. We identify each task with one or more incoming weak dependencies as a candidate task that could be the start of an infinite loop. Next, we check each candidate to determine if the SCC will result in an infinite loop (lines 6:12). To check a candidate task, we first remove the task from the SCC to form a new graph. Next, we use Tarjan's algorithm again to search for SCCs the new graph. If no additional SCCs are found, then the original SCC will result in an infinite loop (lines 9:11). We then add the original SCC to a list representing all infinite loops in the original task graph and return the list to the user.

A. Example

Figure 5 walks through Algorithm 1 with an example. We first copy the graph into a new graph structure with no condition tasks (Step 1). Next, we contract all SCCs in the graph to obtain the graph shown (Step 2). We then check the SCC for an infinite loop. After checking each task in the SCC, we find that task `A` has an incoming weak dependency so we mark it as a candidate. We remove task `A` from the SCC to obtain the resulting subgraph (Step 3). Finally, we check the subgraph for SCCs and find that no SCCs exist. Since no additional SCCs are found, the graph is determined to result in an infinite loop.

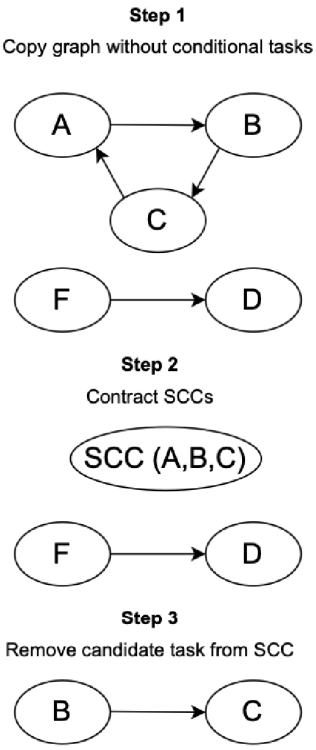

Fig. 5: Illustration of the infinite loop sanitizer algorithm (Algorithm 1).

IV. Deadlock

Another possible problem with cyclic control flow is deadlock. Deadlock occurs when a task in a cycle cannot be scheduled due to unmet strong dependencies. Consider the task graph example in Figure 6, deadlock occurs at task `A` because its strong dependency from task `C` will never be fulfilled. The cyclic dependency between tasks `A`, `B`, and `C` makes it impossible for any of the tasks to be scheduled after an incoming strong dependency from a task outside of the cycle is met. The other cycle shown in Figure 6 does not result in deadlock because task `D` only has a weak dependency from the cycle itself. Therefore, task `D` can be scheduled after `Start` is completed. Another example of deadlock is shown in 7. Deadlock will occur after task `A` executes because task `D` will not be scheduled. Task `D` has a strong dependency from task `E`, but task `E` can only execute after task `D`.

We design Algorithm 2 to detect deadlock in a task graph. Algorithm 2 is similar to Algorithm 1. First, we copy the

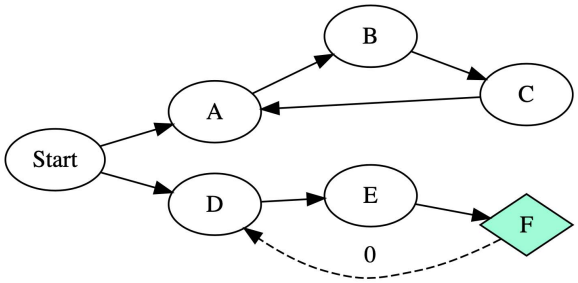

Fig. 6: An example task graph of deadlock.

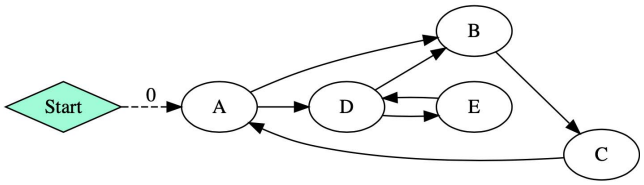

Fig. 7: An example task graph of Deadlock with an incoming weak dependency.

Algorithm 2: deadlock_sanitizer($source$)

Input: a Taskflow graph $source$
Output: a list of SCCs resulting in deadlock L

```
1  G ← copy_graph(source);
2  SCCs ← SCC(G);
3  deadlock ← false;
4  foreach scc ∈ SCCs do
5      foreach t ∈ scc.get_tasks(l) do
6          if t.num_weak_dependencies > 0 then
7              g ← scc.remove_task(t);
8              new_scc_list ← SCC(g);
9              if !new_scc_list.empty() then
10                 deadlock ← true;
11             end
12         end
13     end
14     if deadlock then
15         L.insert(scc);
16     end
17 end
18 return L;
```

original graph into a new graph structure and ignore all condition tasks (line 1). Next, we use Tarjan's algorithm to find all SCCs in the graph (line 2). Since each SCC contains only strong dependencies, it is considered a candidate for deadlock (lines 4:17). An SCC will result in deadlock if it is not an infinite loop. This includes SCCs with any incoming strong dependencies and SCCs with zero incoming strong and weak dependencies. If the SCC does have incoming weak dependencies, we must check each task with a weak dependency in similar way to Algorithm 1 (lines 6:12). We remove the task from the SCC and run the SCC algorithm to search for SCCs in the subgraph formed by removing the candidate nodes from their original SCC. If any additional SCCs are found, then deadlock will occur.

A. Example

Figure 8 illustrates Algorithm 2 based on the example in Figure 7. We first copy the task graph to the new data structure without any condition tasks (Step 1). Next, we contract the SCCs in the graph (Step 2). In this graph, there is only one SCC consisting of tasks `A`, ***B***, `C`, `D`, and `E`. Then, we check this SCC for deadlock. After checking all tasks in the SCC, we identify task `A` as a candidate that needs to be checked since it has an incoming weak dependency. To check the candidate, we remove task `A` from the SCC which results in the subgraph (Step 3). We finally check the subgraph for additional SCCs and find that there is an SCC consisting of tasks `D` and `E` (Step 4). Since there remains an SCC, deadlock is detected.

V. Unreachable Tasks

Another common problem of conditional tasking is unreachable task. Unreachable task is a task that can never be executed. Figure 9 shows an example of unreachable task. If task `B` leads to task `D`, task `E` cannot execute until both tasks `D` and `C` have run. Likewise, if task `B` leads into task `C`, task `E` cannot execute for the same reason. In this case, we call task `E` a *unreachable task* since it can never be executed. We define this type of unreachable task as *merged children* in which the children of a condition task end up merging into one. Figure 10 shows another example of unreachable task. The unreachable task here is `B`, since it has a strong dependency from one of its own children, making it unreachable. For task `B` to execute, tasks `A` and `C` must first run, but task `C` cannot run since task `B` cannot run due to its dependency on task `C`. We define this type of unreachable task as *child to parent condition*, since a non-condition child task has a strong dependency to its parent condition task.

We design Algorithms 3–5 to detect unreachable tasks in a task graph. The algorithm will copy the original task graph into a local graph. The tasks in the local graph have a few important member variables that will be discussed throughout the next couple sections. They are `reachable`, `num_strong_dep`, and `parent`. `reachable` is a boolean that tells whether the task is reachable (defaulted to `false`), `parent` is a variable for this task's parent task (defaulted to `null`), and `num_strong_dep` tells how many strong dependencies this current task has. When the algorithm begins, it places source tasks in a reachable queue. Tasks are put in the reachable queue through an insert wrapper method (Algorithm 3). This method differs from a standard insert by first checking if the task is reachable or not. If the task is reachable, the method returns. If the task is not reachable, the method marks the task as reachable, and the task is put in the reachable queue before the method returns. This guarantees that a task always starts

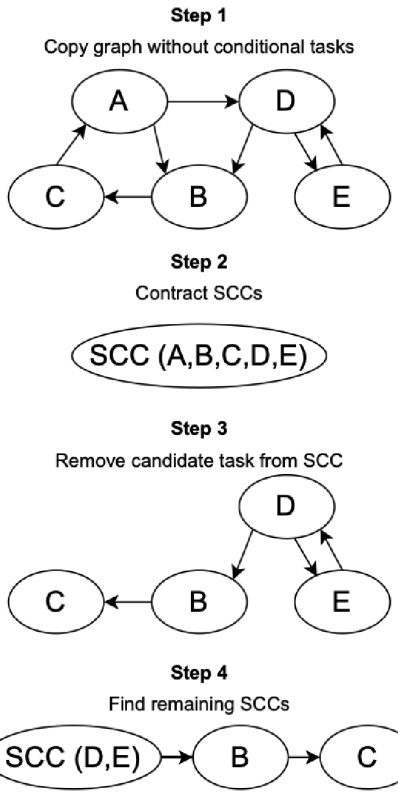

Fig. 8: Illustration of the deadlock sanitizer algorithm (Algorithm 2).

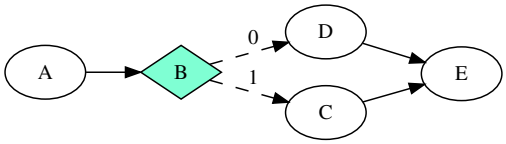

Fig. 9: An exmaple task graph of merged children.

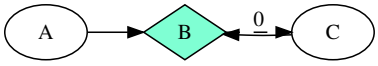

Fig. 10: An example task graph of child-to-parent condition.

in the unreachable state and can only go from an unreachable state to a reachable state. This also guarantees that any task is only pushed into the reachable queue at most one time.

After the source tasks are in the reachable queue, the algorithm enters its main while loop (Algorithm 4). The first couple steps consist of popping the top task of the reachable queue and checking if the task is a condition task. If it is, we explore it through a method called `explore_condition` (Algorithm 5). If the task is not a condition task, its children's strong dependency count is decremented. Any child that has its strong dependency count reach 0 is put in the reachable queue. The most important part of the entire algorithm

Algorithm 3: Insert_wrapper

Input: any Task t, Queue $reachable$

```
1 if !t.reachable() then
2 |   t.reachable ← true;
3 |   reachable.push(t);
4 end
```

is the `explore_condition` method since unreachable tasks would be non-existent without condition tasks. The `explore_condition` method is more complex since we have to track parental tasks for each child task. The goal of tracking parental tasks is to know whether one task has two separate paths from the same parent condition task. An important part of this method is that it will only run on condition tasks that are already reachable. This part of the algorithm, as noted before, has been outlined in Algorithm 5. Note that in `explore_condition` the condition task's child tasks are put into the reachable queue immediately. This is because the condition task can exit on any path at any time, and any child task of a condition task may run even though all of its strong dependencies may not be met (see the scheduling algorithm in Figure 3).

Algorithm 4: Explore_unreachable_tasks_loop

```
1  while !reachable.empty() do
2  |   curr ← reachable.pop();
3  |   if curr.condition = true then
4  |   |   explore_condition(curr);
5  |   end
6  |   else
7  |   |   foreach child c ∈ curr.children do
8  |   |   |   decrement(c.num_strong_dep);
9  |   |   |   if c.num_strong_dep = 0 then
10 |   |   |   |   insert_task(c, reachable);
11 |   |   |   end
12 |   |   end
13 |   end
14 end
```

A. Examples

To better understand how Algorithms 3–5 work, we will go through two examples: one without unreachable tasks (Figure 11) and one with unreachable tasks (Figure 13).

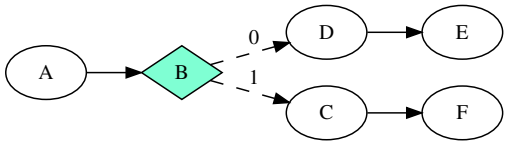

Fig. 11: Example without unreachable tasks

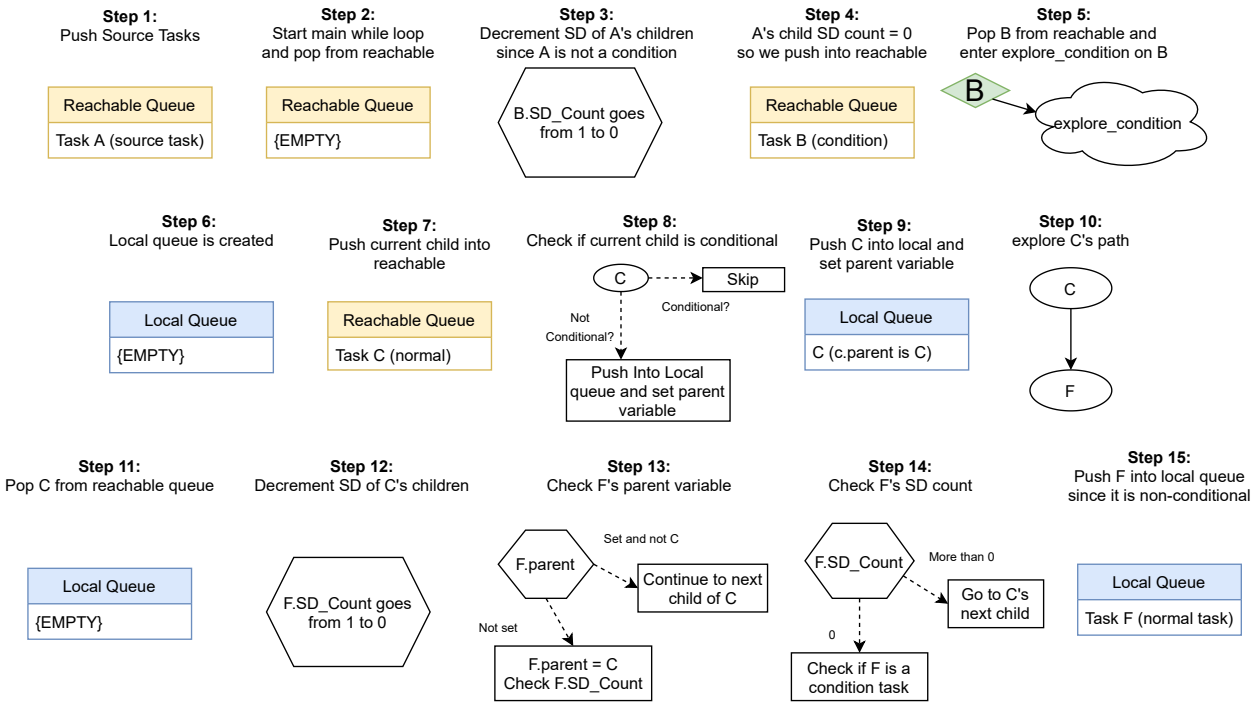

Fig. 12: Illustration of our unreachable task detection algorithm for the task graph in Figure 11.

The first step of the algorithm is to place each source task in the reachable queue. The only source task here is `A`, thus that is the only task in the reachable queue when the algorithm enters the main while loop. The algorithm then pops task `A` and checks if it is a condition task. Task `A` is not a condition task, so we decrement the strong dependency count of each of its children. In this case, task `B` is the only child so we decrement its strong dependency count and it becomes 0. i Since task `B`'s strong dependency count is now 0, the algorithm pushes it into the reachable queue. We end this iteration of the main while loop in Algorithm 4, and continue to the next iteration. Task `B` is the only task in the reachable queue, so we pop it, trigger the if statement on line 3 in Algorithm 4. Since `B` is a condition task, we call `explore_condition` on the current task, `B`. These steps are illustrated in steps 1–5 of Figure 12.

From here, the algorithm creates a local queue as seen in Algorithm 5. This queue is used to explore the different paths of the condition task. The algorithm then starts to iterate through each child of task `B`. Assuming we go in alphabetical order of task names, we start with task `C`. Task `C` is first put in the reachable queue. If task `C` is a condition task, we move on to task `B`'s next child to avoid recursive calls to `explore_condition`. Since task `C` is not a condition task, we push it into the local queue and set the parent of `C` to itself. The local queue now has `C` in it, and we explore `C`'s path by entering the while loop at line 10 of Algorithm 5. This is the starting point of exploring the path of a child that is not a condition. These steps are illustrated in steps 6–10 of Figure 12.

To explore `C`, the algorithm pops it from the local queue and uses it as the current task (local to the explore condition method). The next step is to decrement each of task `C`'s children's strong dependency count. Task `C`'s only child is task `F`, and its strong dependency count will go from 0 to 1. The algorithm will then check if task `F`'s parent has been set or if it is null. Since this is task `F`'s first check, its parent variable is `null`, so the algorithm skips to the else statement at line 17 of Algorithm 5. Since task `F` has not had its parent member variable set (we know this since we skipped the if statement on line 14 of Algorithm 5) we set its parent to the current task, which was task `C` (i.e. `F.parent = C`). From here we check task `F`'s strong dependency count. Since it is 0, the algorithm enters the if statement at line 19 in Algorithm 5. One last check is done on task `F`. Since task `F` is not a condition task, we push it into the local queue to be explored further. These steps are illustrated in steps 11–15 of Figure 12. One thing to note about the if statement at line 14 of Algorithm 5: if it is triggered, we know task `F` would belong to a different path.

From here, the algorithm pops `F` from the local queue, sees that it has no children, and goes to the next iteration of the while loop at line 10 of Algorithm 5. Since the local queue would not have anything left in it, the algorithm would break out of the while loop at line 10 of Algorithm 5. This brings us back up to line 2 of Algorithm 5. We would then go to the next child of task `B`, which is task `D`. Task `D` has the same path as `C`, so the text and diagrams will not be repeated as the outcome would be the same. The end result is that every task in Figure 13 would have been pushed and popped from the reachable

Algorithm 5: Explore_condition

Input: a condition task *cond*

```
1  Queue local;
2  foreach child x ∈ cond.children do
3      insert_task(x, reachable);
4      if x.condition = true then
5          continue;
6      end
7      else
8          x.parent ← x;
9          local.push(x);
10         while !local.empty() do
11             curr ← local.pop();
12             foreach child c ∈ curr.children do
13                 decrement(c.num_strong_dep);
14                 if c.parent and c.parent ≠ curr then
15                     continue;
16                 end
17                 else
18                     c.parent ← curr;
19                     if c.num_strong_dep = 0 then
20                         if c.condition = true then
21                             insert_task(c, reachable);
22                         end
23                         else
24                             local.push(c);
25                         end
26                     end
27                 end
28             end
29         end
30     end
31 end
```

queue at some point, meaning they were all reachable.

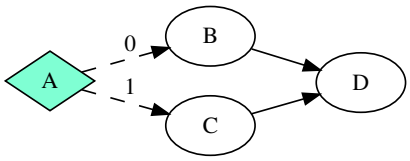

Fig. 13: Example with an unreachable task, `D`.

The second example, illustrated in Figure 13, has an unreachable task `D`. Starting out, the algorithm will push the only source node, task `A` into the reachable queue. We then enter our main while loop in Algorithm 4 and pop task `A` from the reachable queue. Since task `A` is a condition task, we explore it through `explore_condition`. Like Figure 12, a local queue is created and we start to iterate through each child of the condition task. The algorithm will start with task `B` and push it into the reachable queue. The algorithm will then check if task `B` is a condition task. It is not, so its parent member variable gets set to `B` (i.e. `B.parent = B`), and it gets pushed into the local queue. These steps are illustrated in steps 1–5 of Figure 14.

We then enter the while loop at line 10 of Algorithm 5. This while loop will explore the path starting at task `B`. We pop task `B` from the local queue, which will then be our current task local to the `explore_condition` call. The algorithm will then iterate through each of task `B`'s children. `B`'s only child is `D`, so we decrement task `D`'s strong dependency count. In this case, `D`'s strong dependency count would go from 2 to 1. We check task `D`'s parent variable. It has not been set so it is null, so we set it to task `B` (i.e. `D.parent = B`). We now check task `D`'s strong dependency count. It is not 0, so we end this iteration of the while loop at line 10 of Algorithm 5. Since the algorithm did not push anything into the local queue, it exits this while loop and goes back up to the next child of our condition task that we were exploring. In this case, the next child of task `A` is task `C`. We insert that into the reachable queue and check whether it is a condition or not. It is not, so we set its parent variable to `C` (i.e. `C.parent = C`) and push it into the local queue. These steps are illustrated in steps 6–10 of Figure 13.

To explore task `C`'s path, we enter the while loop at line 10 of Algorithm 5. Task `C` is the only thing the local queue, so we pop it and check its children. Task `C`'s only child is task `D`, so we decrement its strong dependency count. Task `D`'s strong dependency count now becomes 0 since we decremented it when we explored the path from task `B`. We then check `D`'s parent, which was set to task `B` in a prior iteration. So, task `D`'s parent variable is not `null`, and it is not task `C`, so we continue to `C`'s next child, essentially skipping the step where we add it to the reachable queue. There is no another child for task `C`, so we break out of the while loop in Algorithm 5. This causes the algorithm to break out of the foreach loop in Algorithm 5 since task `A` has no more children to be checked. These steps are illustrated in steps 11–14 in Figure 14. This concludes the second example. The only task that was never pushed into the reachable queue was task `D`. This would mean that the only thing returned by the algorithm would be task `D`, since the last bit of the algorithm only returns the unreachable tasks.

VI. Acknowledgements

The project is supported by the NSF grant CCF-2126672 and NumFOCUS Small Development Grant. We appreciate all contributors and users of Taskflow for their valuable feedback to improve our system.

VII. Conclusion

In this paper, we have introduced Taskflow-San as an instrumentation tool on top of the popular task graph programming system, Taskflow, to assist programmers with the detection of erroneous control-flow tasks. We have introduced three algorithms to sanitize infinite loop, deadlock, and unreachable tasks, that appear commonly in Taskflow programs. Future

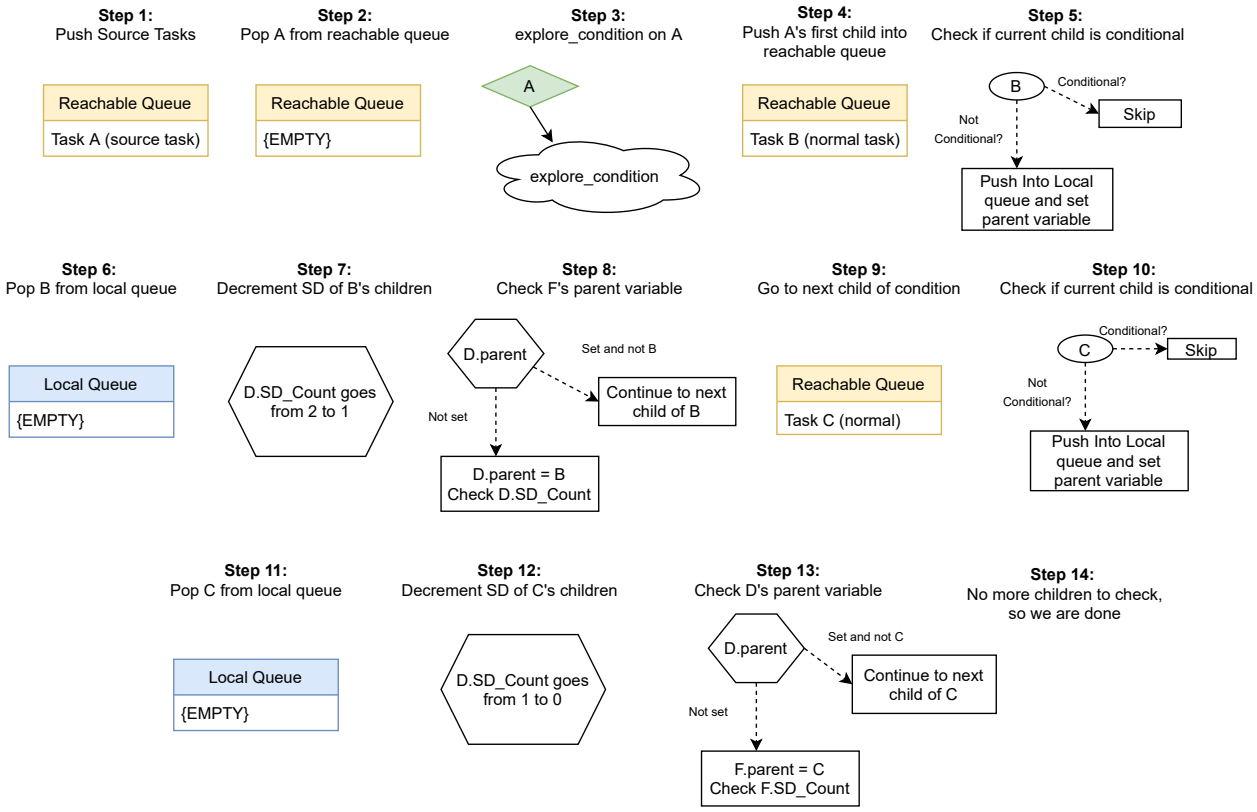

Fig. 14: Illustration of our unreachable task detection algorithm for the task graph in Figure 13.

work will extend Taskflow-San to detect task race caused by condition tasks.

REFERENCES

[1] "Intel oneTBB," https://github.com/oneapi-src/oneTBB.

[2] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, 2011.

[3] D. Leijen, W. Schulte, and S. Burckhardt, "The Design of a Task Parallel Library," in *ACM OOPSLA*, 2009, pp. 227–241.

[4] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *IEEE/ACM SC*, 2012, pp. 1–11.

[5] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014.

[6] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[7] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A Task Based Programming Model in a Global Address Space," in *PGAS*, 2014, pp. 6:1–6:11.

[8] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *Fastflow: High-Level and Efficient Streaming on Multicore*. John Wiley and Sons, Ltd, 2017, ch. 13, pp. 261–280.

[9] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE TPDS*, 2021.

[10] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Peterka, M. Strout, and J. Wilke, "Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity," 2018.

[11] C.-X. Lin, T.-W. Huang, and M. D. F. Wong, "An efficient work-stealing scheduler for task dependency graph," in *IEEE ICPADS*, 2020, pp. 64–71.

[12] T.-W. Huang and M. Wong, "OpenTimer: A high-performance timing analysis tool," in *IEEE/ACM ICCAD*, 2015, pp. 895–902.

[13] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-accelerated Static Timing Analysis," in *IEEE/ACM ICCAD*, 2020, pp. 1–8.

[14] T.-W. Huang, C.-X. Lin, and M. D. F. Wong, "Opentimer v2: A parallel incremental timing analysis engine," *IEEE Design and Test*, vol. 38, no. 2, pp. 62–68, 2021.

[15] T.-W. Huang, G. Guo, C.-X. Lin, and M. Wong, "OpenTimer 2.0: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, vol. 40, no. 4, pp. 776–789, 2021.

[16] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Pash-based Timing Analysis," in *ACM/IEEE DAC*, 2021.

[17] "DARPA Intelligent Design of Electronic Assets (IDEA) Program," https://www.darpa.mil/program/intelligent-design-of-electronic-assets.

[18] T.-W. Huang, C.-X. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *IEEE IPDPS*, 2019, pp. 974–983.

[19] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. F. Wong, "A modern c++ parallel task programming library," in *ACM Multimedia Conference*, 2019, p. 2284–2287.

[20] T.-W. Huang, "A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD," in *IEEE/ACM ICCAD*, 2020.