Final Report - 15618 Spring 2025
Meghna Jain (meghnaj) and Raveena Gupta (raveenag)

TITLE: Parallel Static Time Analysis

SUMMARY:

Our goal for this project was to parallelize a Static TIming Analysis tool which checks if a circuit has any setup or hold time violations. We have parallelized the algorithm on multi-core GHC machines using OpenMP. In this report, we will show multiple algorithms that we implemented from a research paper. The first algorithm we explored is using task loop parallelism by pipelining our algorithm. The next algorithm from the paper explores parallelism by creating a task graph for the circuit. We demonstrate that our results align with those from the paper, in that we see greater speedup from the task graph implementation.

BACKGROUND:

STA is a verification tool that allows designers to verify that their signals propagate through the circuit fast enough to meet setup constraints. It ensures that the signals within the circuit comply with the clock's frequency, and that the clock is able to capture the signals accurately. In our design, we used a “pessimistic approach” where we assumed the worst case to verify correctness of the design. The main goal of a STA is to analyze the propagation delays - arrival time and required time. The forward pass is the **arrival time** which is when the signal reaches a certain point in the circuit. The backward pass is the **required time** that must be adhered to avoid timing failures. The forward pass consists of three stages that must occur sequentially for a component in a circuit: RC, slew, and arrival time calculation. The arrival time depends on the RC and slew delay which are calculated from a component's neighbors. The slew depends on the RC values, thus creating a chain of dependent functions. The backward pass is calculated from the circuit's outputs and goes inwards, keeping into account the time needed for the clock to accurately capture the signal. We use the equation *required time = clock period - setup time* and propagate it backwards to see if the signal reaches before the clock. Since we’re using a pessimistic approach, we take the highest delay for each calculation. We calculate the *slack = required time - arrival time* with a negative slack indicating timing violations. Usually, STA is a very sequential algorithm due to its dependence on sequential functions and paths. However, if the mode of parallelism is captured carefully, we can see significant parallelism.

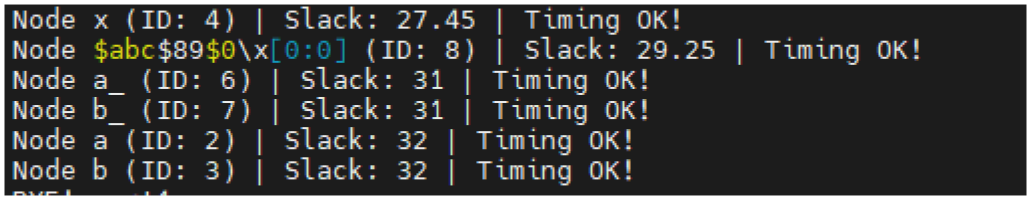
Key data structures:

1. ASIC object
    a. Inputs and outputs of the circuit
    b. Stores a list of all the cells in the circuit
2. Adjacency matrix
    a. Represents the forward connectivity in the graph edges.
    b. Purpose: Forward propagation of arrival time
3. Reverse Adjacency matrix

  a. Represents the backward connectivity in the graph edges.
  b. Purpose: backward propagation of required arrival time
4. Level list
  a. 2-dimensional list of all the nodes in each level
  b. Purpose: Parallelism strategy for task loop parallelism, since all the nodes in a given level are independent of each other and the tasks can be run concurrently.
5. Task Graph
  a. Maps nodes to tasks
  b. In a task graph, each node represents timing propagation tasks and each edge is their dependency. Each node in the graph is either slew, propagation, delay, or an arrival time task while an edge represents a functional dependency (A arrival must happen before B arrival).

I/O:

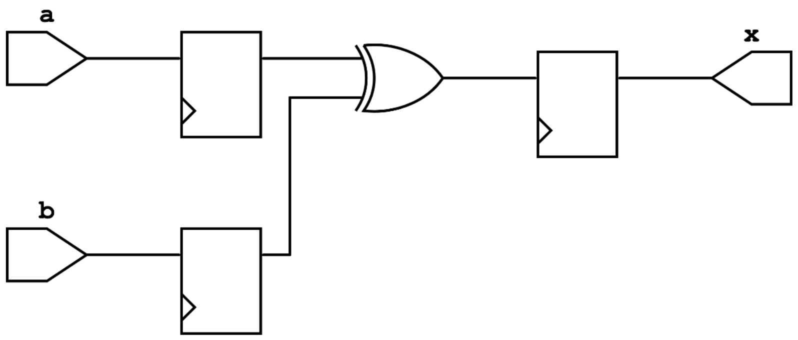
1. Inputs: All the implementations we have explored have the same inputs and outputs, while they may transform the data based on the implementation.
  a. ASIC object which describes the inputs and outputs of the circuit, and all the delays
  b. Adjacency matrix/DAG which represents the connections between the circuit elements.
2. Outputs: The timing for every node. This includes the arrival time, required arrival time, slack, and thus prints if there is a timing violation for each node in the circuit. Below is a sample timing violation for a small circuit.



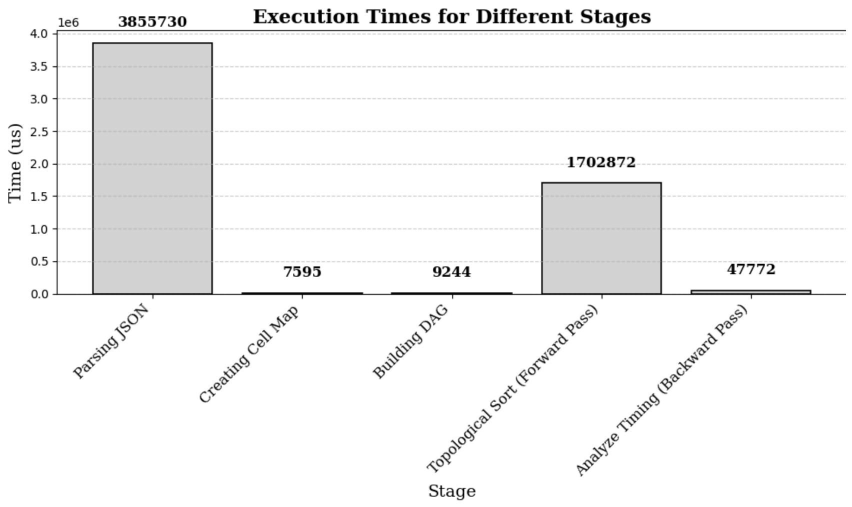
```
Node x (ID: 4) | Slack: 27.45 | Timing OK!
Node $abc$89$0\x[0:0] (ID: 8) | Slack: 29.25 | Timing OK!
Node a_ (ID: 6) | Slack: 31 | Timing OK!
Node b_ (ID: 7) | Slack: 31 | Timing OK!
Node a (ID: 2) | Slack: 32 | Timing OK!
Node b (ID: 3) | Slack: 32 | Timing OK!
```

Our STA Flow:

The flow for our circuit analysis tool starts from verilog design, which describes the hardware behavior. We used synthesis tools like yosys, where we synthesized the circuit into a json netlist where we created a graph data structure of the gates, resistors, capacitors used. We additionally stored the delay values for each component. For instance, an “AND” gate takes approximately 12 nanoseconds to propagate the signal. Below is a diagram of a simple circuit and its gate components.



Additionally, each gate corresponds to a rc value and a slew value. Since wires are not perfect, they have an additional time overhead that our tool tried to encompass. The rc delay is calculated by multiplying the resistance and the neighboring capacitance. The slew is the time it takes for a signal to transition from low to high or high to low. This is dependent on the maximum rc delay value.



Our basic algorithm is divided into different stages. We first parsed the JSON and created a cell cell map that has all the components (AND gate/OR gate) and its inputs and outputs. From this, we created and directed acyclic graph list that has an edge between all the inputs and outputs. We do not consider the time to read the inputs since that is a required cost, and cannot be parallelized. Thus, as seen above, the two main functions that we focused on optimizing are the forward and backward pass.

Dependencies:
There are two key dependencies in the static timing analysis algorithm:

1. Function-level dependencies
   Within the forward propagation for one node, the operations being performed on the node are dependent on previous operations. This pessimistic approach ensures that every possible path into a gate is accounted for accurately.

   The RC value of a node is used to calculate the Slew of a node.
   Both the RC values and the Slew values for a node are used to calculate the arrival time for a node. They must be done in order.

2. Path-level dependencies
   In the forward and backward pass, the nodes in level `l + 1` are dependent on the nodes in level `l`. This dependency is captured in the DAG and adjacency matrix, and means that we need to ensure that the nodes in each level are being executed sequentially.

   Additionally, the backward pass for a node can only begin after the forward pass for that node has completed, since the required arrival time and slack is based on the final arrival time at the outputs.

Thus, there are a lot of dependencies that need to be resolved in STA. However, there is definitely room for parallelism, which we explore by pipelining the forward and backward passes separately (task loop parallelism), and creating a task graph.

APPROACH:

As mentioned earlier, we explored multiple algorithms in our final project. We started off with implementing a sequential version from scratch and then added modifications for the parallel algorithms. We have targeted the GHC multicore machines (8 cores, 1 thread/core).

1. **Parsing Inputs**
   This part is completely sequential. We parse through a JSON file containing details about the circuit, and create an ASIC object that stores these details as described earlier. We also create a cell map which helps us identify and keep track of the nodes.

2. **Sequential Algorithm**
   This algorithm is broken into two distinct parts: forward propagation and backward propagation. In the forward pass we assign “inDegrees” to each of the nodes, which represents how many inputs a node has. We enqueue the nodes that have an “inDegree‘ of 0, and then calculate the RC, Slew and arrival time, by taking the maximum value computed thus far, for the nodes in the queue. Once we complete a pass over a node, we decrement the inDegree of its neighbors, nodes which depend on it. Again, once the

inDegree of a node becomes 0, it is added to the queue and processed. The nodes are added to a “sorted” vector as they are processed and this vector is the final output of the forward pass.
The “sorted” vector is the main input to the backward pass. In this part of, we first reverse the adjacency matrix so that we can move backwards in the circuit. Once that is done, we go backwards from the sorted vector through the circuit and calculate the required arrival time by taking the minimum required arrival time, following the pessimistic approach described earlier.

There are multiple shared data structures here used for bookkeeping such as arrival_time[nodes], required_arrival_time[nodes], slack[nodes].

3. **Node Parallelism**
Before we started implementing the algorithms described in the paper, we tried a naive version of parallelism. This implementation involved using the same sequential implementation without making major modifications to the data structures.

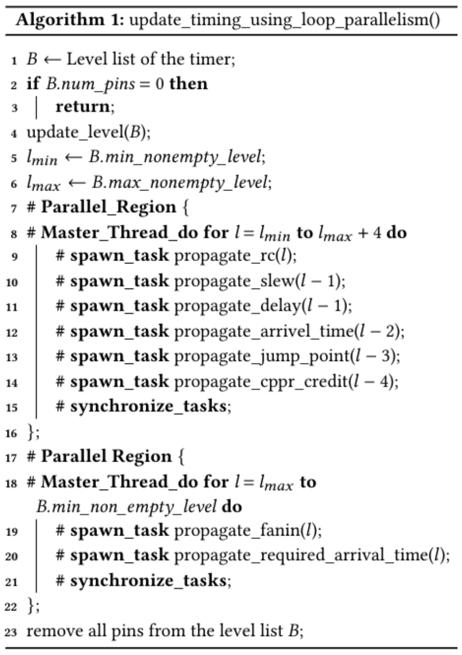
The main modification done for this implementation was to replace a queue with a vector so that threads can be assigned to different elements within the vector.

Forward pass:
Just as earlier, we would add all the nodes with an inDegree of 0 to the “queue” which is actually a vector. We used a “parallel for” pragma to divide up the work among the threads. Since the queue at this point only contains the independent, these tasks can be run concurrently. Here, the function-level dependencies are not violated because the entire iteration is done sequentially. One major consideration was that multiple threads should not be able to update the inDegree of their common neighbor at the same time, thus this update was locked in a critical section. Once a node reached inDegree 0, we would update the global queue before the next iteration to find all the nodes which can be run independently.

Backward pass:
Similar to the forward pass, we would start from the end of the “sorted” vector and parallelize the loop that runs through the independent nodes level by level, and propagate backwards.

In both passes, we had to ensure that the updates to the global bookkeeping data structures like arrival_time, etc. are enclosed in a critical section.

4. **Task Loop Parallelism**
This is the first algorithm we worked on from the paper. In this algorithm, we still keep the forward and backward passes of the algorithm sequential. Each of the passes now has a pipeline of tasks with synchronizations as required.



Algorithm 1: update_timing_using_loop_parallelism()

```
B ← Level list of the timer;
if B.num_pins = 0 then
    return;
update_level(B);
$l_{min}$ ← B.min_nonempty_level;
$l_{max}$ ← B.max_nonempty_level;
# Parallel_Region {
# Master_Thread_do for $l = l_{min}$ to $l_{max} + 4$ do
    # spawn_task propagate_rc($l$);
    # spawn_task propagate_slew($l - 1$);
    # spawn_task propagate_delay($l - 1$);
    # spawn_task propagate_arrivel_time($l - 2$);
    # spawn_task propagate_jump_point($l - 3$);
    # spawn_task propagate_cppr_credit($l - 4$);
    # synchronize_tasks;
};
# Parallel Region {
# Master_Thread_do for $l = l_{max}$ to
  B.min_non_empty_level do
    # spawn_task propagate_fanin($l$);
    # spawn_task propagate_required_arrival_time($l$);
    # synchronize_tasks;
};
remove all pins from the level list B;
```

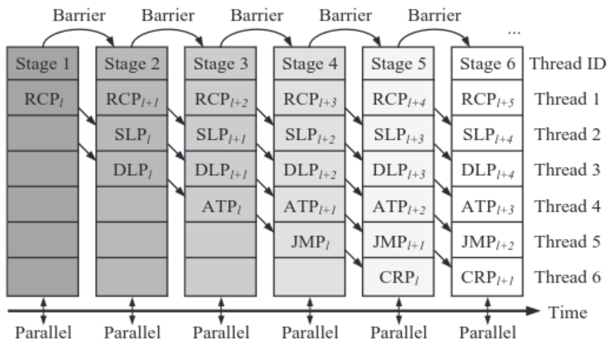


Figure 1: Loop-based parallel timing propagation. Each level applies a `parallel_for` to update timing from the fanin of each node [42].

These images are taken from the paper. They describe the task loop parallelism that we implemented.

This implementation required us to switch from a "sorted" vector to a level list. The creation of the level list does not compute the arrival time for any of the nodes, instead it just ensures that we have a list of all the nodes at each level of the circuit. As we showed earlier, all the nodes at a particular level can be computed concurrently, so the level list helps us with parallelizing the forward and backward passes of STA.

For the circuit shown above in our report, we get the following level list:

```
Level 0: size: 2: 2 3
Level 1: size: 2: 6 7
Level 2: size: 1: 8
Level 3: size: 1: 4
```

where each number in the list represents a circuit element.

In this implementation, we have one master thread that spawns tasks for each level of the circuit. The key thing to note here is the level for which the tasks are being created - the tasks operations with dependencies are launched in later iterations of the loop, thus ensuring that the dependencies are not violated. Also, another thing to note is that for our implementation, we only used the RC, Slew, and arrival time computations, and did not include the jump point, and CPPR functions due to our dataset having limited complexity. We also have barriers implemented using "taskwait" between iterations of the loop. This ensures that all the tasks launched in a certain iteration complete before new tasks are launched. Additionally, the barriers can increase idle time as threads wait on the slowest thread in a level to finish work. Since some nodes can take longer, we don't have good load balancing.

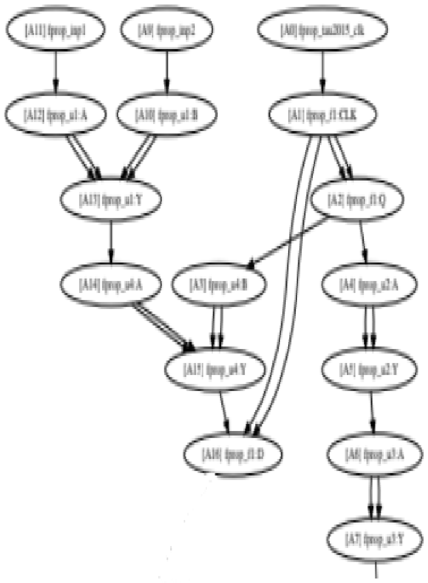
For instance: When we start at level 0, we only create a task for RC for the nodes in level 0. In the next iteration, we create RC tasks for all the nodes in level 1 and also create Slew tasks for level 0. When we are at level 2, we launch RC tasks for level 2 nodes, Slew tasks for level 1 nodes, and ArrivalTime tasks for nodes in level 0, and so on.

5. **Task Graph Parallelism - Forward Propagation**

Introduction

As seen in our previous iterations, we ran into the issue of synchronization barriers. The synchronization barrier on each level caused overhead as all the threads had to wait on each other to finish. This caused significant delays and idle times as workload wasn't balanced and certain nodes took faster. Thus, our next stage was to eliminate barriers by employing task graphs.

```
function topological_TaskGraph(dag, cell_map):

    Initialize inDegree map for each task to 0
    Initialize result list (empty)
    Initialize queue q (empty)

    # Step 1: Calculate in-degrees
    for each task in taskGraph:
        Calculate the inDegree[task]

    # Step 2: Enqueue tasks with in-degree 0
    for each task in inDegree:
        if inDegree[task] == 0:
            add task to queue q

    # Step 3: Process queue in parallel
    while q is not empty
        parallel for each task in q_copy
            dag.processQueue(task, dag, cell_map)
            for each neighbor of task in taskGraph:
                atomically decrement inDegree[neighbor]
                if inDegree[neighbor] == 0:
                    add neighbor to local next_q list

        Merge all local next_q lists into global next_q

        q = next_q

    return result
```



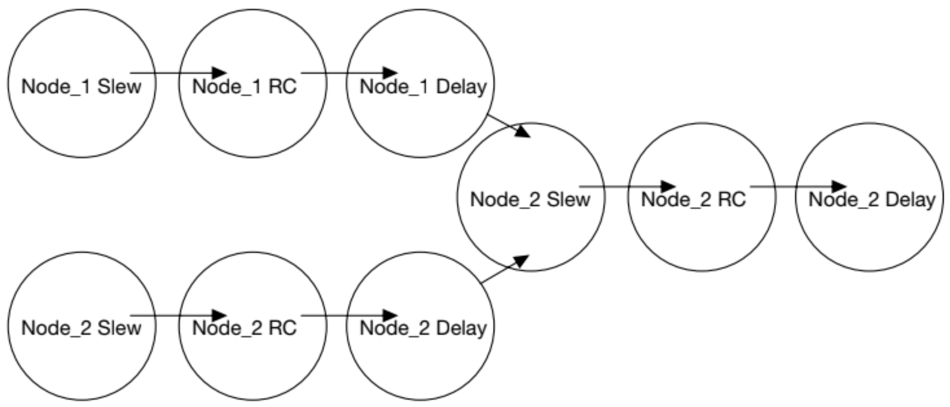
In this algorithm, we employed more parallelism by creating a taskgraph to allow more tasks to run in parallel. The task graph replaces the barriers between levels by using "edges" to represent dependencies.

This mode of parallelism allows for threads to continuously have “work” and creates a larger task pool. In our previous algorithm, we had to finish tasks spawned in level 1 before moving on to level 2. We had a set of barriers between components which caused synchronization stalls for threads. Now, we no longer synchronize by level, but by tasks and their edges. If a component is on a different level, it can still be processed with other components as long as its dependencies are met. Additionally, we saw better performance with a static assignment because our queue had enough tasks to statically assign without incurring the overhead of dynamic scheduling. Since there are more tasks, we have better load balancing because the threads don’t stall and are assignment more concurrent tasks.

In order to support a task graph, we changed the sequential algorithm to support a processing unit that takes a task and processes it accordingly, assuming that its dependencies have been met.

Dependency:

a. Dependency of order: We still have task and function dependencies between Task A and Task B. The figure below shows that the functions and nodes must be processed in a specific order for us to get the accurate values.



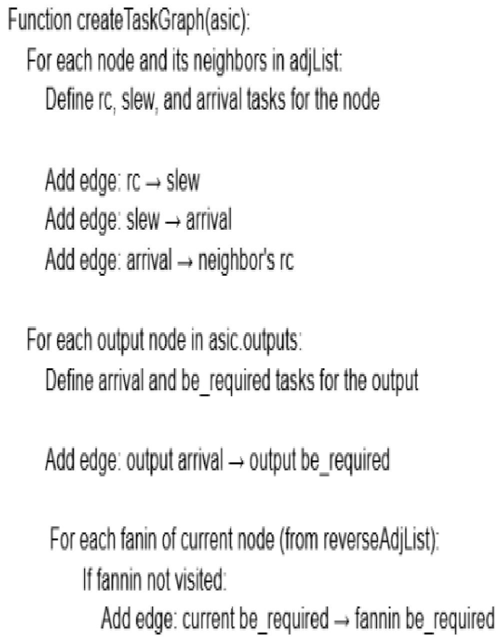
b. Contention and Storage: The algorithm employs a global vector to store the arrival values, causing contention if there are multiple threads processing the same task. Additionally, our task graph is extremely large and can cause cache evictions and false sharing.

6. Task Graph with Backward Propagation

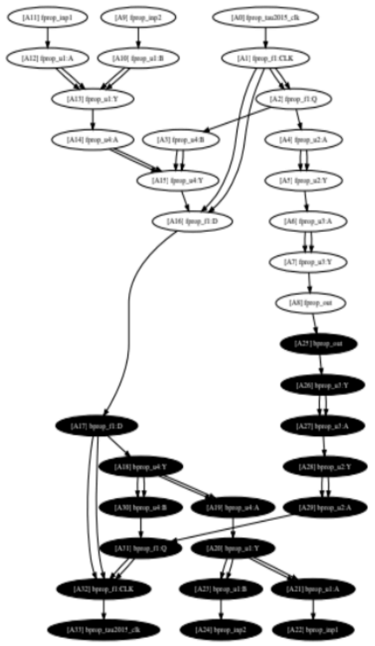
Introduction:

The algorithm before uses task graphs for forward propagation, but still has an “implicit” barrier between forward and backward pass. In the next algorithm, we further parallelize our algorithm by making backward more concurrent. In this algorithm, we run both forward and backward together.

Algorithmic Breakdown:



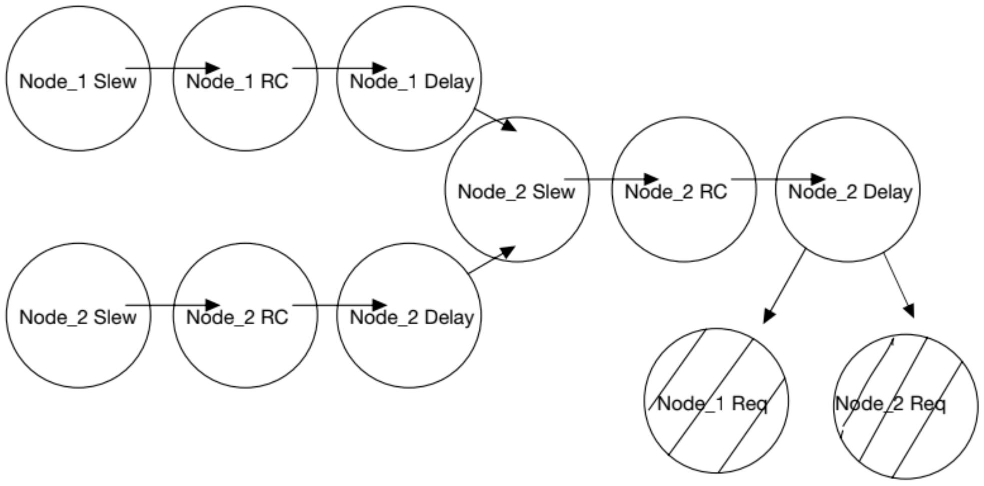
```
Function createTaskGraph(asic):
  For each node and its neighbors in adjList:
    Define rc, slew, and arrival tasks for the node

    Add edge: rc → slew
    Add edge: slew → arrival
    Add edge: arrival → neighbor's rc

  For each output node in asic.outputs:
    Define arrival and be_required tasks for the output

    Add edge: output arrival → output be_required

     For each fanin of current node (from reverseAdjList):
        If fannin not visited:
           Add edge: current be_required → fannin be_required
```



This algorithm focuses on the fact that once a path has been fully traversed, we can start the backward propagation.

In the graph, we created the task “be_required” which signifies that we can calculate the backward pass for the node. In order to make this change, I added a backward processing unit in the sequential code.

The mode of parallelism changed from having a barrier between forward and backward to having edges that represent dependency. When we hit the output of a path, we can go backwards and calculate the required time. This eliminates the need for a barrier between the white (forward) and black (backward) nodes in the figure above, as we can do both computations in parallel. The figure below shows the new added tasks.



Dependency:

Backward Propagation: The dependencies remain the same as forward propagation discussed in the previous algorithm. The backward propagation tasks are still dependent on the forward propagation. This sequential dependency showcases “Amdahl’s law” and brings down our speedup as we still need to wait for the edges to be complete.

DATASET:

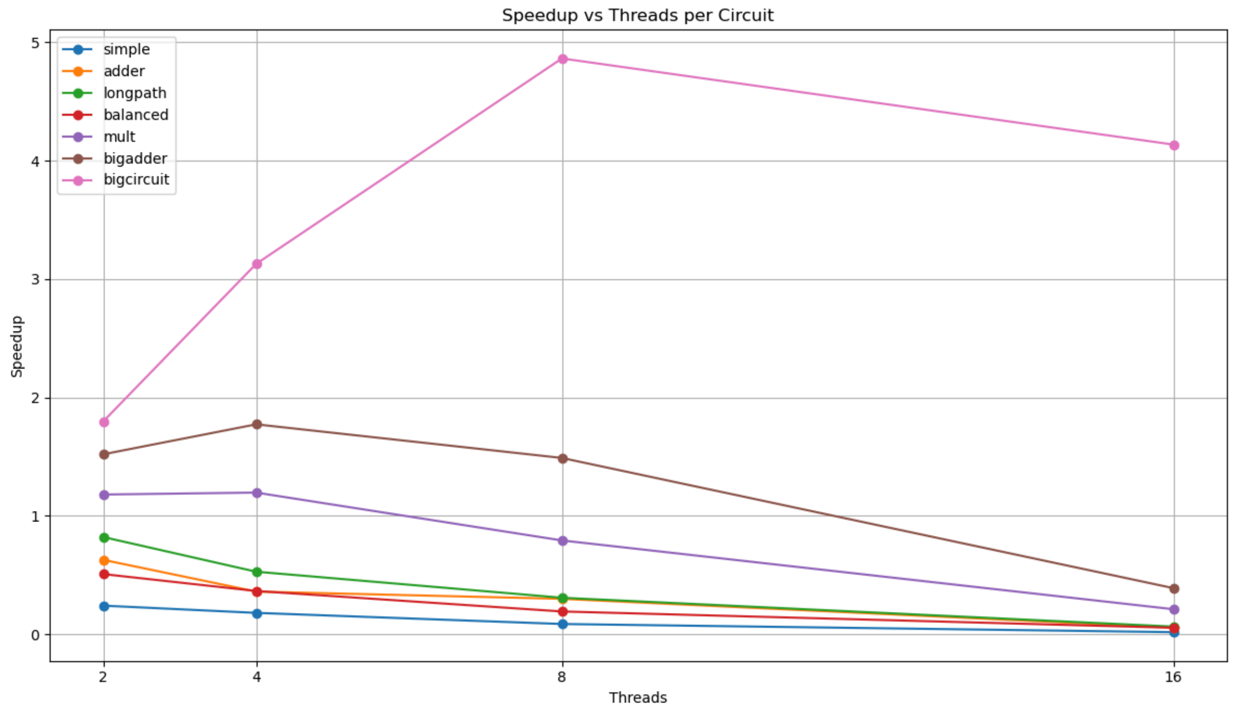
We have a number of circuits which we use to test the performance of our implementations.

Circuit	**Number of nodes**	**Features**
simple	6	Small circuit, mainly for debugging
adder	28	Adder circuit
longpath	24	One long critical path with mainly one node per level
balanced	24	Size of level changes on log scale
mult	94	Relatively larger circuit
bigadder	191	Larger circuit
bigcircuit	11728	Very big circuit

With some preliminary testing we realized that most of our circuits were very small, and the overhead of parallelism outweighed its benefits. Thus we created the “bigcircuit”, which is the largest circuit in our dataset. This is important because not only does it have a lot of levels, it also has a lot of nodes in each of them. Thus, whether we use take loop parallelism where we spawn threads by the level, or task graph parallelism where we create tasks as nodes, we can benefit from having more independent tasks that can run concurrently, thus improving performance.

RESULTS:

1. **Node Parallelism:**
    a. **Speedup:**



The graph above depicts the speedup of the node parallelism algorithm with respect to the number of threads being used across all the circuits. The baseline used here is setting the number of threads to 1. This configuration was chosen because of the modification we had to make from the very first sequential algorithm, as discussed in the previous section.

One of the main takeaways from this implementation, which actually motivated us to create a new circuit, was that the speedup is very low, and even below 1 for really small circuits. This takeaway is quite intuitive because in most of the circuits we looked at, we had less than 100 nodes. Even though we have more threads than nodes, we have a lot of dependencies in the algorithm itself which limit our speedup. This is quite clear from the graph because we can see that as the number of nodes increases, even the “slowdown” of the smaller circuits decreases.

As we add more threads, we can see that the performance is decreasing across all the circuits, except for the “bigcircuit”. This general trend is attributed to the fact that most circuits have almost a source and sink pattern, where the number of nodes falters closer to the inputs and outputs as compared to the middle section of the circuit. This means that in all these “levels” of the circuits, there may be too few elements to be processed in parallel, which then leads to low resource utilization and poor performance. Also, using more threads than available on the machine leads to a lot of context switches and overhead, which is reflected in the graph. In the bigcircuit, however, due to the sheer number of nodes which can be run concurrently, we can see a significant speedup from 2 to 8 threads.

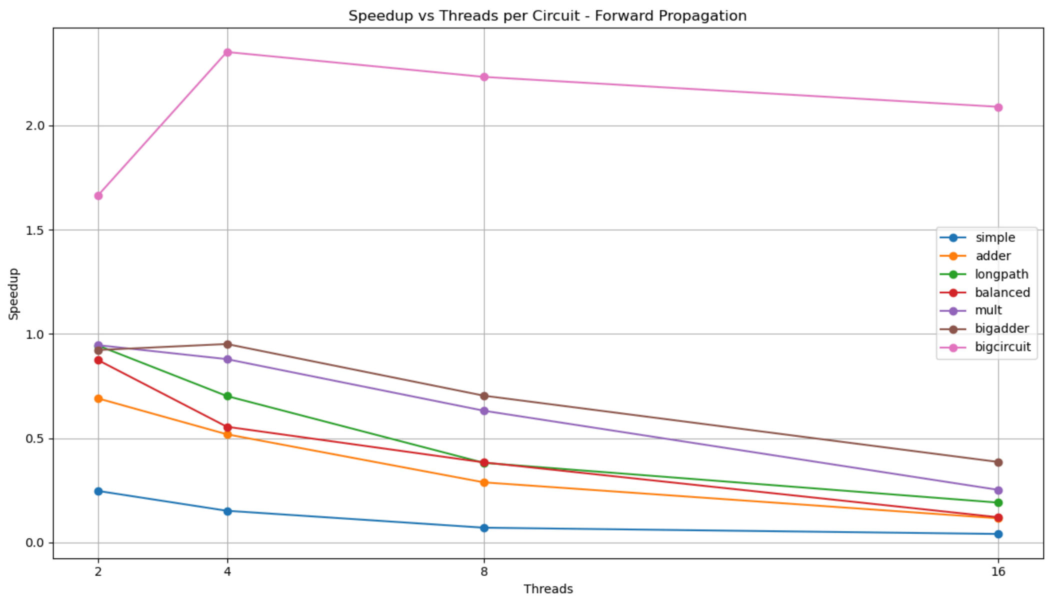
The node parallelism algorithm is only parallelizing one level of the DAG at a time. Thus, if there are fewer than “numThreads” nodes in the level, many of the threads will be idle and not improve our performance. Moreover, after every iteration, we update the neighbors of the nodes, which requires atomic updates. Finally, after all the threads are done with an iteration/node of a given level, we can move onto the next level. Now, we have to accumulate all the other nodes whose inDegree has reached 0. All the threads are blocked at this step because the new level cannot start execution until we have all the nodes and are sure about all the dependencies being resolved.

Despite all these dependencies, we can see that this algorithm can perform well on large circuits when all the cores are being fully utilized, with a speedup of almost 5x on the bigcircuit.

2. **Task Loop Parallelism:**
    a. **Speedup:**

The graphs below show the speedup of the task loop parallel algorithm over the number of threads for different circuits for the forward and backward propagation segments of the algorithm, which is the most intensive segment of the STA algorithm. The baseline used here is setting the number of threads to 1. This configuration was chosen because of the modification we had to make from the node parallelism algorithm.

        i. **Forward Propagation**



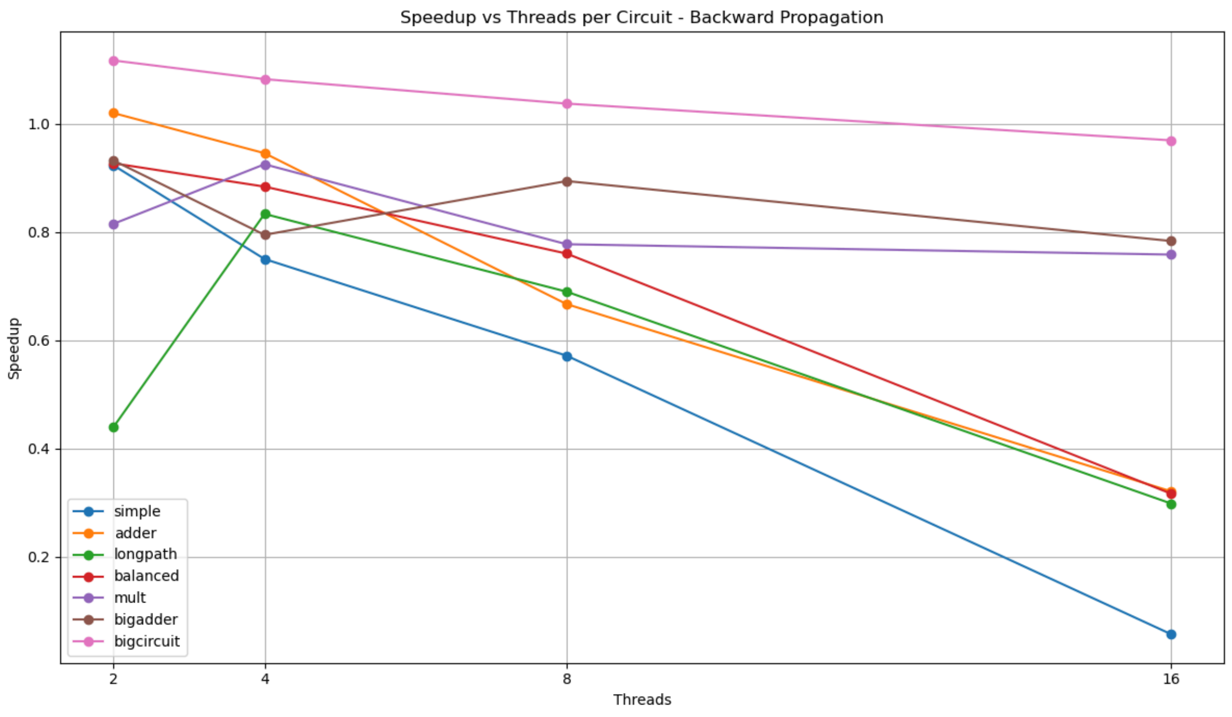
Here, we can see that there is only a speedup for the large circuit with over 10,000 elements. For all the other smaller circuits, there is actually a slowdown. However, we can see here as well that the larger circuits have better performance than the smaller ones, even in terms of slowdown.

In this algorithm, the main limitation to speedup was the heavy use of barriers between tasks of different levels. Moreover, we are incurring the cost of creating so many tasks, since we create 3 tasks (RC, Slew, Arrival time) for each node in the graph, and scheduling them in such a way not to violate the dependencies. We are also updating the globally shared data structures storing the RC, Slew and arrival time values for the nodes in a critical section in each task. All of these factors contribute to the overhead of this algorithm which then results in very poor performance. This result is aligned with the results from the paper, which motivated the implementation of the task graph implementation.

Just as shown in the previous implementation, adding more threads (especially 16 threads) decreases the performance of the algorithm due to the poor load balancing and low resource utilization. Again, the bigcircuit would have more tasks being launched which can be picked up by the threads.

Our maximum speedup for the task loop parallelism is about 3.6x with using 8 threads on the bigcircuit.

ii. Backward Propagation



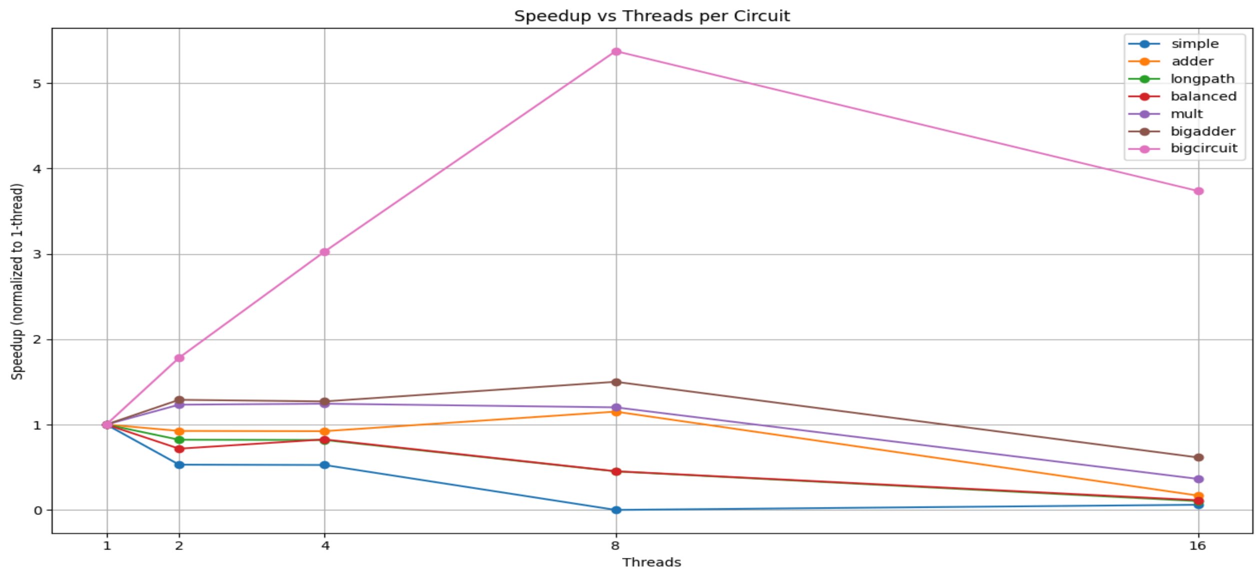
In this segment, the main limitation to speedup was the heavy use of barriers between tasks of different levels. Moreover, we are incurring the cost of creating so many tasks, since we create 2 tasks (Fanin, and required arrival time) for each node in the graph, and scheduling them in such a way not to violate the dependencies.

The backward pass shows a decline in performance as we increase the number of threads for all the circuits. The backward pass is also by nature less parallelizable. This is because this segment of the algorithm has to reverse the adjacency matrix before the propagation can start. This is done sequentially, and is a required cost for the back propagation because this has to be done after the forward propagation for an end-to-end solution. This also follows a similar pipeline pattern as the forward propagation as shown in the pseudocode earlier, which means that there are a lot of dependencies here as well.

Again, adding more threads (until 16) is useful for big circuits because there are more nodes that can be run concurrently. Here, we can see that the mult and bigadder circuits saw a small spike in speedup - which is aligned with the general trend we have seen. The smaller circuits end up with lower resource utilization and poor performance.

3. Task Graph Parallelism:

a. Speedup for forward pass

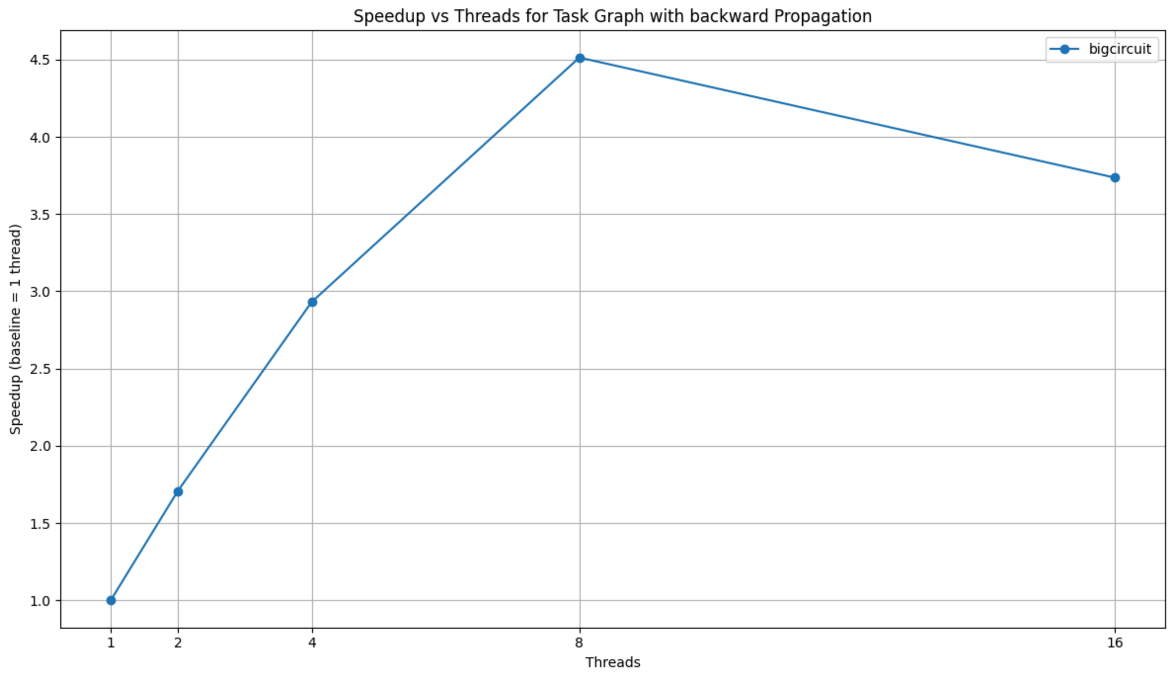


The graph above depicts the speedup of using the taskgraph algorithm for forward pass. As evident, this gave us the most speedup because we no longer have any synchronization stalls or barriers between levels. By making the mode of parallelism “tasks”, we increase the task pool and replace barriers with edges.

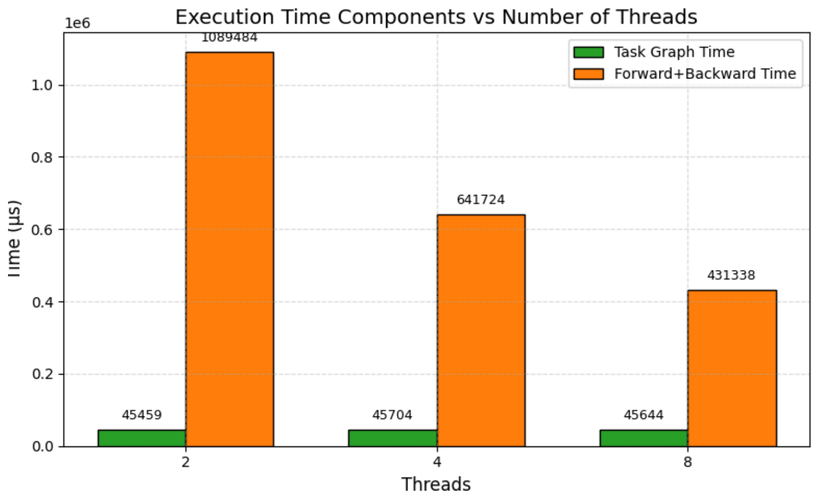
With 8 threads, we reach a speedup of around 5.2. This is expected because we have better parallelization: less barriers, less synchronization stalls, and more tasks working concurrently. This in turn gives us better load balancing as we can always have tasks on the queue. As we add more threads, we are able to execute more tasks in the task queue. The speedup was steepest between 2 and 4 and dwindles between 8 and 16. At 16, due to context switching and our machine only supporting 8 threads, we see less speedup. It’s also possible that the cost of dispatching and synchronizing eats at our speedup. Additionally, we get sublinear performance. This is because of Amdahl’s law. The “serial” work, the edges, cannot be parallelized. The task and node dependency dominate how parallel our algorithm can be. With large circuits especially, there are multiple critical paths. (This is a common design occurrence in industry). These critical paths significantly weigh us down as they need to be run sequentially to get accurate results. Also, we have critical sections between global variables which can cause stalls and cause threads to wait.

The smaller circuits showed less speedup because the overhead of running threads in parallel outweigh the benefits for circuits with fewer nodes. The other netlist that showed speedup is big adder which is the second largest circuit, proving that task graph parallelism is only efficient when our problem size is big.

b. Speedup including backpropagation:



In this algorithm, we created a taskgraph with both forward and backward propagation. This allows for more concurrency and a larger task pool. As evident above, our speedup is slightly slower than the forward task graph algorithm. This makes sense because we're combining back propagation and its dependencies in our taskgraph. Required time can only be calculated once the arrival time is calculated, and thus certain tasks stay in our dependency matrix longer. There are more sequential dependencies, limiting our parallelism. Additionally, our task graph is bigger. If our threads are constantly grabbing values, it can cause cache evictions or false sharing between threads. The contention of appending values to global structures also causes stalls for threads processing the same tasks. These dependencies weigh down our speedup for this algorithm.

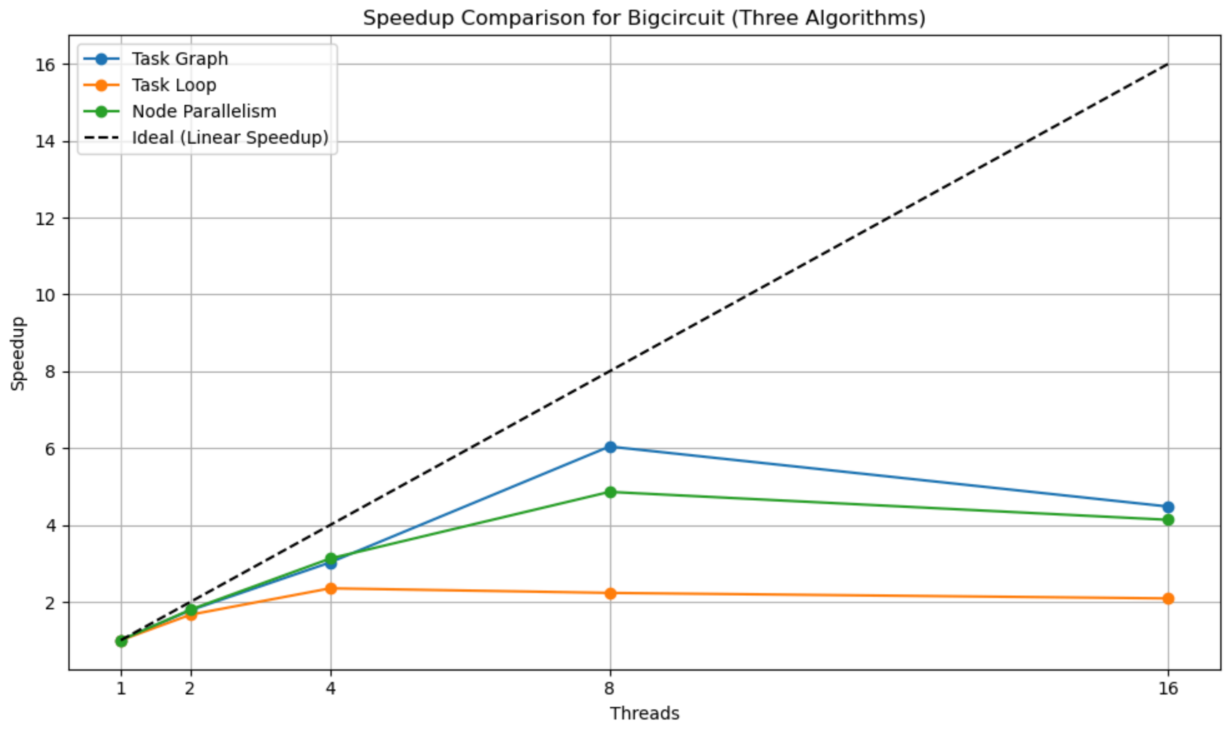


Another thing to consider is the time to build the taskgraph. This is the “extra” overhead of making the process more parallelizable. The task graph construction is a one-time cost and stays consistent with different number of threads. Even with the creation of a task graph, we still see great speedup with 8 threads - showcasing that changing the mode of parallelism can help achieve significant speedup.

OVERALL RESULTS:

In this section we will compare the three major implementations for Parallel Static Timing Analysis.

a. Speedup



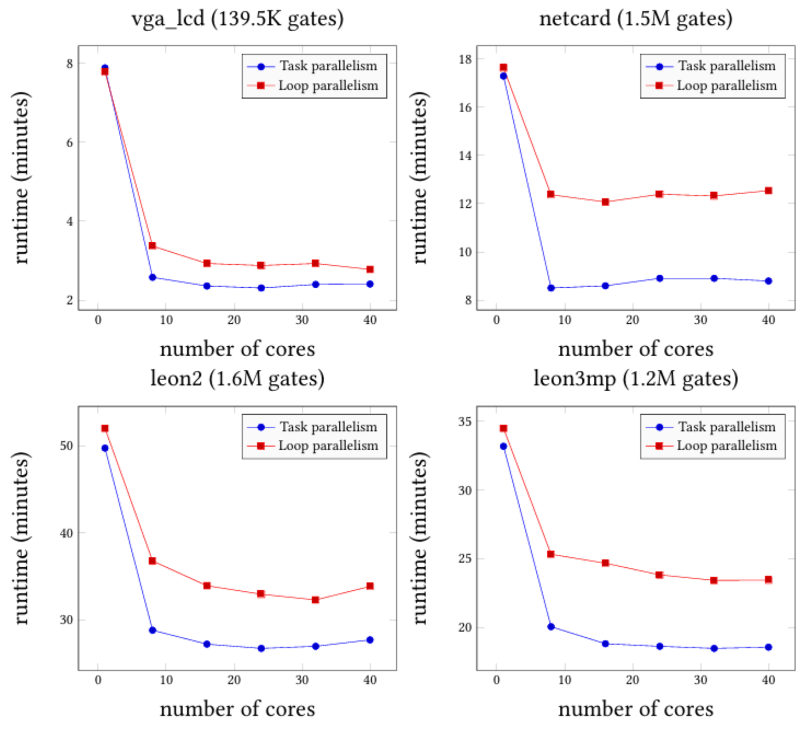
This is a speedup graph which compares the performance of the **forward propagation** of the three major algorithms we have implemented using the bigcircuit. The speedup for each of the implementations was calculated by setting the number of threads to 1, due to the difference in data layout. The dashed line shows the linear speedup, which is ideally what we want.

We can see that out of all the algorithms, the task graph implementation performs the best, while the task loop parallelism has the lowest speedup. The fact that none of the three implementations reach linear speedup shows that there are a lot of dependencies, which make parallelizing this algorithm a difficult task.

The benefit of the task graph is that we are not using a barrier at each level of the circuit. This allows the nodes along a path to continue processing if their inputs have already been processed, thus lowering the synchronization time and idle time for threads. Both other implementations use barriers between two consecutive levels of the circuit.

The node parallelism implementation ends up performing better than the task loop implementation. Even though there is a barrier between each level, the node parallelism uses static scheduling when it assigns threads to nodes within a queue to process. While static scheduling may sometimes result in poor load balancing, here we are assured that each of the nodes are not waiting on any other inputs and can start being processed, which allows us to benefit from the lower overhead of static scheduling.

The task loop implementation incurs a lot of overhead due to creating 3 tasks for each node, dynamically assigning tasks to threads, scheduling decisions, and barriers at every level.

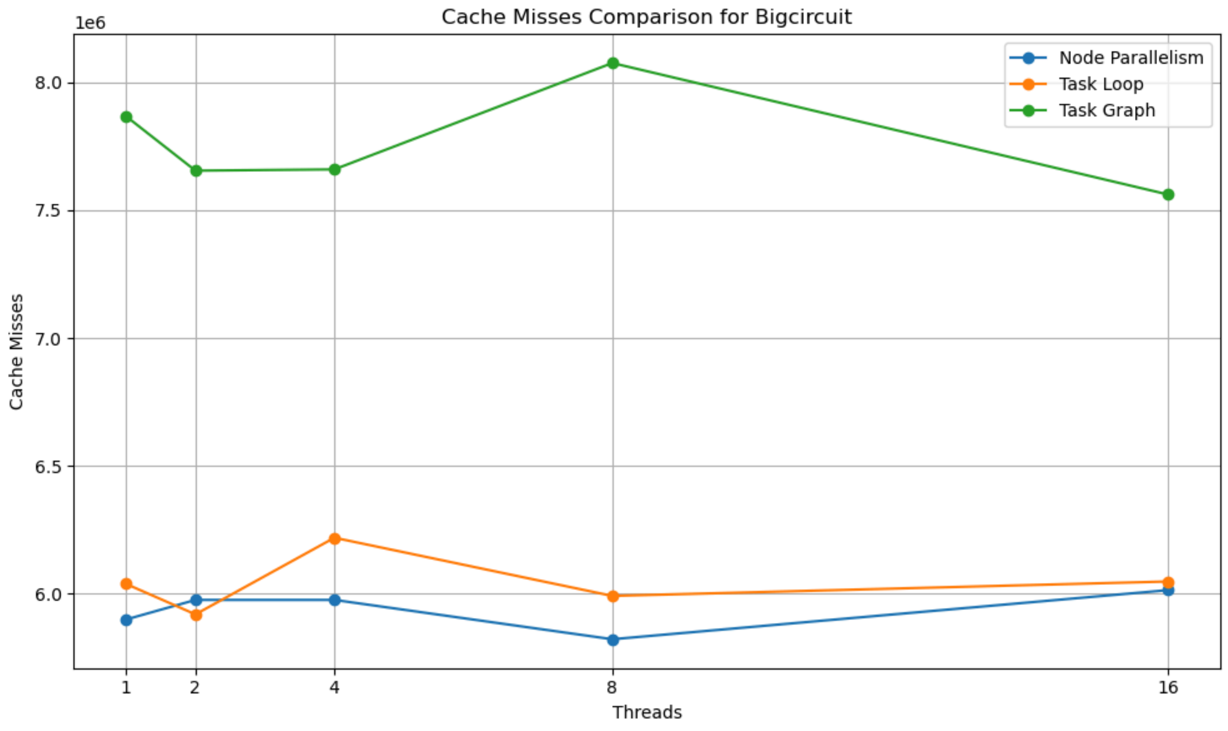


Above, is an image from the paper we are using to implement the algorithms. The graphs depict the runtime of the implementations.

We can see that for loop parallelism, the runtime decreases by less than half. This results in a speedup of about 1.5x to 2.5x, which is consistent with the speedup we saw for the bigcircuit. For the task graph implementation, the runtimes are lowered by about half. This results in a speedup of about 2x.

Thus, we can conclude that the results from our implementation of task graph parallelism and task loop parallelism are consistent with the results from the paper.

b. Cache misses



For our algorithms, we documented the cache misses per thread for our three algorithms. Node parallelism has cache misses that stay fairly stable (around 5.8 M). This suggests that node parallelism has good cache locality. This means that threads work on nearby data without heavily disrupting caches. This makes sense as node parallelism maps a thread to a node, keeping its neighbors and local data in the cache. This is different from task loop parallelism where a thread maps to different nodes within a level, causing cache evictions and managing more data which causes the task loop to have more misses. The task graph has the highest cache misses which makes sense as the task graph is extremely big and cannot fit on the cache, causing frequent evictions and misses. Thus, the general trend is that node parallelism is the best in terms of cache efficiency and task graph is the worst because moving graphs to and from the cache has overhead.

REFERENCES:

1. **Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In Proceedings of the 2024 International Symposium on Physical Design (ISPD '24). Association for Computing Machinery, New York, NY, USA, 51–59. https://doi.org/10.1145/3626184.3635278**

CONTRIBUTIONS:

Distribution of work:

- Creating circuits and parsing inputs
  - Raveena
- Sequential algorithm:
  - Forward pass: Raveena
  - Backward pass: Meghna
- Node parallelism:
  - Naive implementation: Meghna
- Task Loop parallelism
  - Forward propagation: Meghna
  - Backward propagation: Raveena
- Task Graph parallelism
  - Forward propagation: Meghna
  - Backward propagation: Raveena

Distribution of credit: 50-50