2023

Pipeflow

Task-Parallel Pipeline Scheduling Framework

Cheng-Hsiang Chiu

C++ now

Outlines

- Know what pipeline parallelism is and why we need it
- Understand the challenges of existing libraries
- Introduce our solution Pipeflow
- Demonstrate how to use Pipeflow in an application
- Dive into the scheduling algorithm
- Conclude the talk

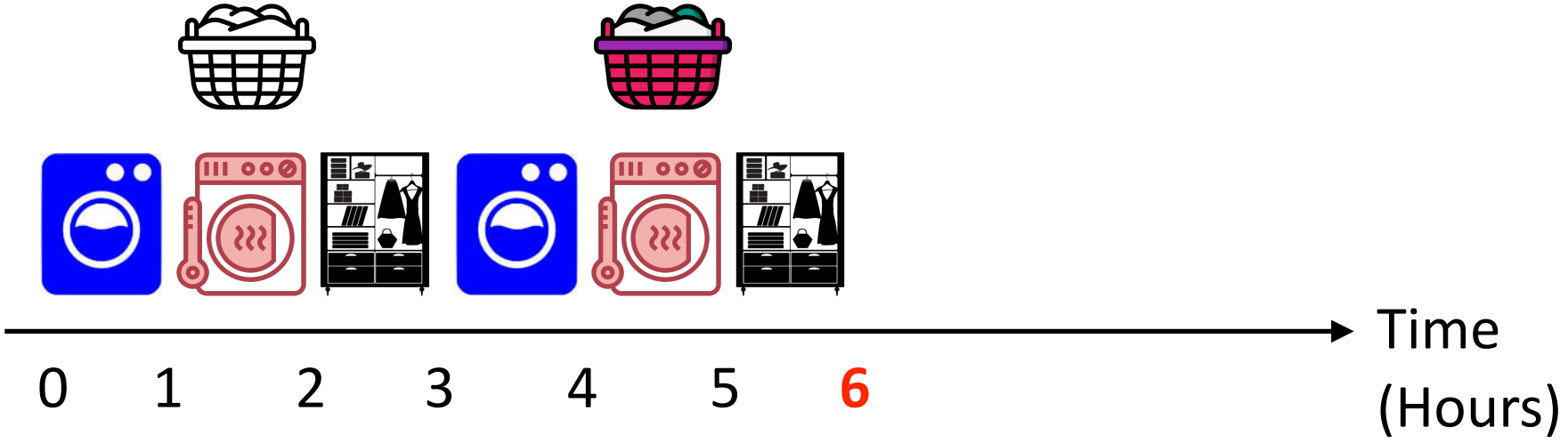
What Is Pipeline Parallelism?

- Let's start with a laundry analogy
  - The laundry task consists of 3 stages, washer, dryer, closet
  - Each stage costs 1 hour
  - One task costs **3** hours

Source : https://www.flaticon.com/free-icon/

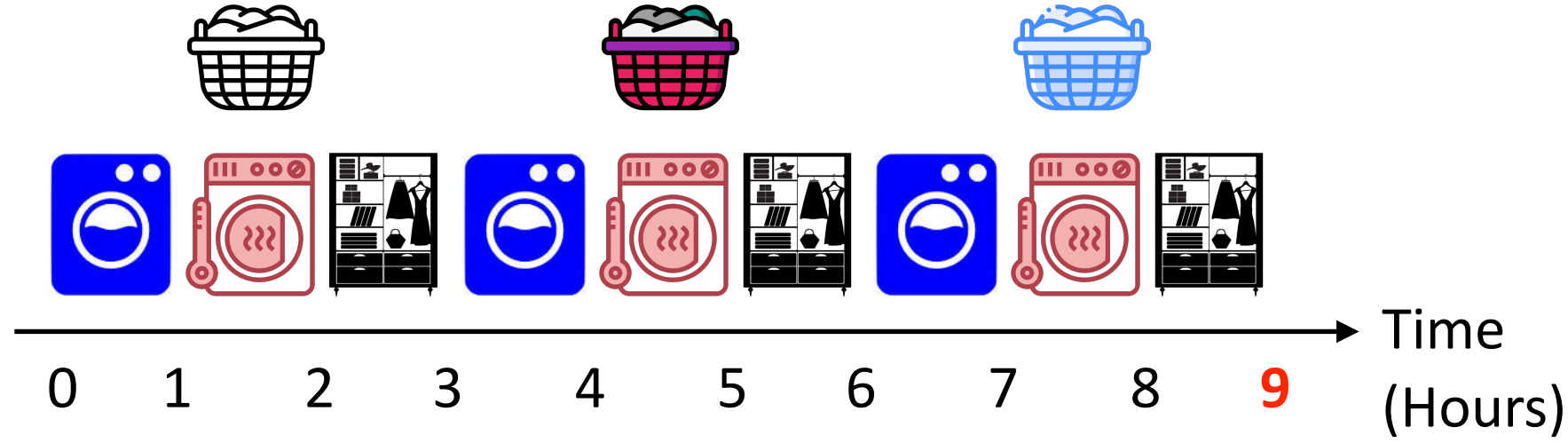
What Is Pipeline Parallelism?

- Let's start with a laundry analogy
  - The laundry task consists of 3 stages, washer, dryer, closet
  - Each stage costs 1 hour
  - 2 tasks cost **6** hours



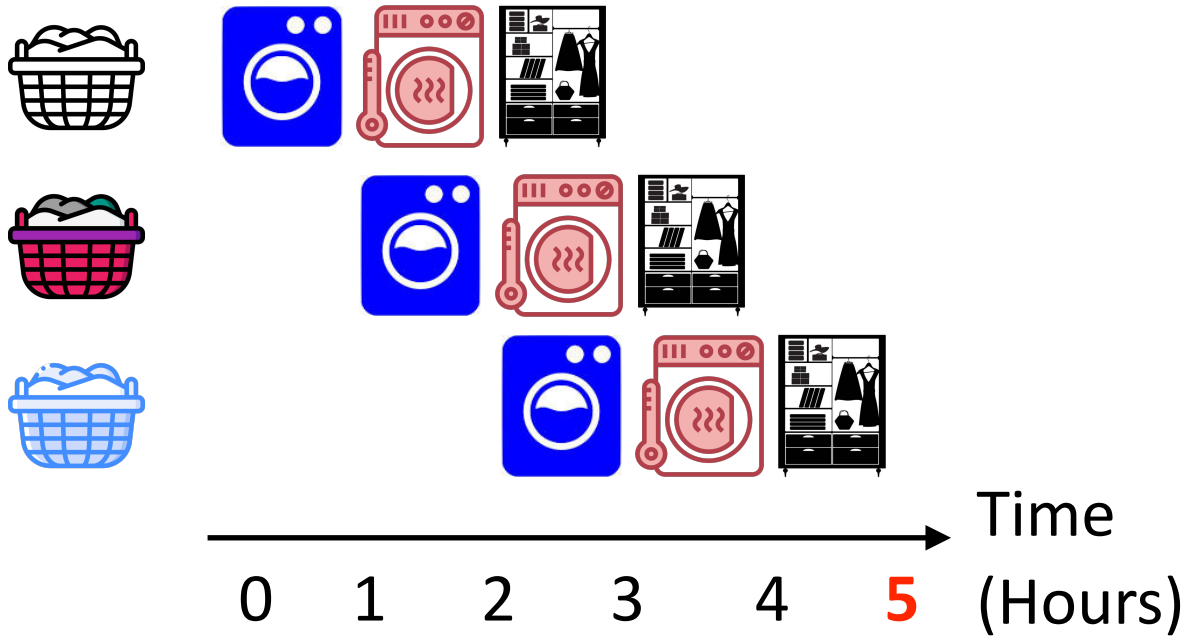
What Is Pipeline Parallelism?

- Let's start with a laundry analogy
  - The laundry task consists of 3 stages, washer, dryer, closet
  - Each stage costs 1 hour
  - 3 tasks cost **9** hours



What Is Pipeline Parallelism?

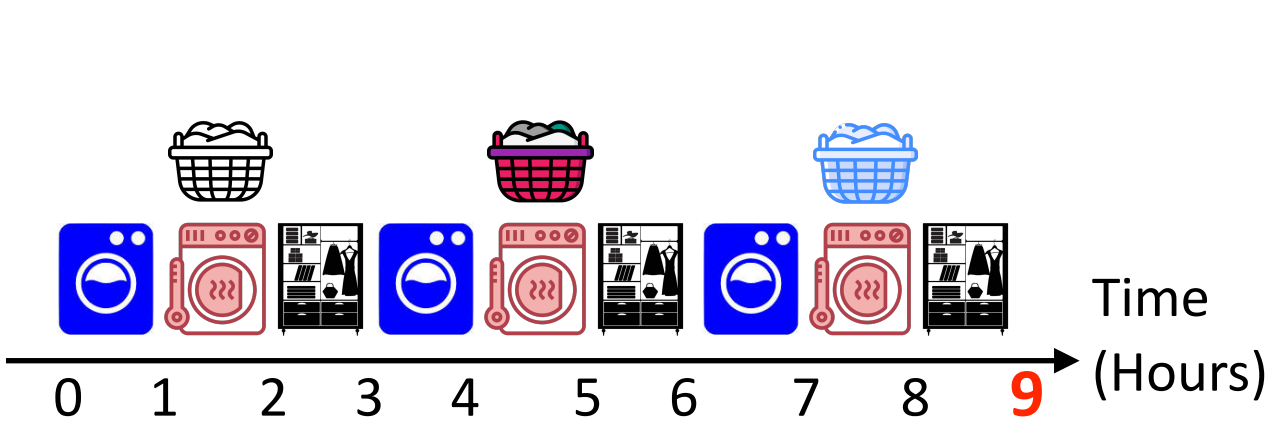
- Pipeline parallelism is a form of task parallelism, in which multiple tasks are overlapped in execution
- In our laundry analogy
  - We only need **5** hours to finish 3 tasks



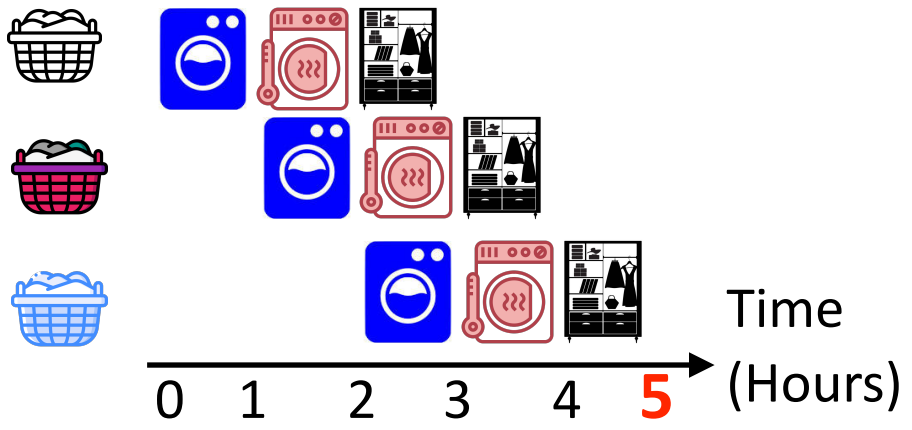
Why Do We Need Pipeline Parallelism?

- Pipeline parallelism **saves our time**
- In our laundry analogy
  - It saves us 4 hours

Without pipeline parallelism



With pipeline parallelism

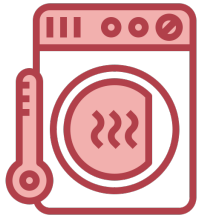


Outlines

- Know what pipeline parallelism is and why we need it
- Understand the challenges of existing libraries
- Introduce our solution Pipeflow
- Demonstrate how to use Pipeflow in applications
- Dive into the scheduling algorithm
- Conclude the talk

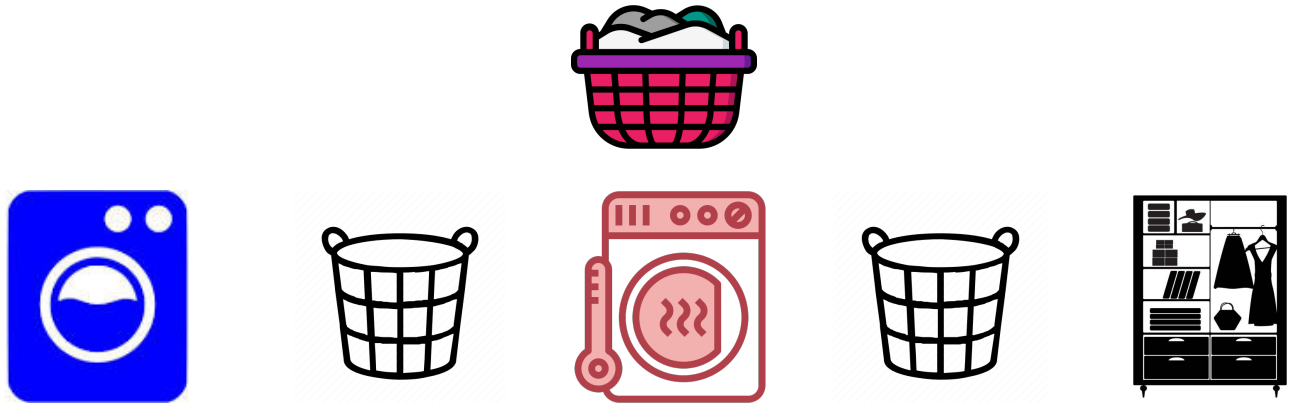
How Does Dryer Know Washer Is Done?

- In our laundry analogy
  - How does dryer know washer is done?
  - How does closet know dryer is done?
  - Dryer gets to dry out the clothes only when washer finishes.
  - Closet gets to fold the clothes only when dry finishes.



Use Buffers to Do Data Synchronization

- Store the result generated by the previous stage to a buffer such that the next stage can take the data from that buffer as input
- In our laundry analogy
  - We move the dirty clothes in our basket to washer
  - Washer places the clean clothes to Buffer 1



What Does Data Synchronization Look Like?

- Let's take oneTBB's laundry implementation as an example (https://github.com/oneapi-src/oneTBB)

```
oneapi::tbb::parallel_pipeline(
    lines,
    oneapi::tbb::make_filter<void, CleanWetClothes>
        (oneapi::tbb::filter_mode::serial_in_order,
         Washer()) &
    oneapi::tbb::make_filter<CleanWetClothes, CleanDryClothes>
        (oneapi::tbb::filter_mode::parallel,
         Dryer()) &
    oneapi::tbb::make_filter<CleanDryClothes, void>
        (oneapi::tbb::filter_mode::serial_in_order,
         Closet())
);
```

What Does Data Synchronization Look Like?

- Let's take oneTBB's laundry implementation as an example (https://github.com/oneapi-src/oneTBB)

```
1  oneapi::tbb::parallel_pipeline(




12 );
```

What Does Data Synchronization Look Like?

- Let's take oneTBB's laundry implementation as an example (https://github.com/oneapi-src/oneTBB)

```
1  oneapi::tbb::parallel_pipeline(
2      lines,
12 );
```

What Does Data Synchronization Look Like?

- Let's take oneTBB's laundry implementation as an example (https://github.com/oneapi-src/oneTBB)

```
1  oneapi::tbb::parallel_pipeline(

3      oneapi::tbb::make_filter<void, CleanWetClothes>
4          (oneapi::tbb::filter_mode::serial_in_order,
5           Washer()) &

12 );
```

What Does Data Synchronization Look Like?

- Let's take oneTBB's laundry implementation as an example (https://github.com/oneapi-src/oneTBB)

```
1  oneapi::tbb::parallel_pipeline(
6      oneapi::tbb::make_filter<CleanWetClothes, CleanDryClothes>
7          (oneapi::tbb::filter_mode::parallel,
8           Dryer()) &
12 );
```

What Does Data Synchronization Look Like?

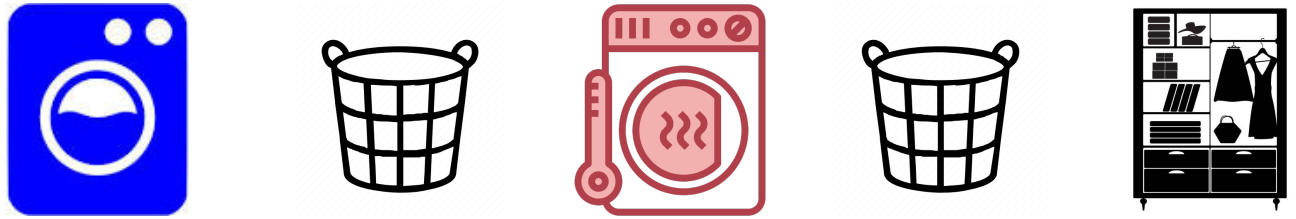
- Let's take oneTBB's laundry implementation as an example (https://github.com/oneapi-src/oneTBB)

```
1  oneapi::tbb::parallel_pipeline(
9      oneapi::tbb::make_filter<CleanDryClothes, void>
10         (oneapi::tbb::filter_mode::serial_in_order,
11          Closet())
12 );
```

Use Library's API to Define Application Data

- Define application data using library's API
- In our laundry analogy
  - I need to use the store's buffer

My basket Store's Buffer



Buffer 1 Buffer 2

Challenges of Existing Libraries

- Users have their own application data
- User do not want to define the application data using library's API
- Users want to have full control over their application data
- Users need a new **task-parallel pipeline scheduling framework**

Outlines

- Know what pipeline parallelism is and why we need it
- Understand the challenges of existing libraries
- Introduce our solution Pipeflow
- Demonstrate how to use Pipeflow in an application
- Dive into the scheduling algorithm
- Conclude the talk

Pipeflow: Task-parallel Pipeline

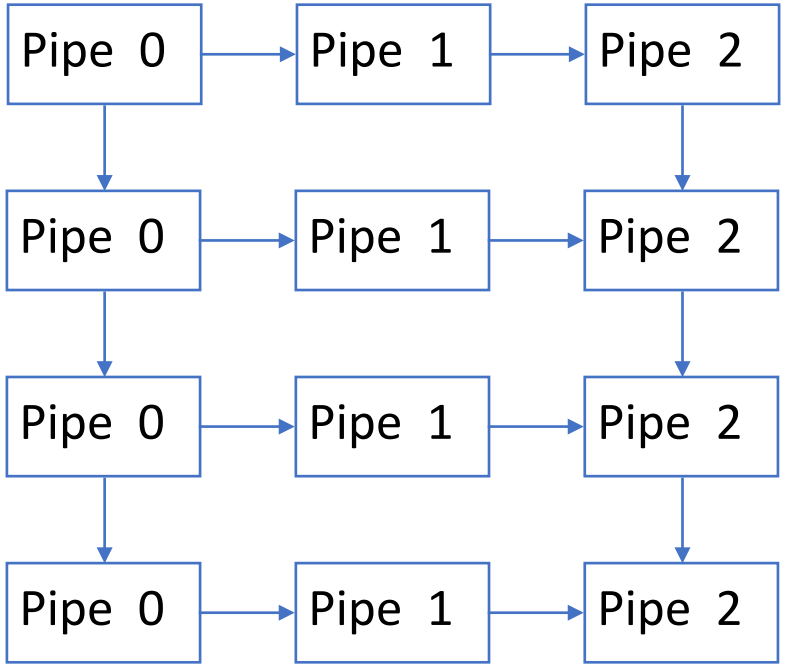
- Pipeflow does not use buffers to manage application data
- Pipeflow describes pipeline parallelism as a task dependency graph
- Pipeflow uses std::atomic operations to synchronize and schedule tasks

Pipeflow: Task-parallel Pipeline

- Pipeflow does not use buffers to manage application data
- Pipeflow describes pipeline parallelism as a task dependency graph
- Pipeflow uses std::atomic operations to synchronize and schedule tasks

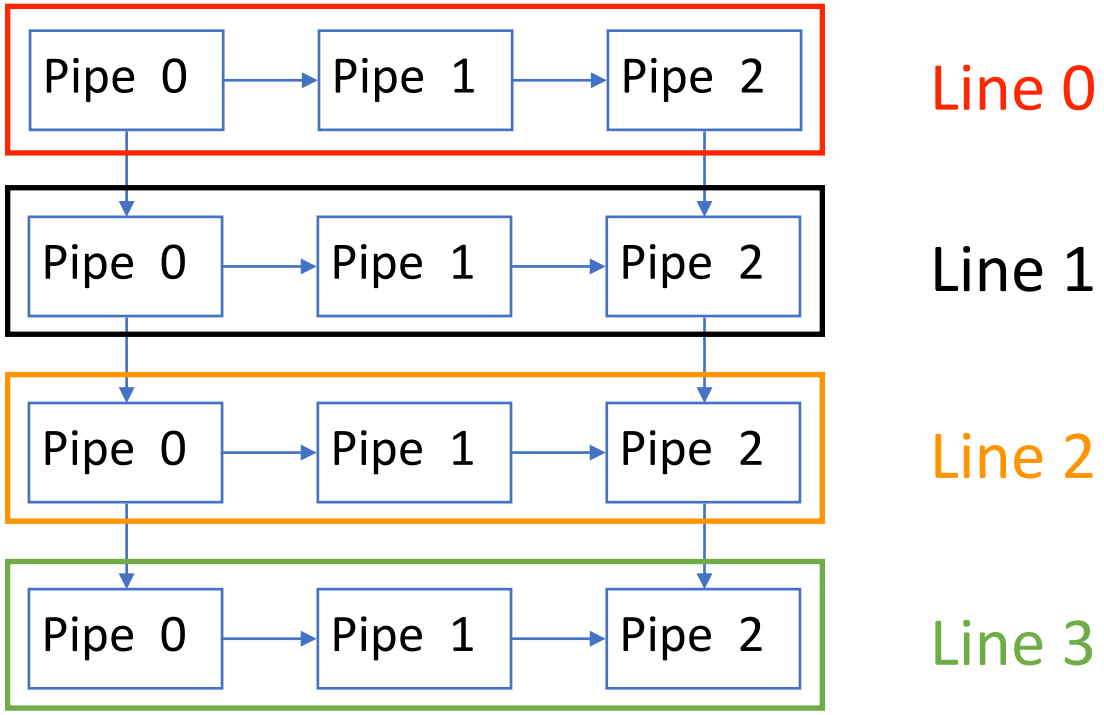
Task Dependency Graph in Pipeflow

- Task dependency graph is a directed acyclic graph (DAG)



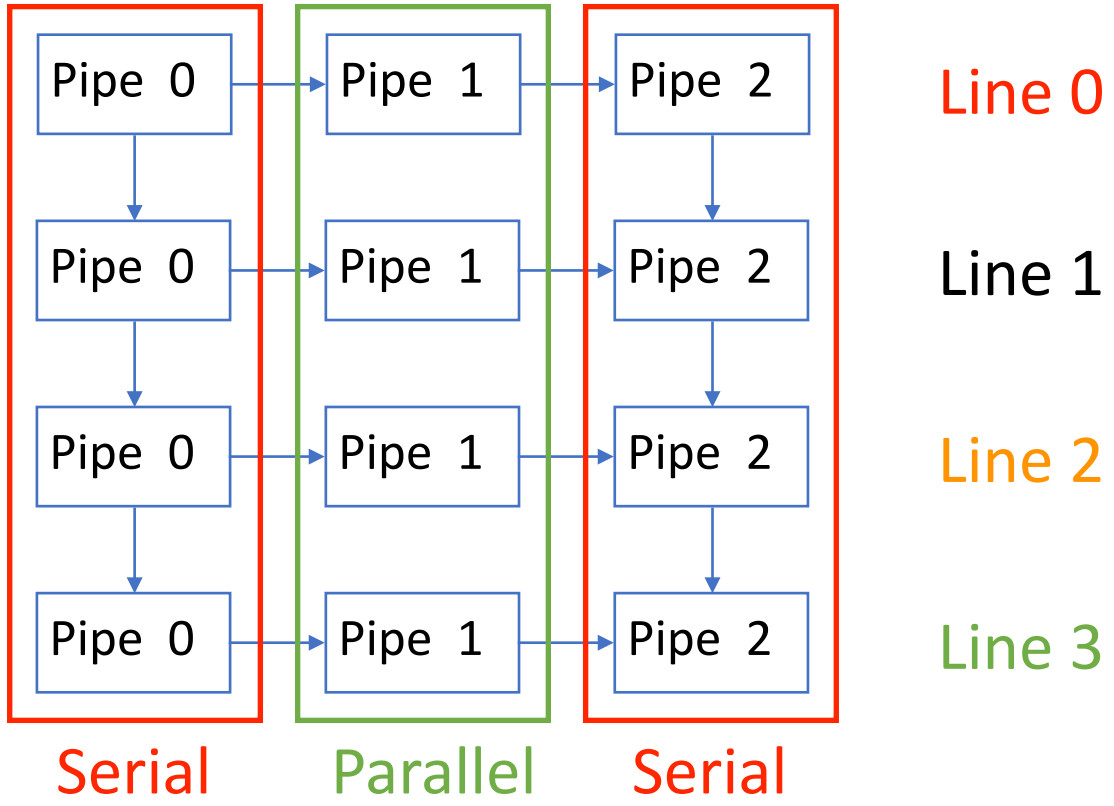
Line Is a Chain of Pipes

- Refer a chain of pipes (stages) as a line



Pipes Could be of Serial or Parallel Pipes

- 3 pipes are Serial-Parallel-Serial (SPS)

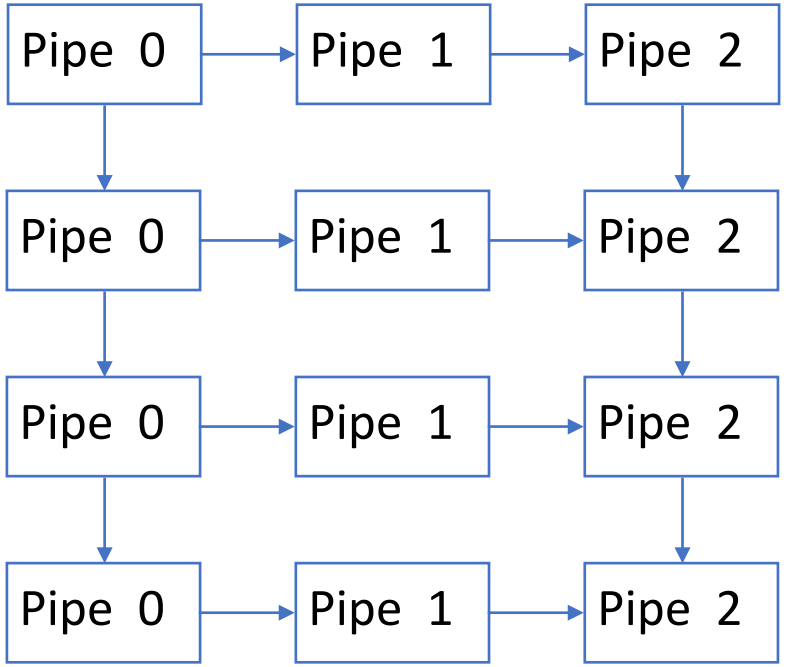


Pipeflow: Task-parallel Pipeline

- Pipeflow does not use buffers to manage application data
- Pipeflow describes pipeline parallelism as a task dependency graph
- Pipeflow uses std::atomic operations to synchronize and schedule tasks

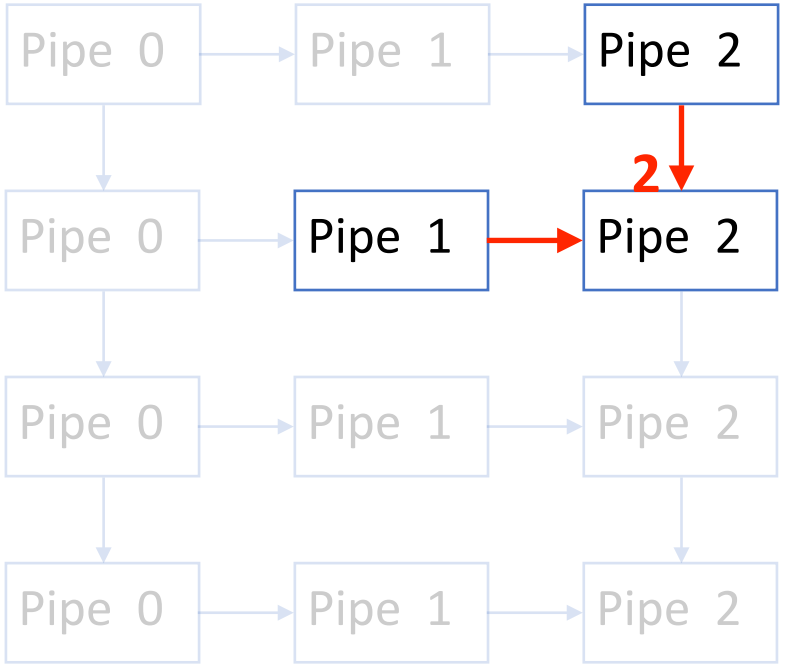
Use Atomic Counters to Represent Dependencies

- Every pipe has an atomic counter denoting the dependencies between pipes



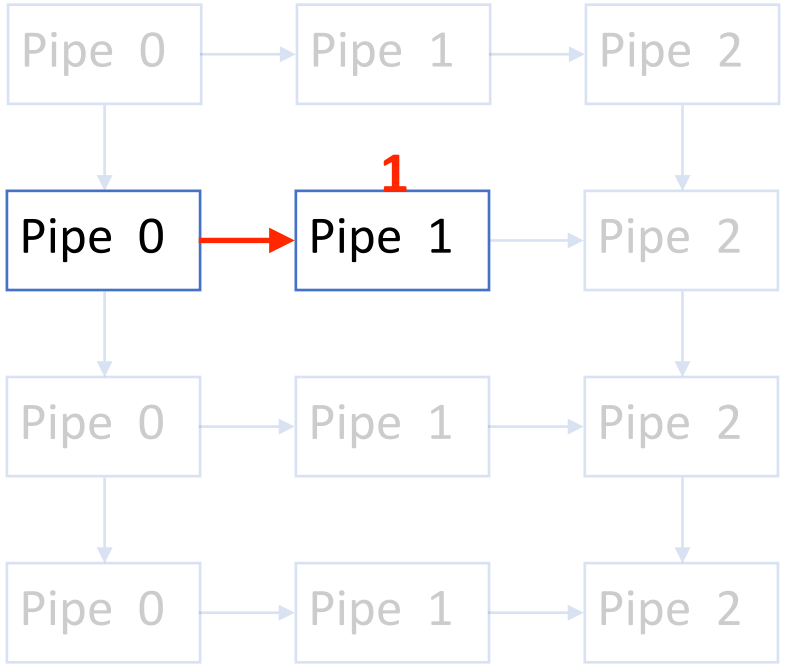
Atomic Counter of a Serial Pipe Is 2

- Serial pipe has an atomic counter equals to 2



Atomic Counter of a Parallel Pipe Is 1

- Parallel pipe has an atomic counter equals to 1

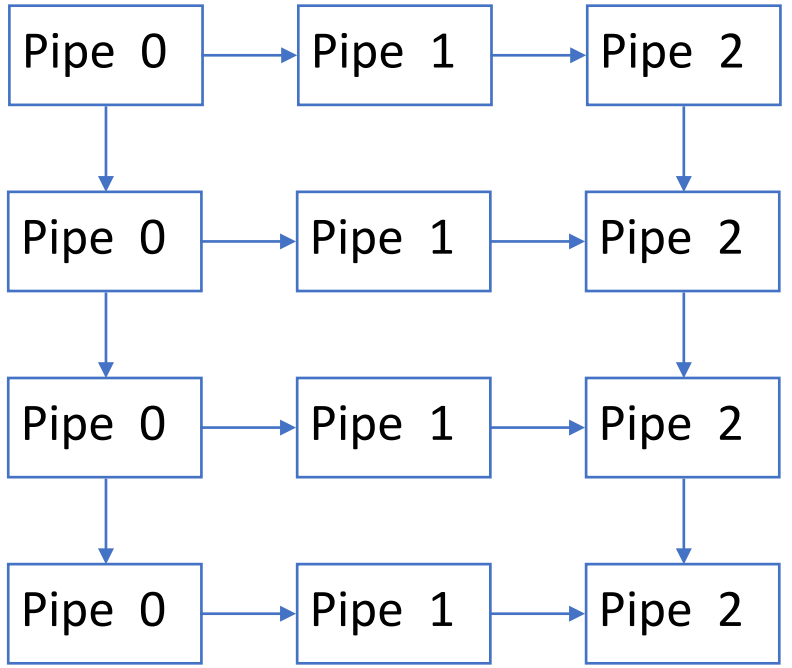


Pipeflow: Task-parallel Pipeline

- Pipeflow does not use buffers to manage application data
- Pipeflow describes pipeline parallelism as a task dependency graph
- Pipeflow uses std::atomic operations to synchronize and schedule tasks

Construct Pipeline in Pipeflow

- To construct a pipeline in Pipeflow, we need to
  - Define the number of lines
  - Define the pipe type (serial or parallel)
  - Define the pipe callable



Define Pipeflow atop Taskflow

- Pipeflow is a task-parallel pipeline scheduling framework
- We leverage Taskflow's work-stealing runtime to achieve load balancing
  - Taskflow is a general-purpose parallel and heterogeneous task programming system
  - https://github.com/taskflow/taskflow
  - https://www.youtube.com/watch?v=MX15huP5DsM, CppCon 2020
- Pipeflow is integrated in Taskflow project

Outlines

- Know what pipeline parallelism is and why we need it
- Understand the challenges of existing libraries
- Introduce our solution Pipeflow
- Demonstrate how to use Pipeflow in an application
- Dive into the scheduling algorithm
- Conclude the talk

Text Processing Pipeline

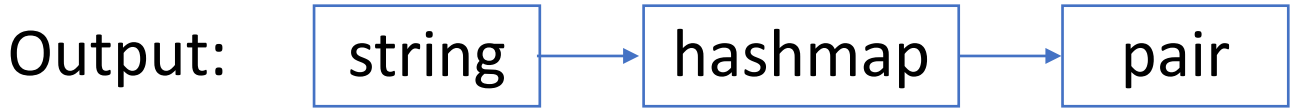
- The text processing pipeline finds the most frequent character of each string from an input vector

```
# input strings
abade
ddddf
eefge
xyzzd
ijjjj
jiiii
kkijk
```

```
# output
a:2
d:4
e:3
z:2
j:4
i:4
k:3
```

Text Processing Pipeline

- We use 4 lines in this text processing pipeline
- Pipe 0 is a serial pipe that stores an input string in the application data structure
- Pipe 1 is a parallel pipe that counts the frequency of each character in a string and stores that in a hashmap
- Pipe 2 is a serial pipe that finds the most frequent character in a hashmap and prints out the result



Text Processing Pipeline Implementation

```
1   #include <taskflow/taskflow.hpp>
2   #include <taskflow/algorithm/pipeline.hpp>
3
```

Include headers

Text Processing Pipeline Implementation

Function to print a hashmap

```
4   // Function: format the map
5   std::string format_map(const std::unordered_map<char, size_t>& map) {
6       std::ostringstream oss;
7       for(const auto& [i, j] : map) {
8           oss << i << ':' << j << ' ';
9       }
10      return oss.str();
11  }
```

Text Processing Pipeline Implementation

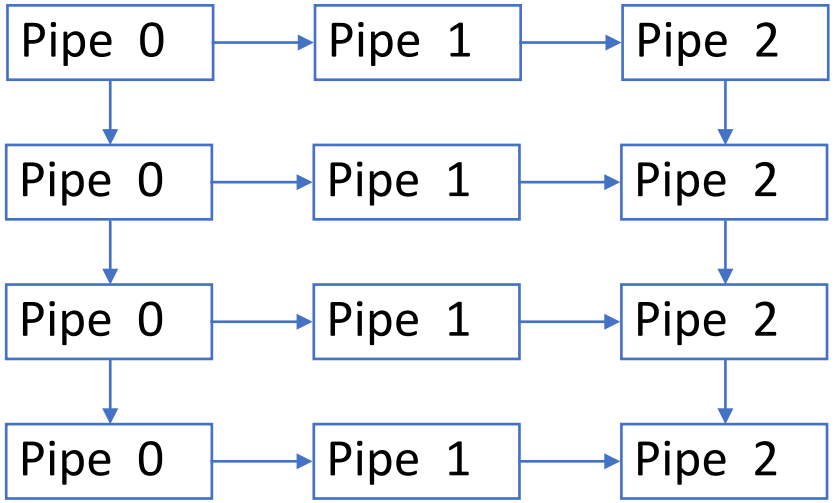
```
13  int main() {
14
15      tf::Taskflow taskflow;
16      tf::Executor executor;
17
```

Define taskflow and executor objects

Text Processing Pipeline Implementation

```
17
18      const size_t num_lines = 4;
19
```

Define the number of concurrent lines



Text Processing Pipeline Implementation

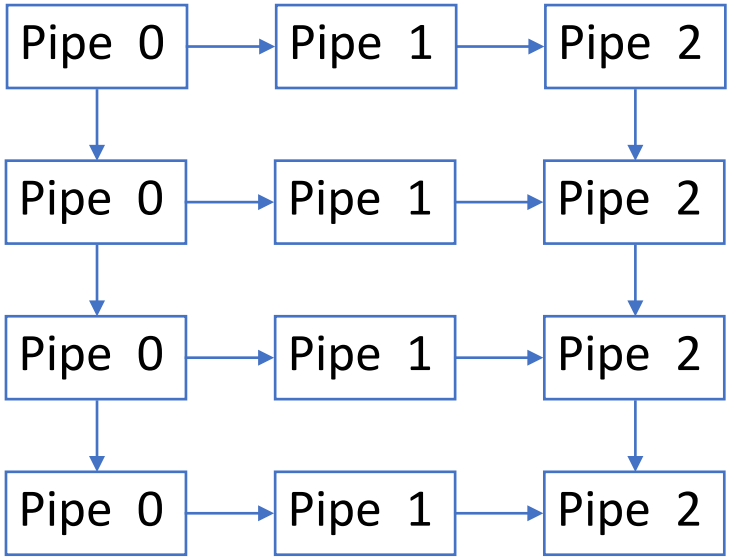
```
20      // input data
21      std::vector<std::string> input = {
22          "abade",
23          "ddddf",
24          "eefge",
25          "xyzzd",
26          "ijjjj",
27          "jiiii",
28          "kkijk"
29      };
30
```

Define an input vector which has 7 strings

Text Processing Pipeline Implementation

```
31      // custom data storage
32      using data_type = std::variant<
33          std::string, std::unordered_map<char, size_t>, std::pair<char, size_t>>;
34      std::array<data_type, num_lines> mybuffer;
35
```

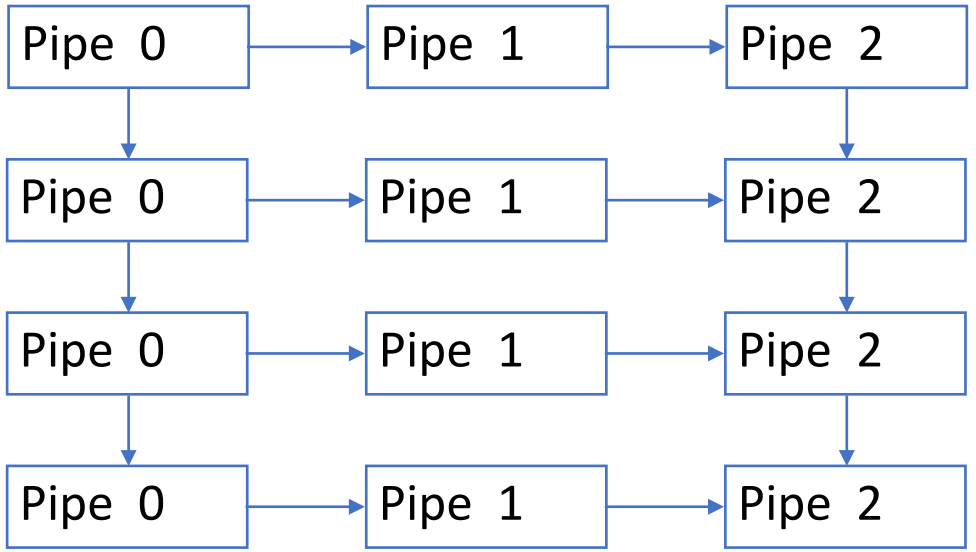
Application defines its own data storage



Text Processing Pipeline Implementation

```
36      // the pipeline consists of three pipes (SPS)
37      // and up to four concurrent scheduling tokens
38      tf::Pipeline pl(num_lines,
39
```

Specify the number of lines



Text Processing Pipeline Implementation

Define the first pipe

```
39
40          // first pipe processes the input data
41          tf::Pipe{tf::PipeType::SERIAL, [&](tf::Pipeflow& pf) {
42              if(pf.token() == input.size()) {
43                  pf.stop();
44              }
45              else {
46                  printf("stage 1: input token = %s\n", input[pf.token()].c_str());
47                  mybuffer[pf.line()] = input[pf.token()];
48              }
49          }},
50
```

Text Processing Pipeline Implementation

Specify the first pipe as a SERIAL pipe

```
39
40          // first pipe processes the input data
41          tf::Pipe{tf::PipeType::SERIAL,
```

Text Processing Pipeline Implementation

Define the pipe callable which takes a pipeflow object pf

```
39
40          // first pipe processes the input data
41                                      , [&](tf::Pipeflow& pf) {
42              if(pf.token() == input.size()) {
43                  pf.stop();
44              }
45              else {
46                  printf("stage 1: input token = %s\n", input[pf.token()].c_str());
47                  mybuffer[pf.line()] = input[pf.token()];
48              }
49          }},
```

Text Processing Pipeline Implementation

Specify the termination condition

```
39
40          // first pipe processes the input data
41                                          , [&](tf::Pipeflow& pf) {
42              if(pf.token() == input.size()) {
43                  pf.stop();
44              }
45
46
47
48
49          }},
```

Text Processing Pipeline Implementation

Read the input string and store it in application's storage mybuffer

```
39
40          // first pipe processes the input data
41                                      , [&](tf::Pipeflow& pf) {
45              else {
46                  printf("stage 1: input token = %s\n", input[pf.token()].c_str());
47                  mybuffer[pf.line()] = input[pf.token()];
48              }
49          }},
50
```

Text Processing Pipeline Implementation

```
51          // second pipe counts the frequency of each character
52          tf::Pipe{tf::PipeType::PARALLEL, [&](tf::Pipeflow& pf) {
53              std::unordered_map<char, size_t> map;
54              for(auto c : std::get<std::string>(mybuffer[pf.line()])) {
55                  map[c]++;
56              }
57
58              printf("stage 2: map = %s\n", format_map(map).c_str());
59              mybuffer[pf.line()] = map;
60          }},
61
```

Define the second pipe

Text Processing Pipeline Implementation

```
51          // second pipe counts the frequency of each character
52          tf::Pipe{tf::PipeType::PARALLEL,
```

Specify the second pipe as a PARALLEL pipe

Text Processing Pipeline Implementation

```
51          // second pipe counts the frequency of each character
52                                          , [&](tf::Pipeflow& pf) {
53              std::unordered_map<char, size_t> map;
54              for(auto c : std::get<std::string>(mybuffer[pf.line()])) {
55                  map[c]++;
56              }
57
```

Count the frequency of each character

Text Processing Pipeline Implementation

```
51        // second pipe counts the frequency of each character
52                                          , [&](tf::Pipeflow& pf) {
57
58            printf("stage 2: map = %s\n", format_map(map).c_str());
59            mybuffer[pf.line()] = map;
60        }},
61
```

Save the results in mybuffer at the corresponding entry

Text Processing Pipeline Implementation

```
62          // third pipe finds the most frequent character
63          tf::Pipe{tf::PipeType::SERIAL, [&mybuffer](tf::Pipeflow& pf) {
64              auto& map = std::get<std::unordered_map<char, size_t>>(mybuffer[pf.line()]);
65              auto sol = std::max_element(map.begin(), map.end(), [](auto& a, auto& b){
66                  return a.second < b.second;
67              });
68
69              printf("stage 3: %c:%zu\n", sol->first, sol->second);
70              // not necessary to store the last-stage data, just for demo purpose
71              mybuffer[pf.line()] = *sol;
72          }}
73      );
74
```

Define the third pipe

Text Processing Pipeline Implementation

```
62          // third pipe finds the most frequent character
63          tf::Pipe{tf::PipeType::SERIAL,
```

Specify the third pipe to be a SERIAL pipe

Text Processing Pipeline Implementation

```
62          // third pipe finds the most frequent character
63                                      , [&mybuffer](tf::Pipeflow& pf) {
64              auto& map = std::get<std::unordered_map<char, size_t>>(mybuffer[pf.line()]);
65              auto sol = std::max_element(map.begin(), map.end(), [](auto& a, auto& b){
66                  return a.second < b.second;
67              });
68
```

Find the most frequent character

Text Processing Pipeline Implementation

```
62          // third pipe finds the most frequent character
63                                          , [&mybuffer](tf::Pipeflow& pf) {
69              printf("stage 3: %c:%zu\n", sol->first, sol->second);
70              // not necessary to store the last-stage data, just for demo purpose
71              mybuffer[pf.line()] = *sol;
72          }}
73      );
74
```

Save the result to mybuffer

Text Processing Pipeline Implementation

```
75      tf::Task task = taskflow.composed_of(pl);
76
```

We create a module task from the pipeline object pl through Taskflow's method composed_of

https://taskflow.github.io/taskflow/ComposableTasking.html

Text Processing Pipeline Implementation

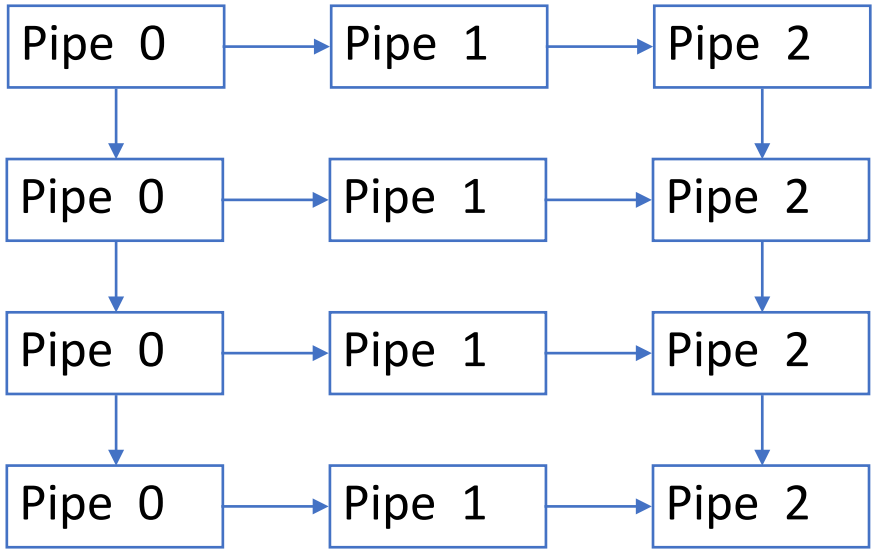
```
77      // run the pipeline
78      executor.run(taskflow).wait();
79
80      return 0;
81  }
```

Submit the taskflow graph to the executor and run it

One Possible Outcome

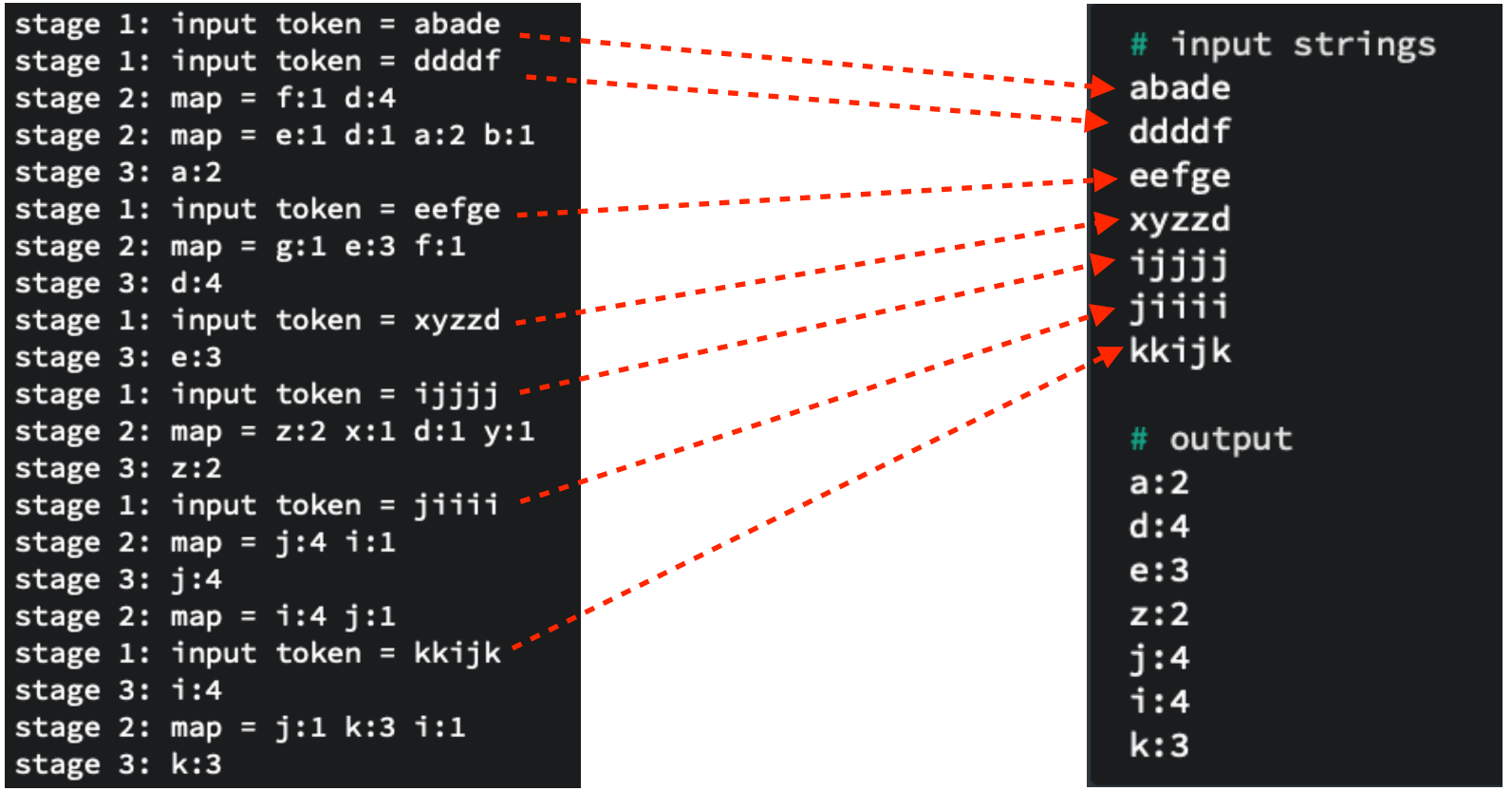
- You may see different outcome because pipes run in parallel

```
stage 1: input token = abade
stage 1: input token = ddddf
stage 2: map = f:1 d:4
stage 2: map = e:1 d:1 a:2 b:1
stage 3: a:2
stage 1: input token = eefge
stage 2: map = g:1 e:3 f:1
stage 3: d:4
stage 1: input token = xyzzd
stage 3: e:3
stage 1: input token = ijjjj
stage 2: map = z:2 x:1 d:1 y:1
stage 3: z:2
stage 1: input token = jiiii
stage 2: map = j:4 i:1
stage 3: j:4
stage 2: map = i:4 j:1
stage 1: input token = kkijk
stage 3: i:4
stage 2: map = j:1 k:3 i:1
stage 3: k:3
```



One Possible Outcome

- First pipe runs in serial



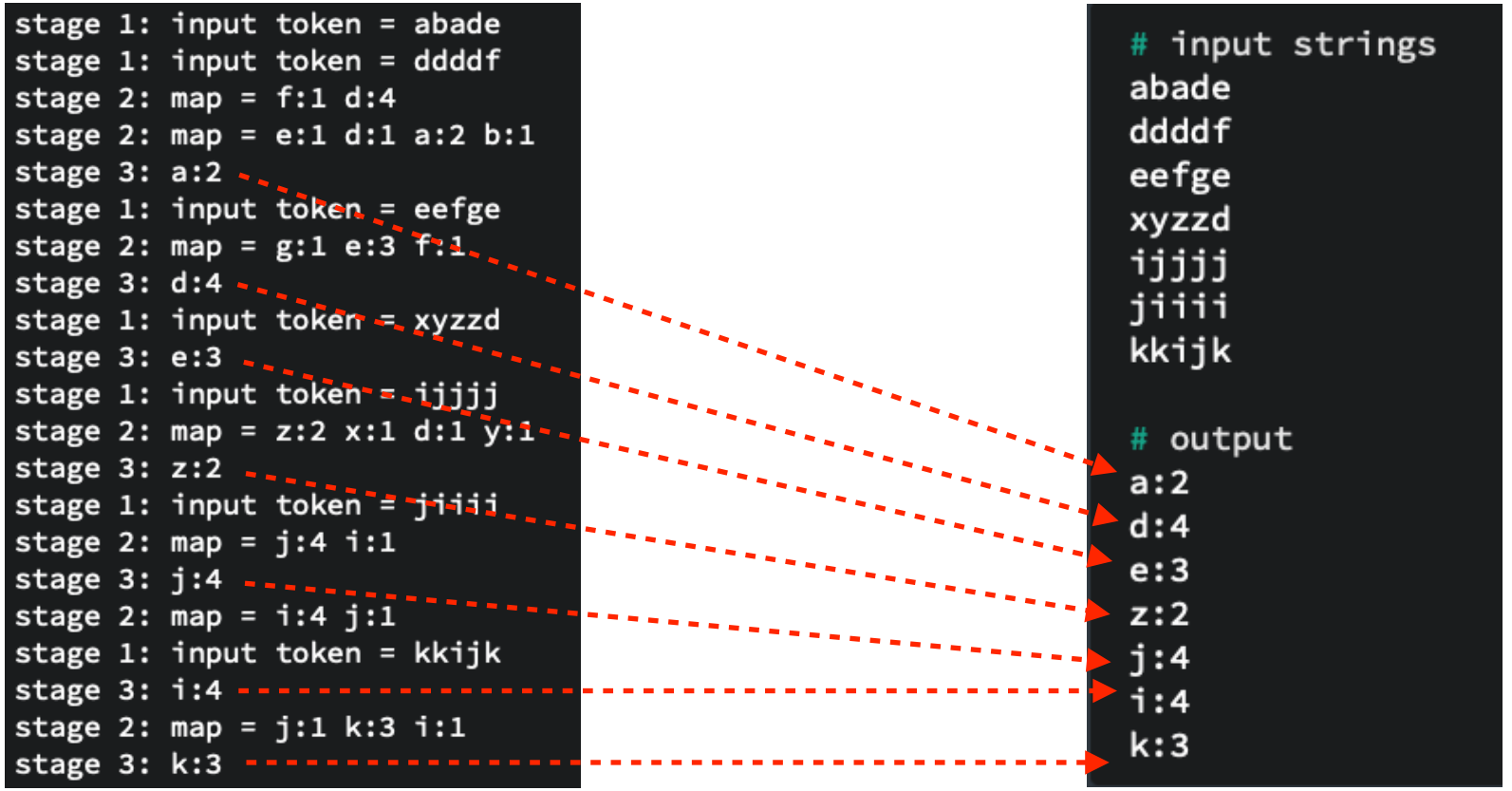
One Possible Outcome

- Second pipe runs in parallel

```
stage 1: input token = abade
stage 1: input token = ddddf
stage 2: map = f:1 d:4
stage 2: map = e:1 d:1 a:2 b:1
stage 3: a:2
stage 1: input token = eefge
stage 2: map = g:1 e:3 f:1
stage 3: d:4
stage 1: input token = xyzzd
stage 3: e:3
stage 1: input token = ijjjj
stage 2: map = z:2 x:1 d:1 y:1
stage 3: z:2
stage 1: input token = jiiii
stage 2: map = j:4 i:1
stage 3: j:4
stage 2: map = i:4 j:1
stage 1: input token = kkijk
stage 3: i:4
stage 2: map = j:1 k:3 i:1
stage 3: k:3
```

One Possible Outcome

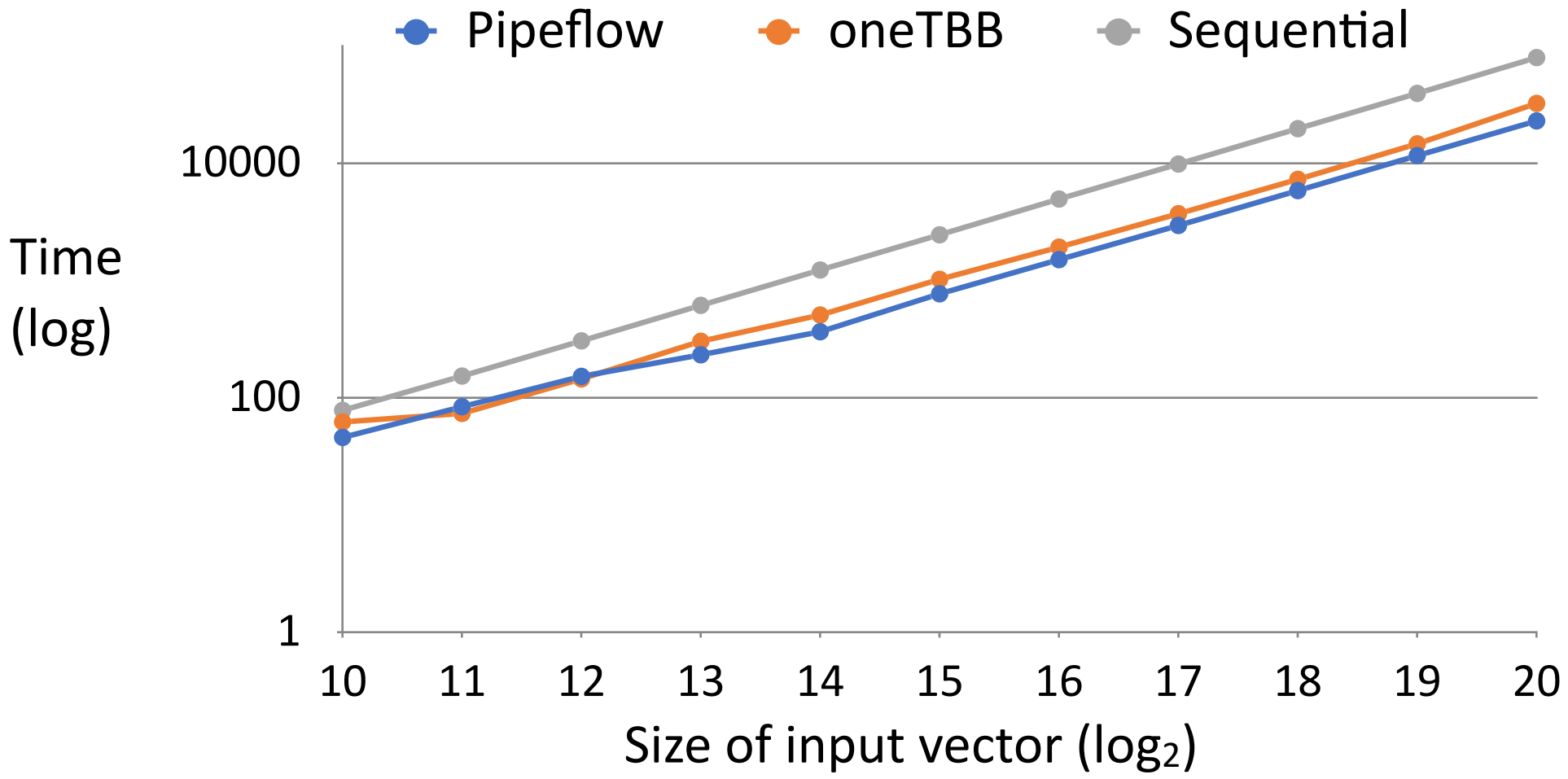
- Third pipe runs in serial



Performance Comparison

- Compare performance between Pipeflow, oneTBB, and a sequential version
- Environment
  - Intel Xeon CPU cores @ 2.00 G Hz
  - 256 GB RAM
  - Linux-5.3.0-64-generic
  - g++ 9.2.1
  - -O3 enabled
- Input vector
  - 2^10 ~ 2^20 strings
  - Each string consists of 10,000 characters

Performance Comparison



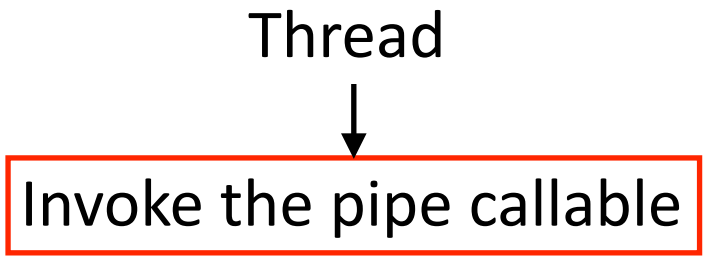
Outlines

- Know what pipeline parallelism is and why we need it
- Understand the challenges of existing libraries
- Introduce our solution Pipeflow
- Demonstrate how to use Pipeflow in an application
- Dive into the scheduling algorithm
- Conclude the talk

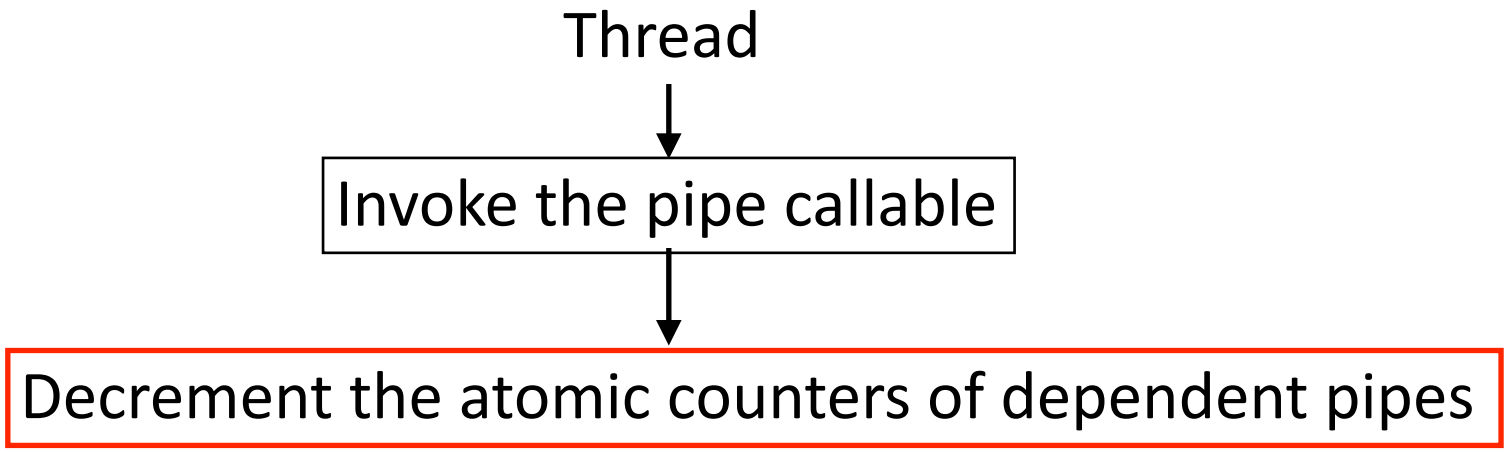
Scheduling Algorithm per Thread

Thread

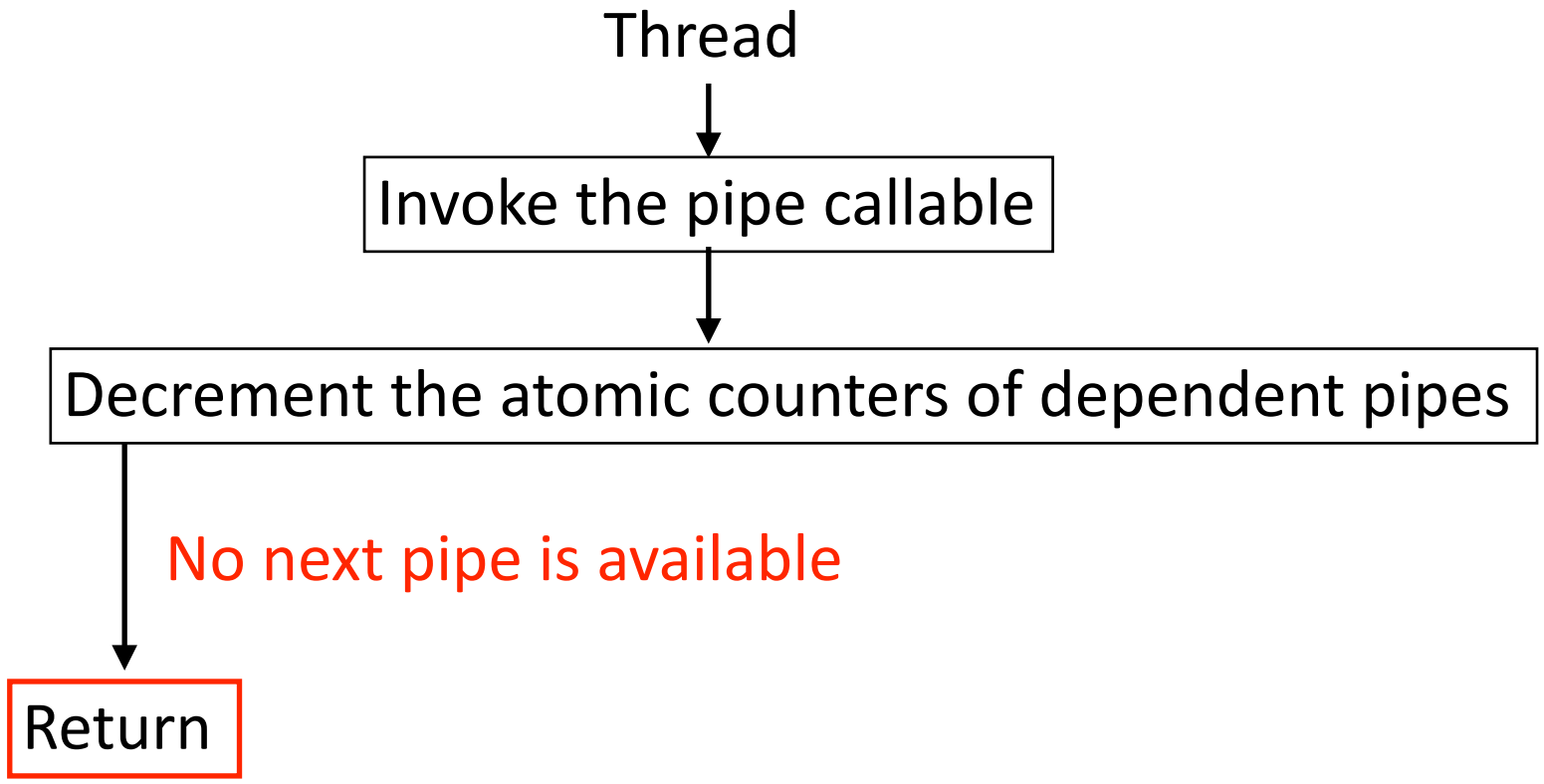
Scheduling Algorithm per Thread



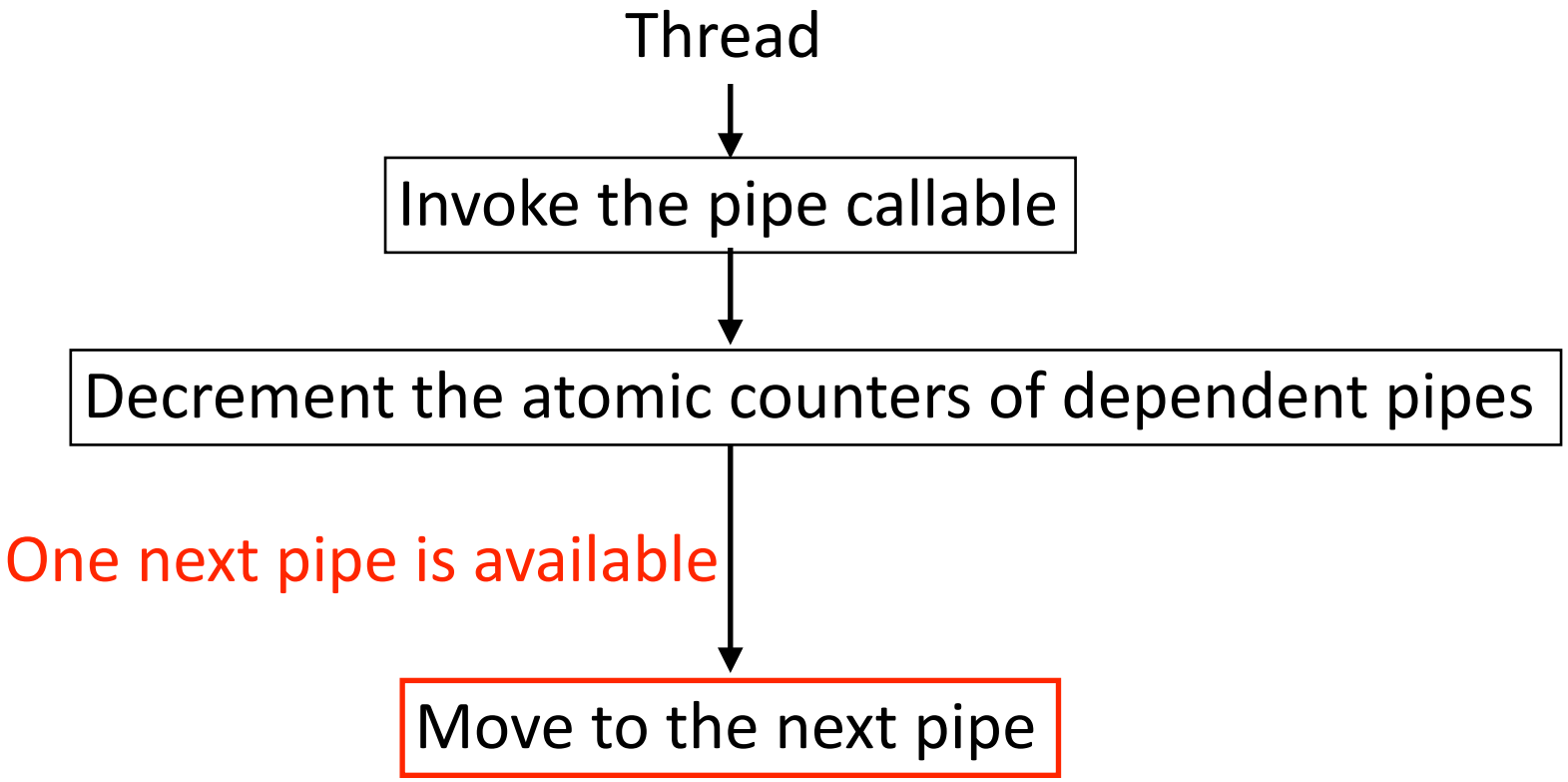
Scheduling Algorithm per Thread



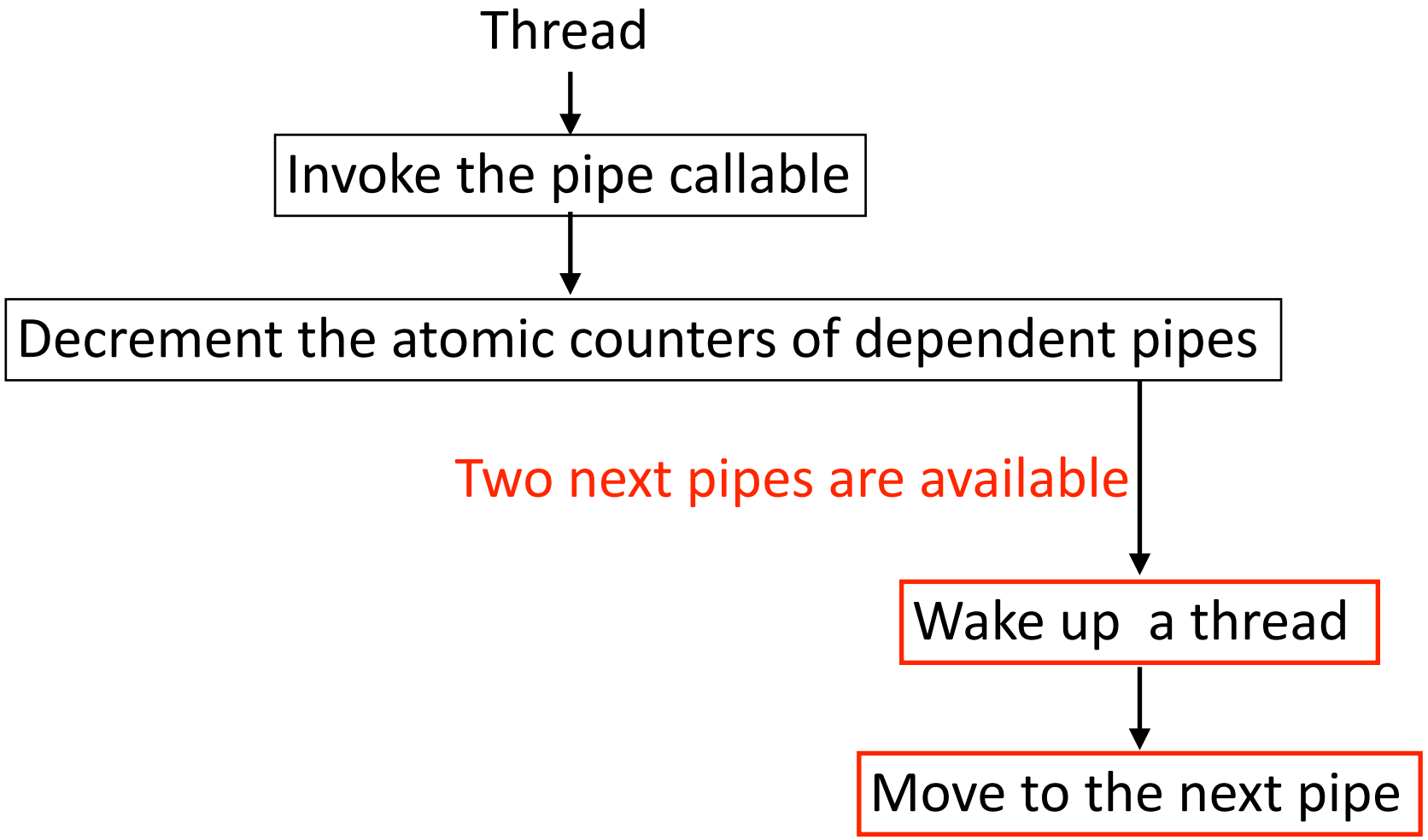
Scheduling Algorithm per Thread



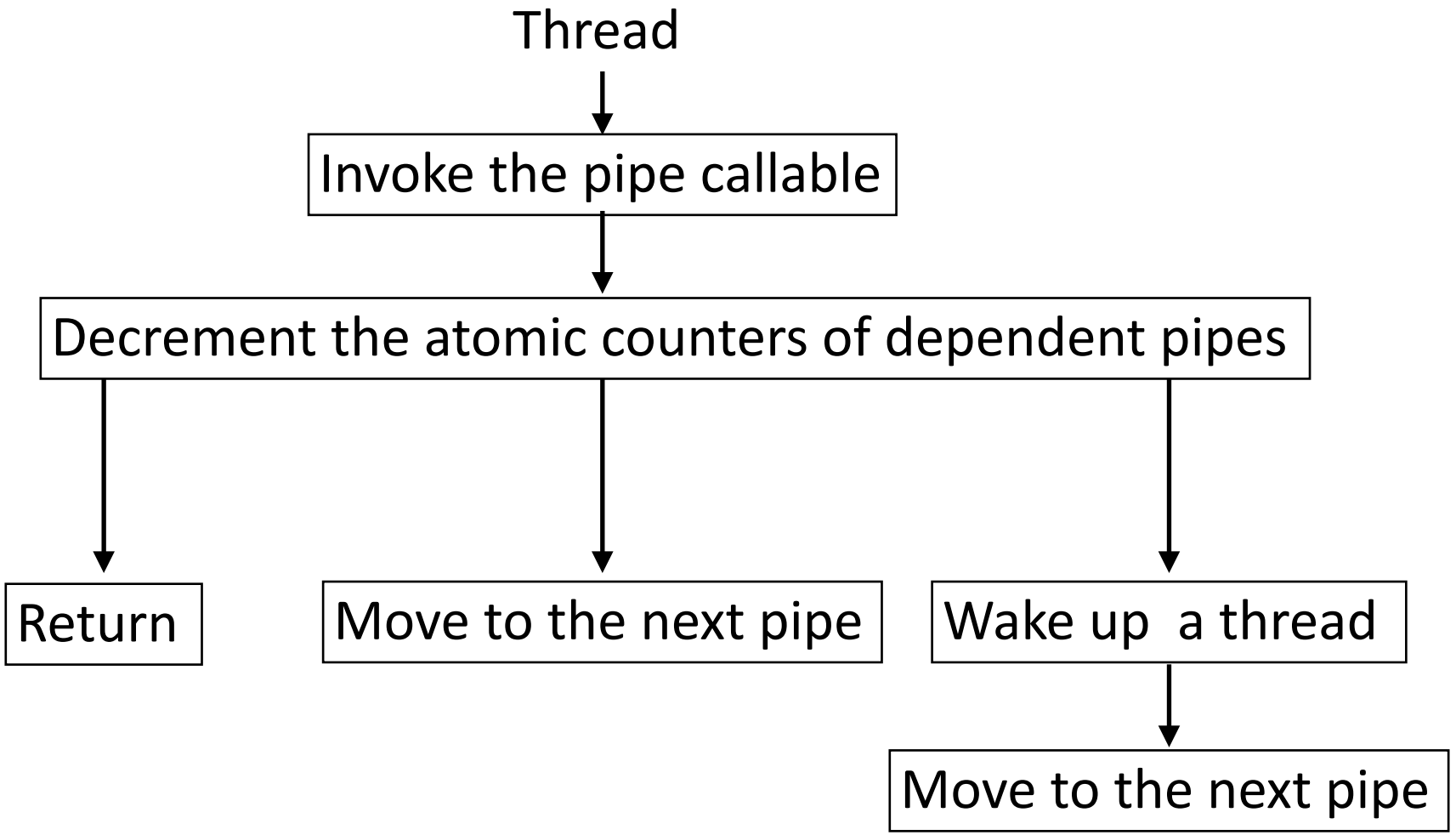
Scheduling Algorithm per Thread



Scheduling Algorithm per Thread

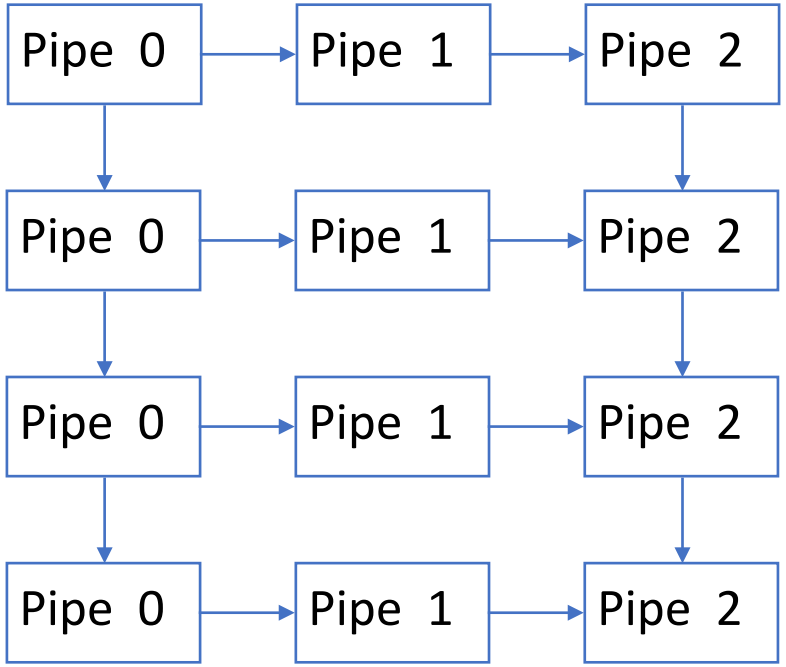


Scheduling Algorithm per Thread



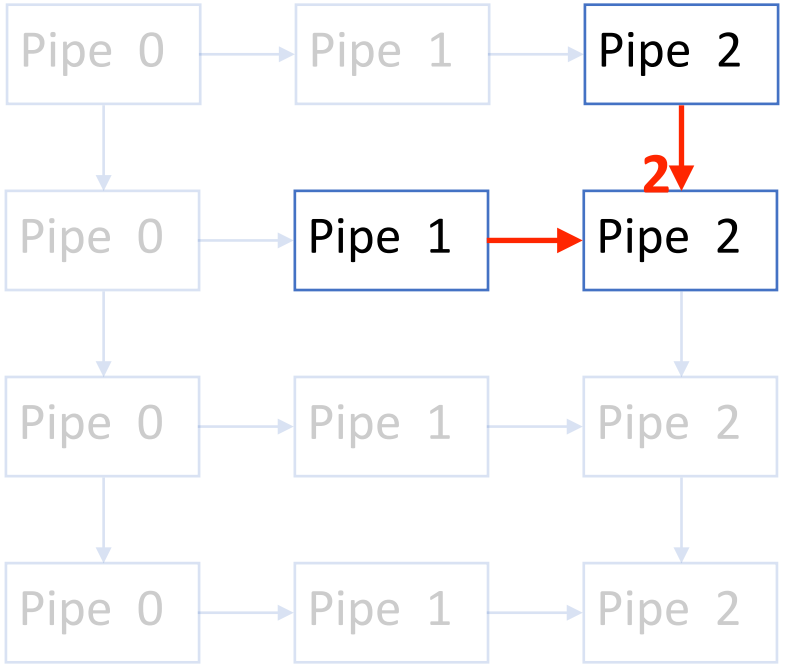
Atomic Counters Are Dependencies

- Every pipe has an atomic counter denoting its dependencies between pipes



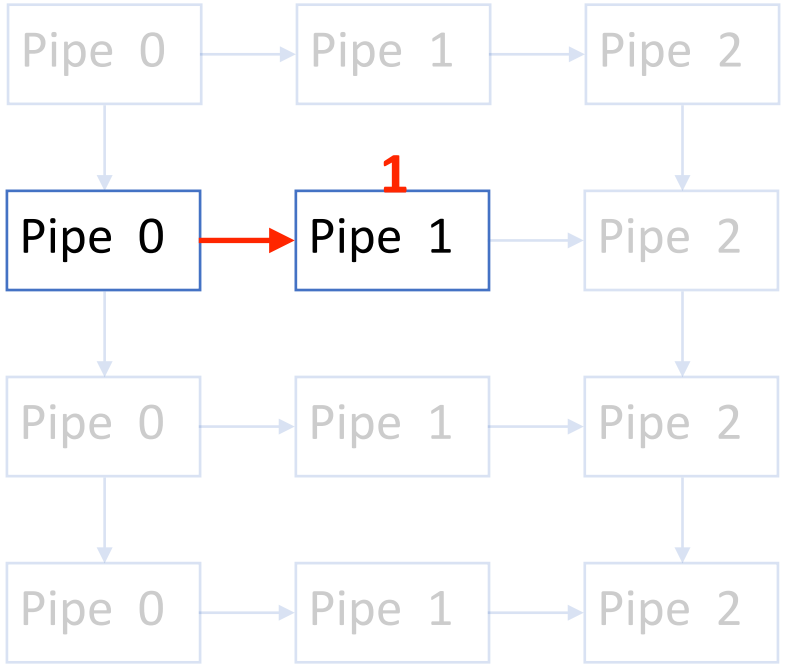
Atomic Counter of a Serial Pipe Is 2

- Serial pipe has an atomic counter equals to 2



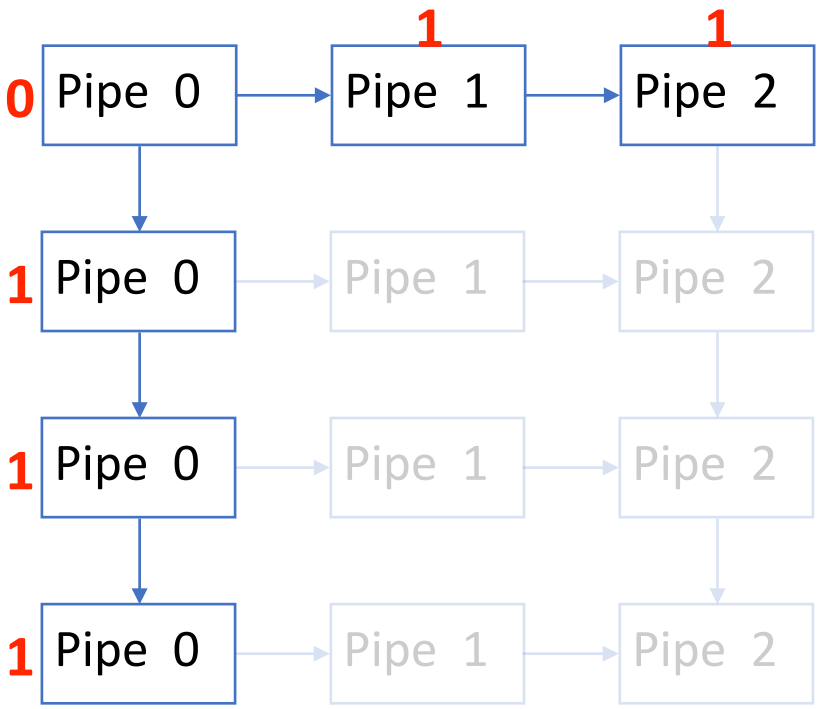
Atomic Counter of a Parallel Pipe Is 1

- Parallel pipe has an atomic counter equals to 1

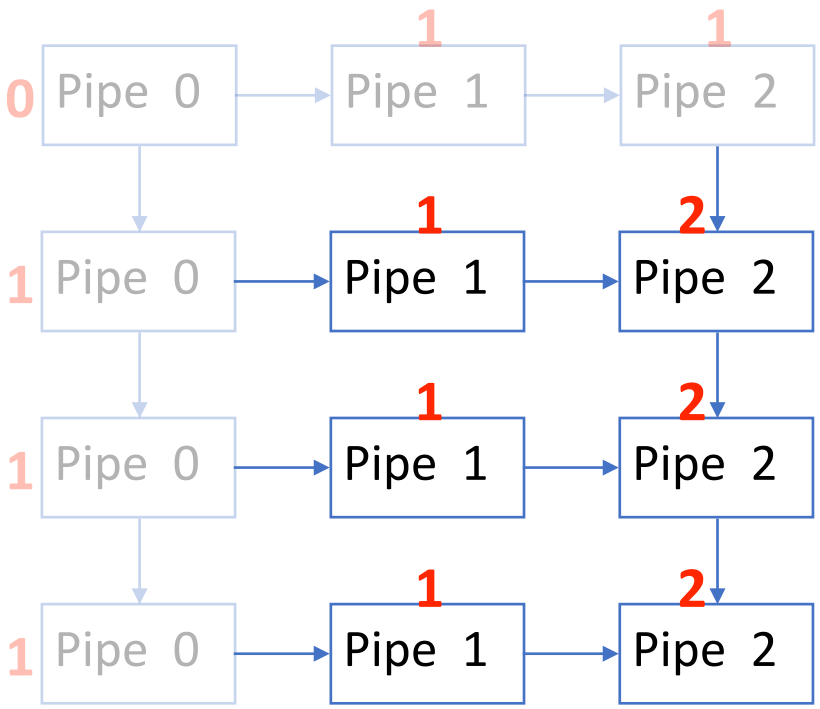


Initial Atomic Counters

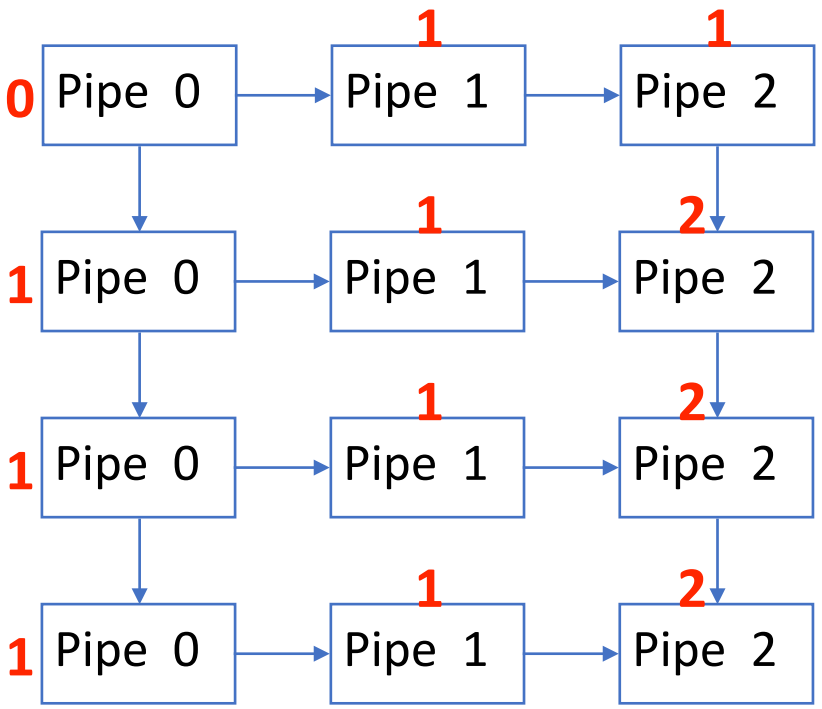
- Initial atomic counters for serial pipes at the border are different



Initial Atomic Counters

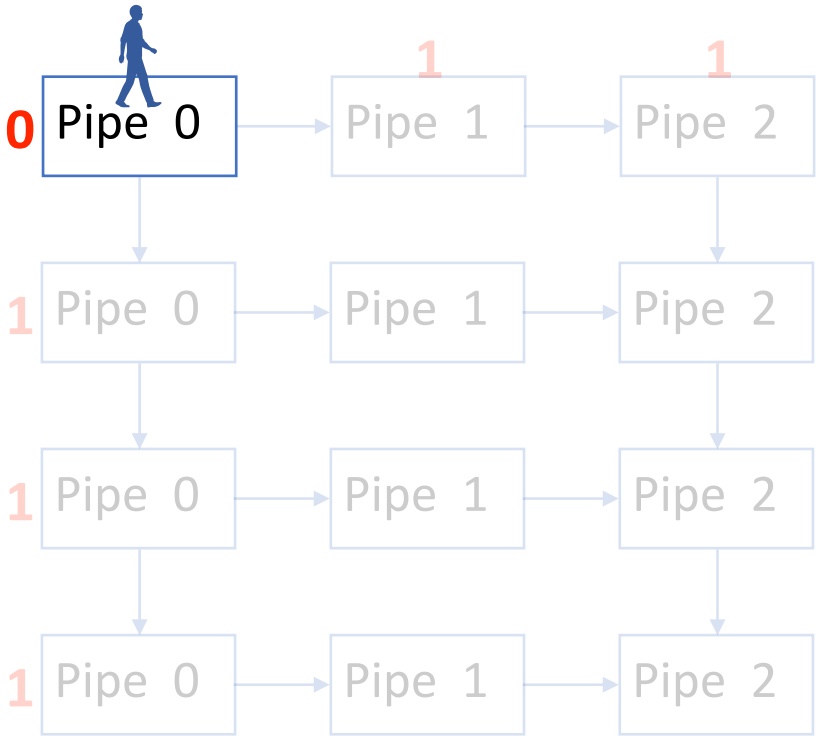


Initial Atomic Counters

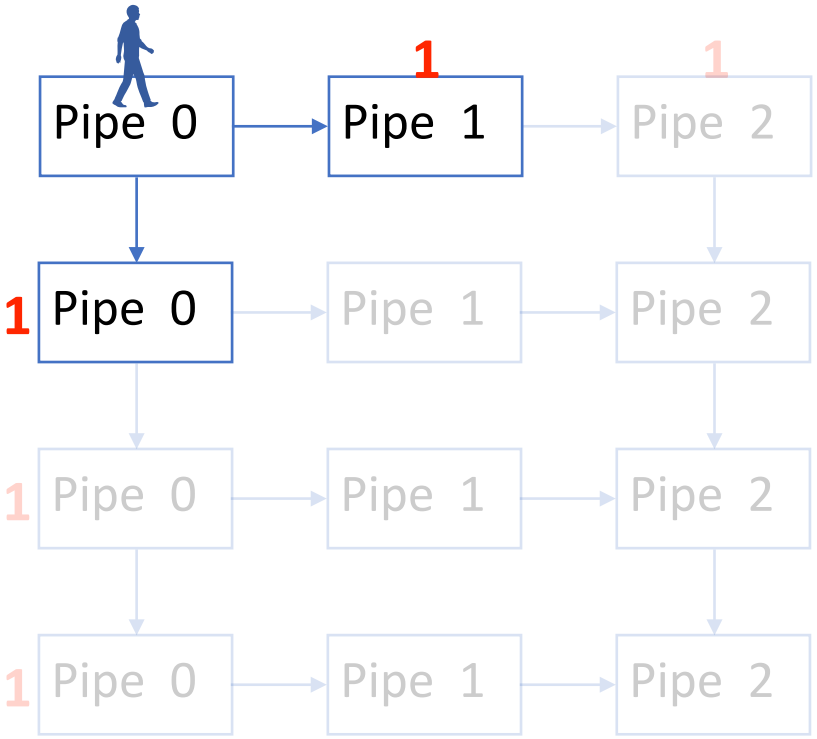


Scheduling Algorithm Walkthrough

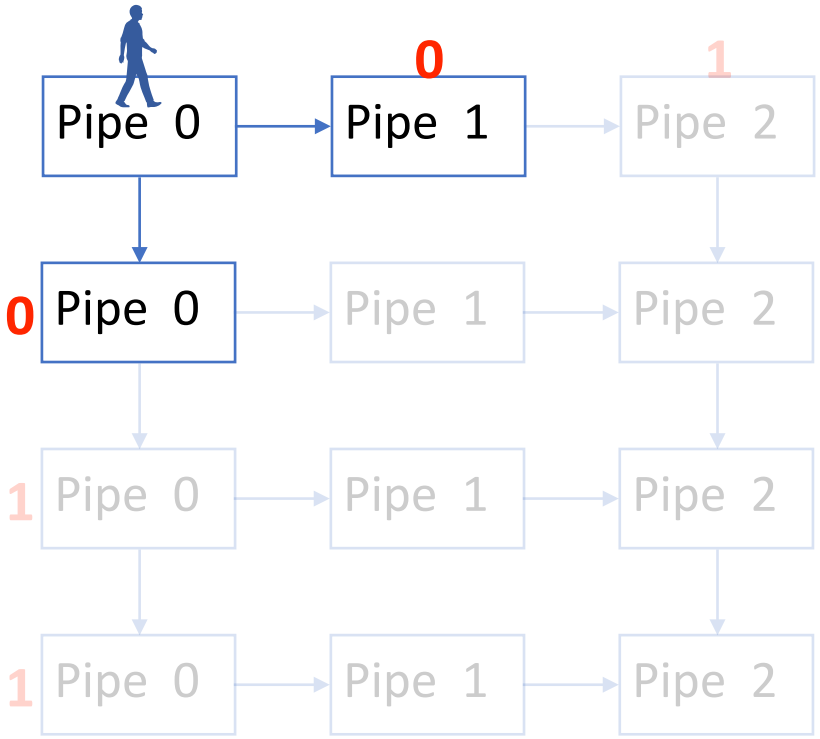
- **Thread is represented as** 



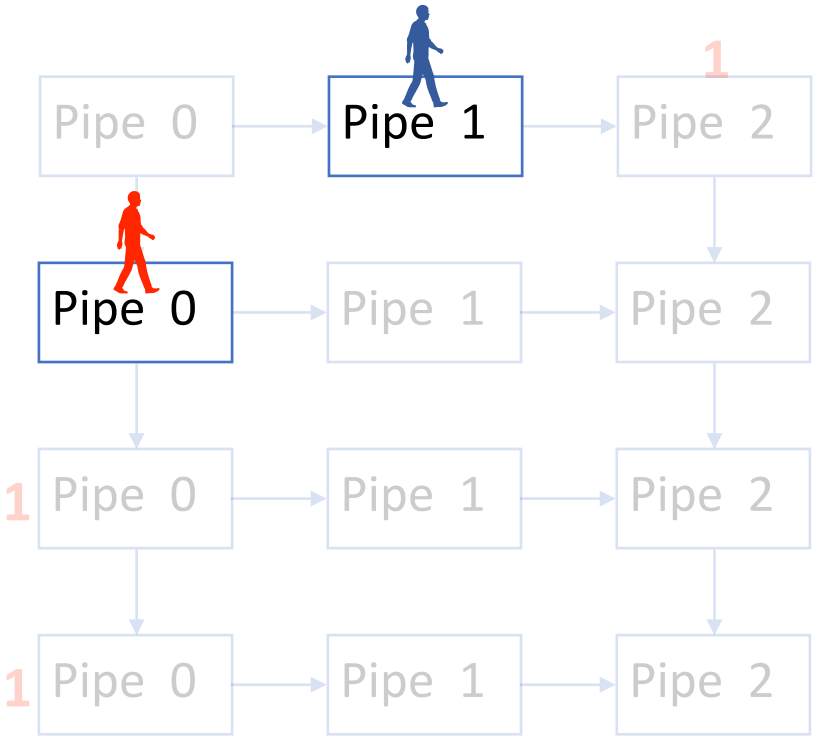
Scheduling Algorithm Walkthrough



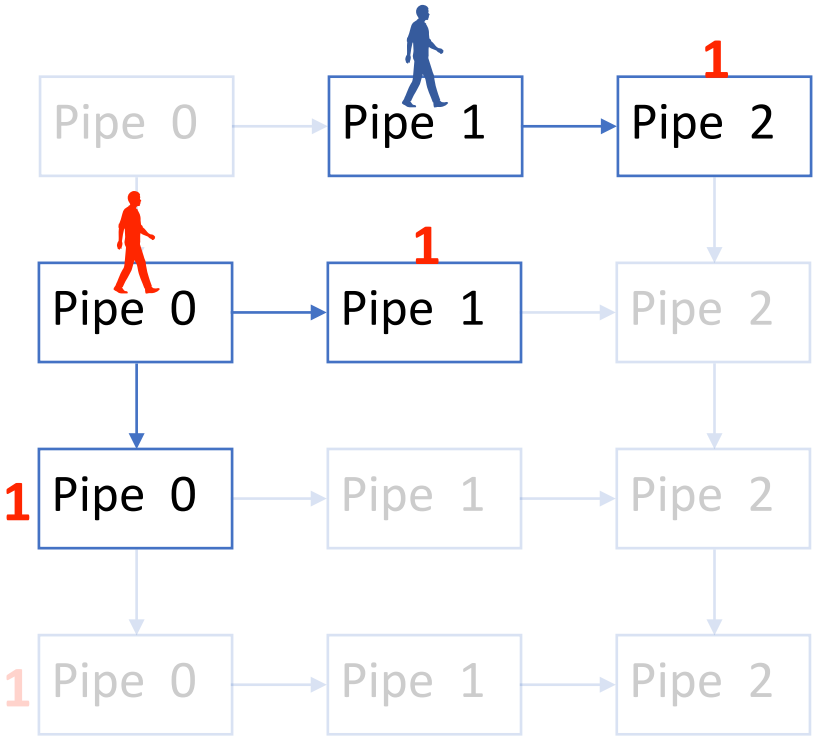
Scheduling Algorithm Walkthrough



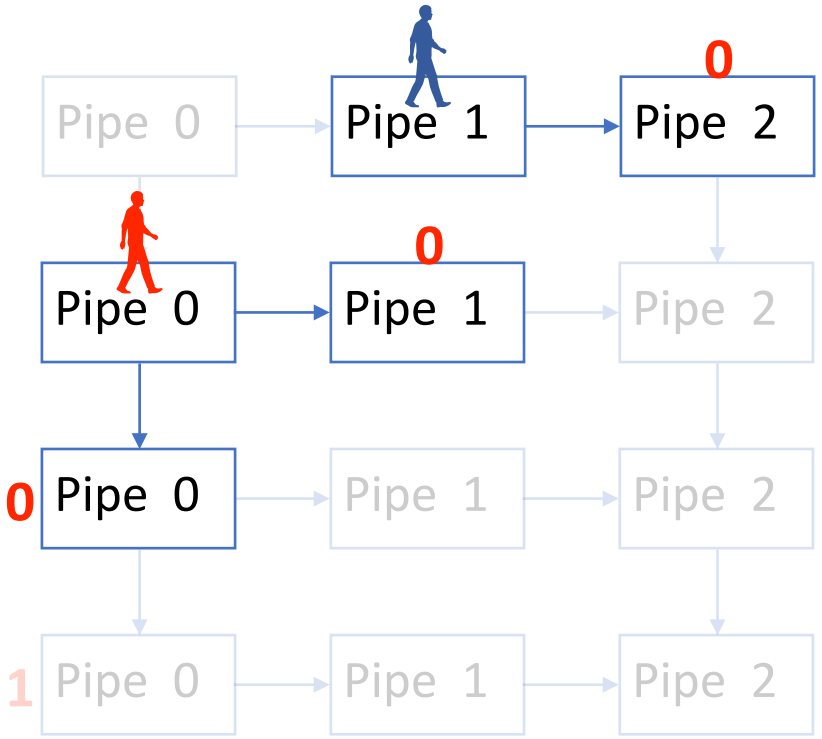
Scheduling Algorithm Walkthrough



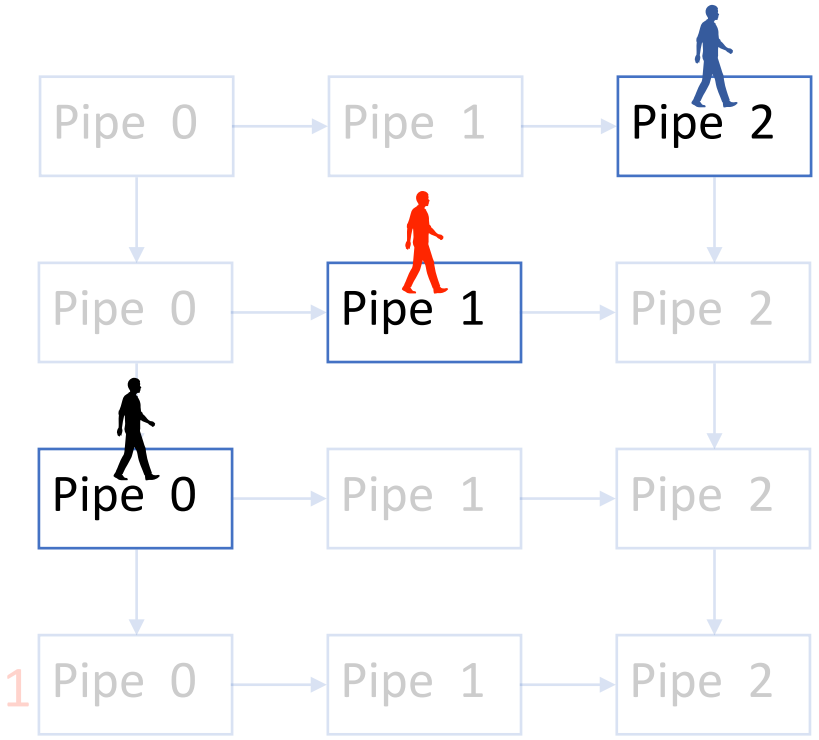
Scheduling Algorithm Walkthrough



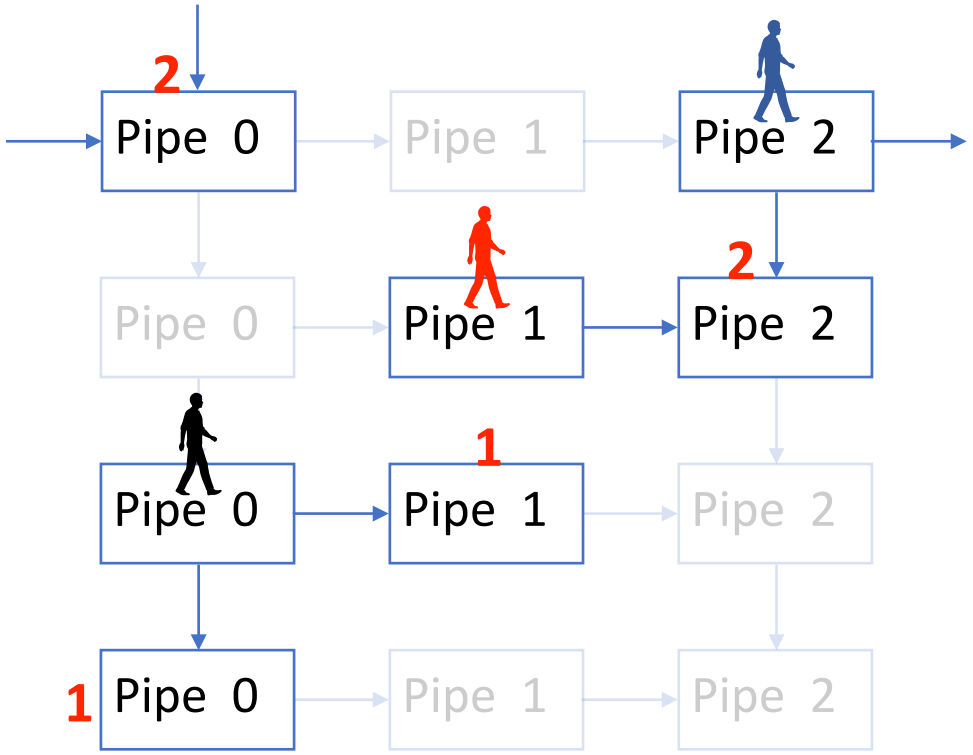
Scheduling Algorithm Walkthrough



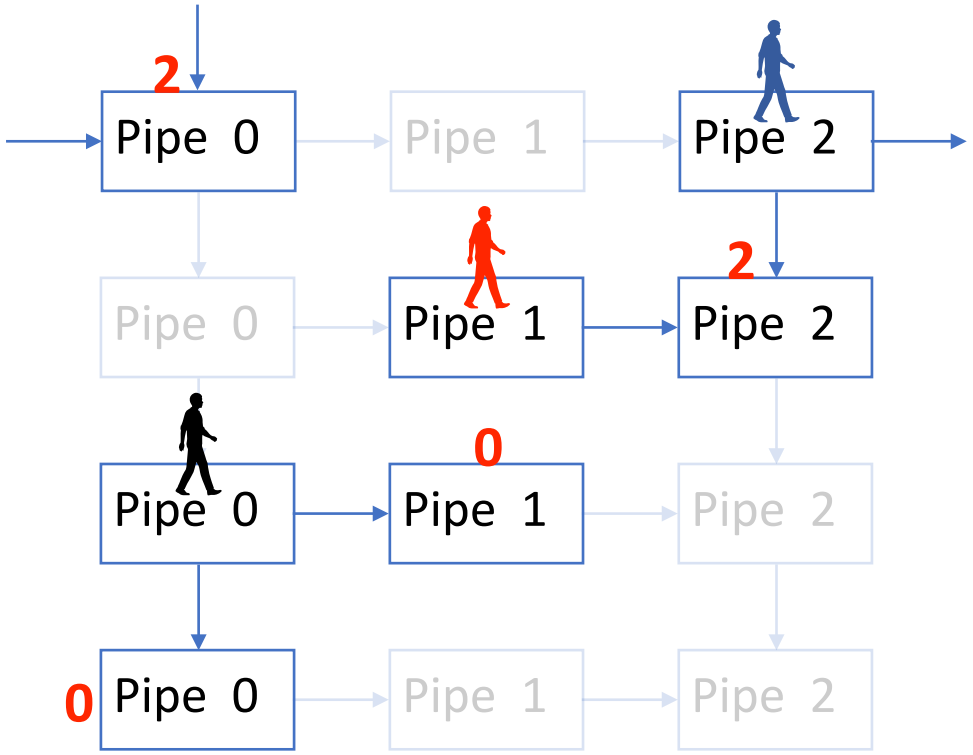
Scheduling Algorithm Walkthrough



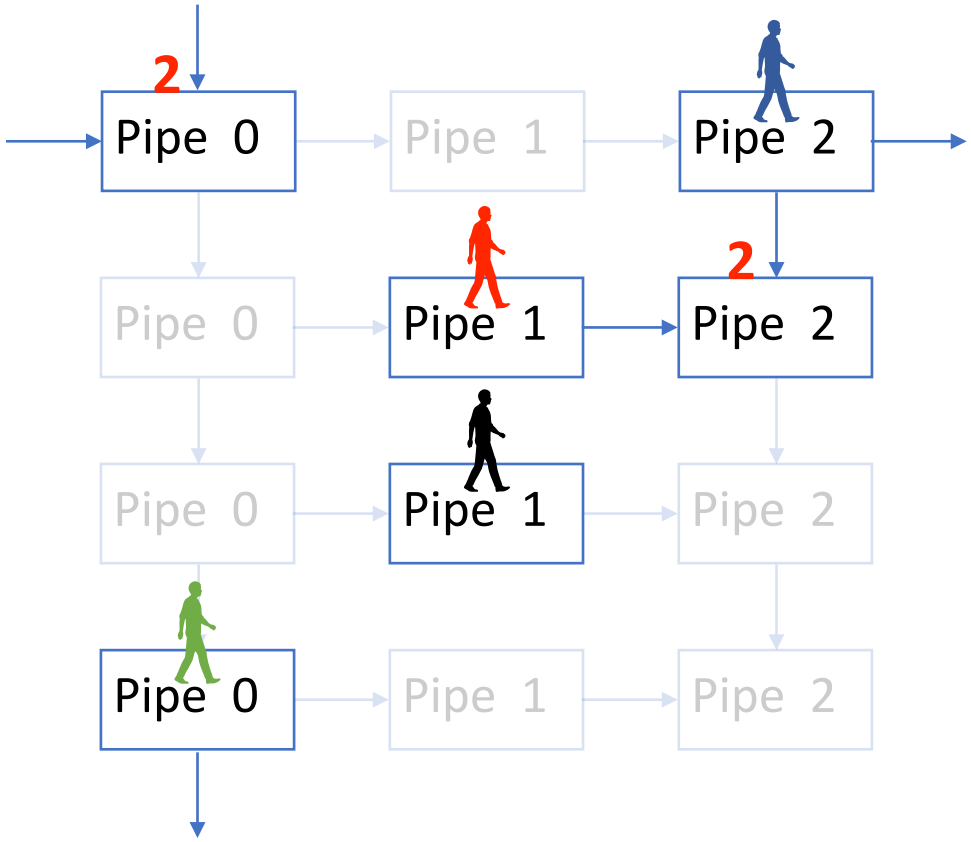
Scheduling Algorithm Walkthrough



Scheduling Algorithm Walkthrough

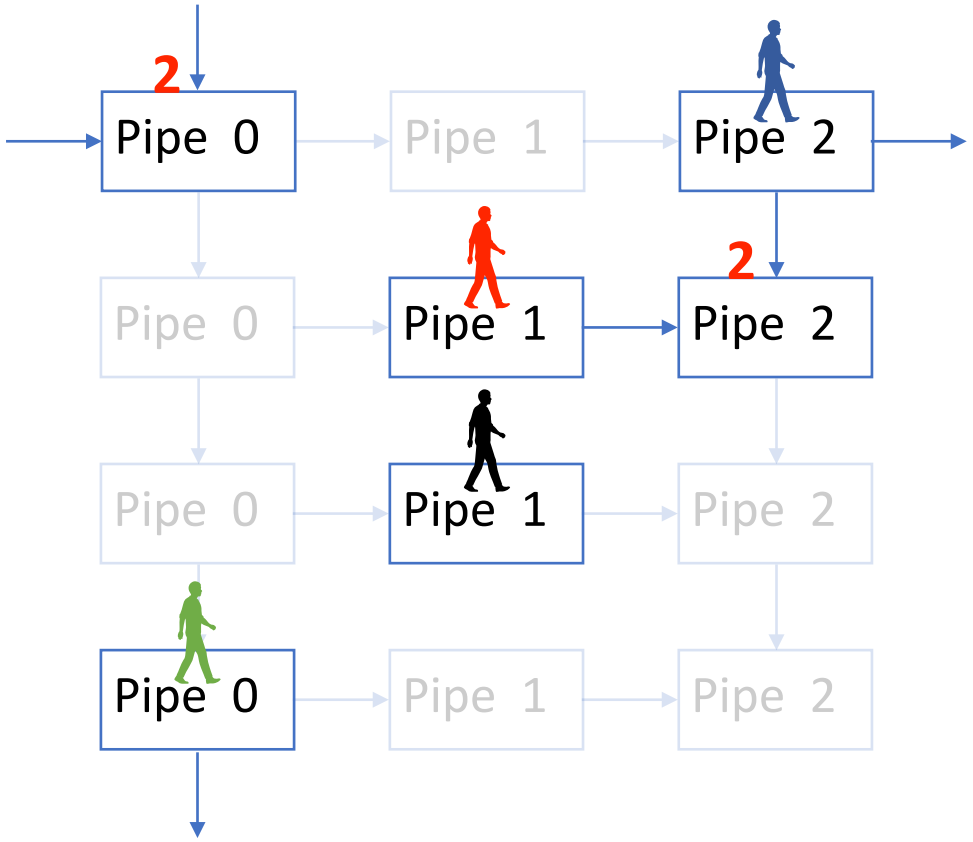


Scheduling Algorithm Walkthrough



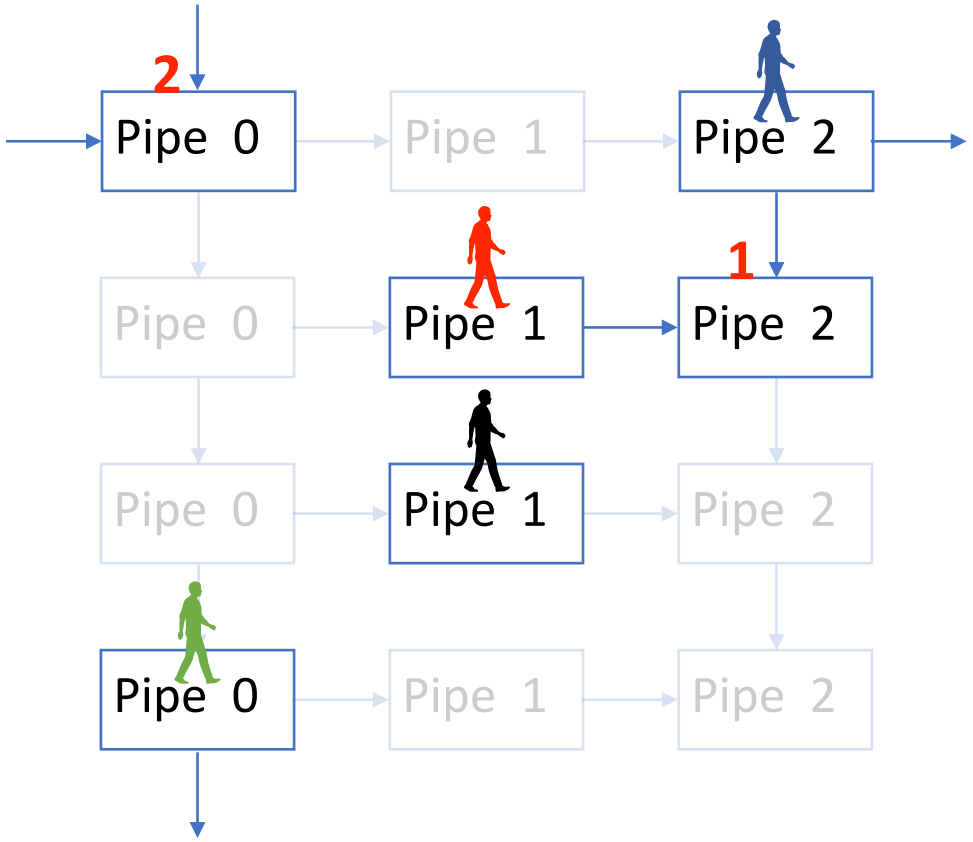
Scheduling Algorithm Walkthrough

- Case 1 : **Red** thread decrements the counter of right pipe first



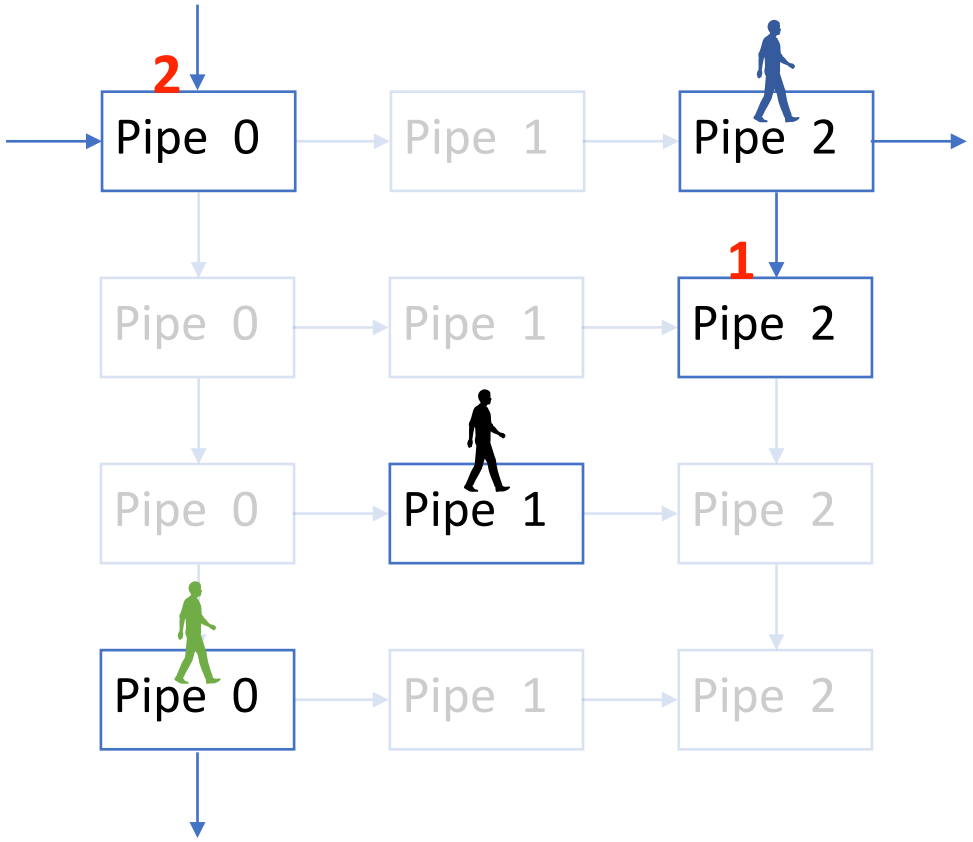
Scheduling Algorithm Walkthrough

- Case 1 : **Red** thread decrements the counter of right pipe first



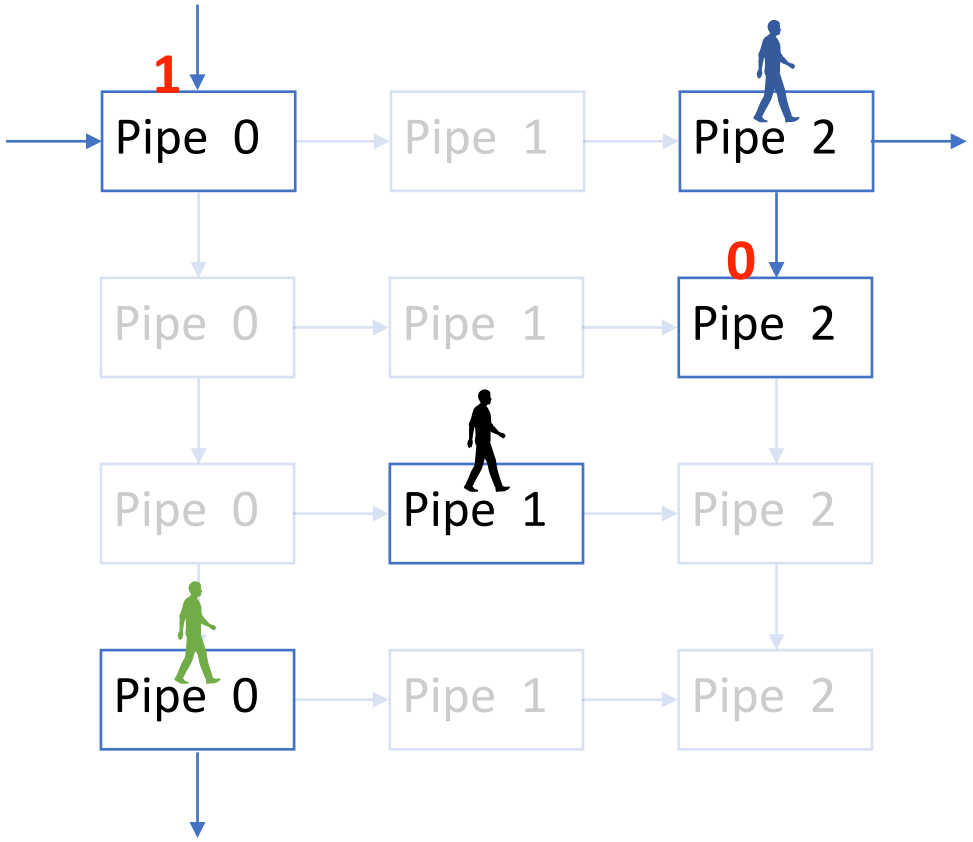
Scheduling Algorithm Walkthrough

- Case 1 : **Red** thread decrements the counter of right pipe first



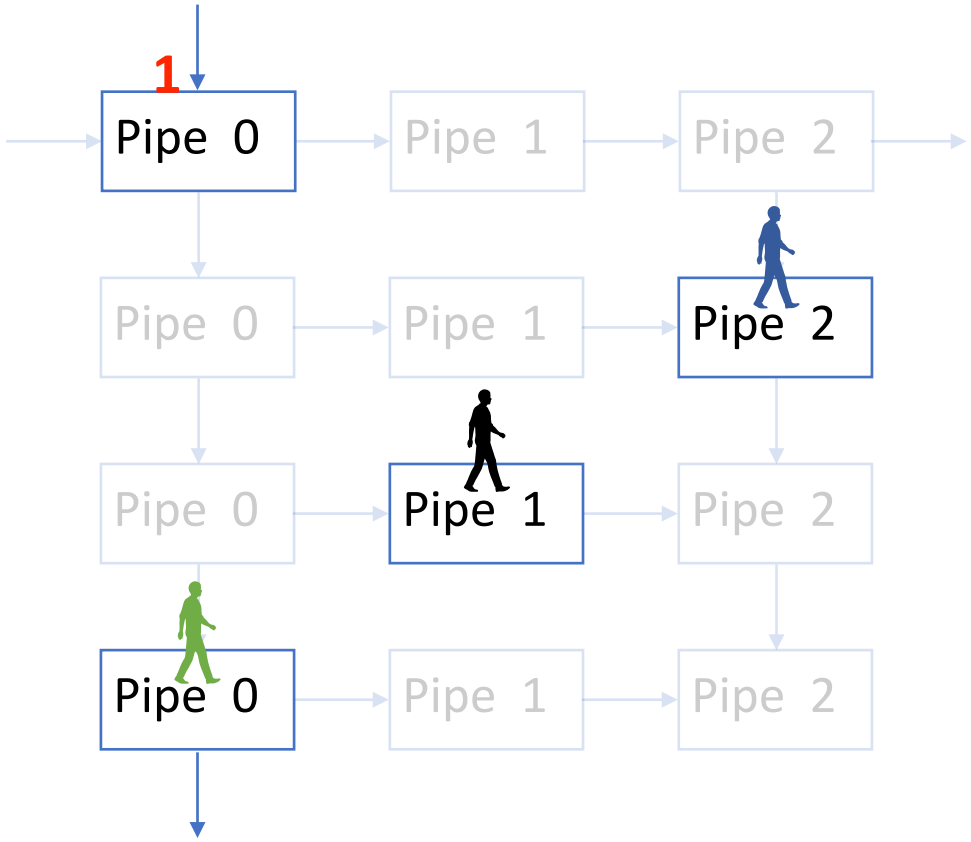
Scheduling Algorithm Walkthrough

- Case 1 : **Red** thread decrements the counter of right pipe first



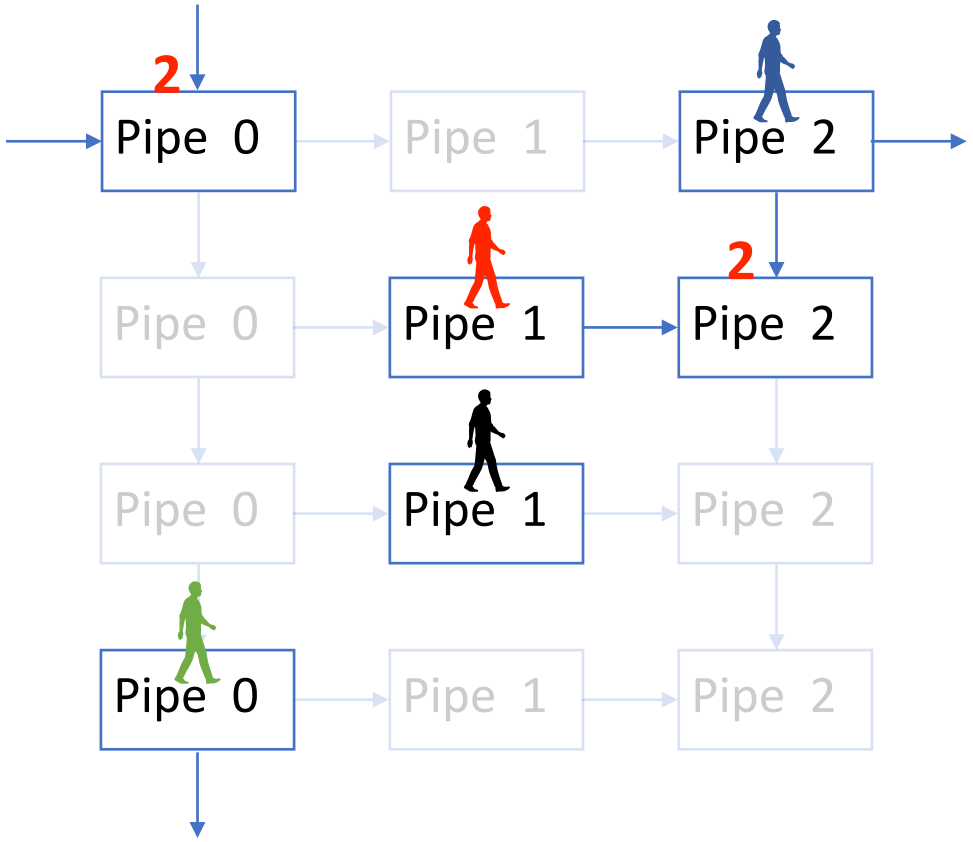
Scheduling Algorithm Walkthrough

- Case 1 : **Red** thread decrements the counter of right pipe first



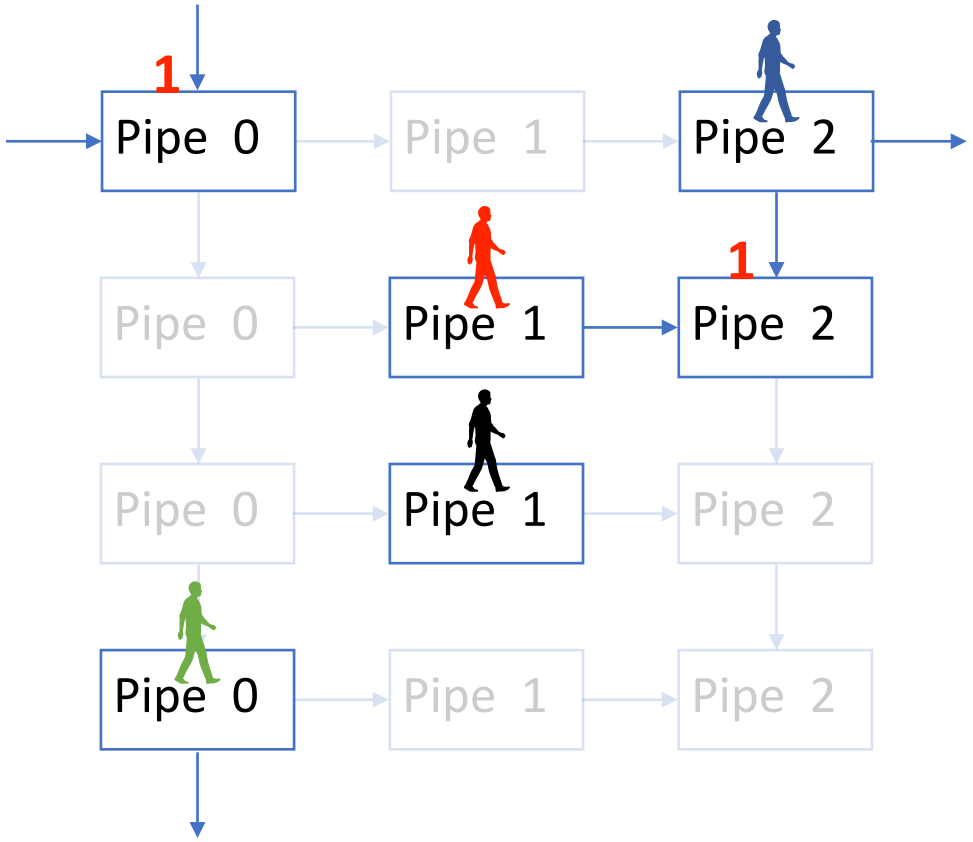
Scheduling Algorithm Walkthrough

- Case 2 : **Blue** thread decrements the counter of below pipe first



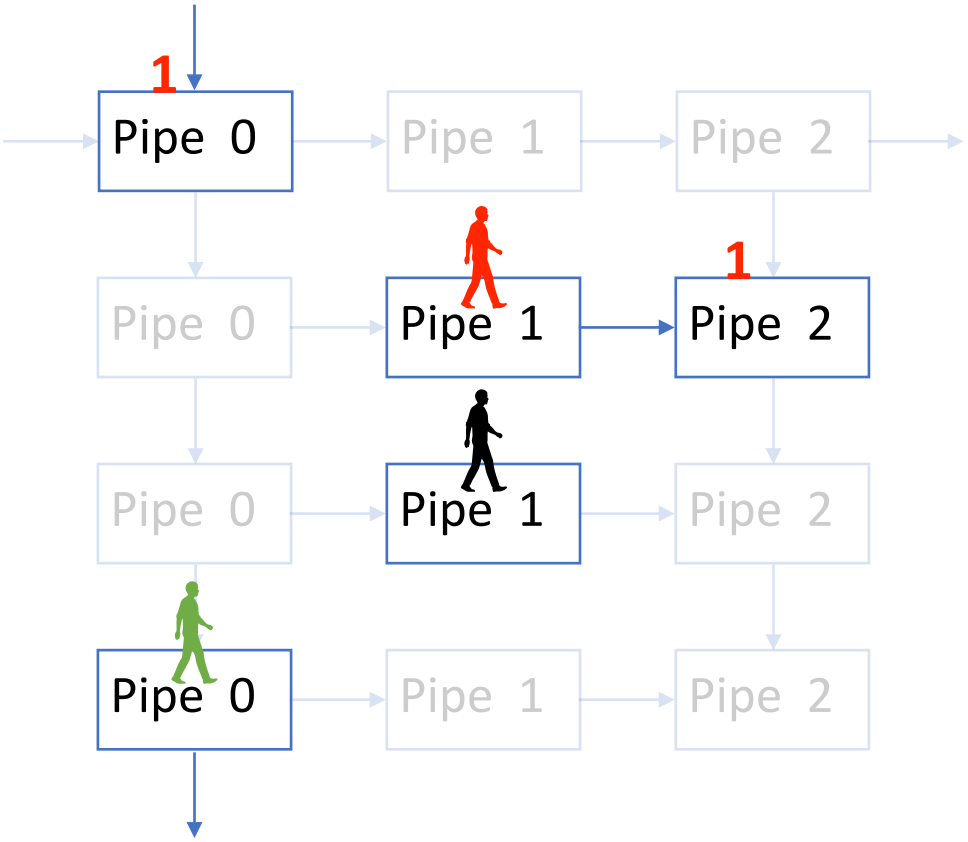
Scheduling Algorithm Walkthrough

- Case 2 : **Blue** thread decrements the counter of below pipe first



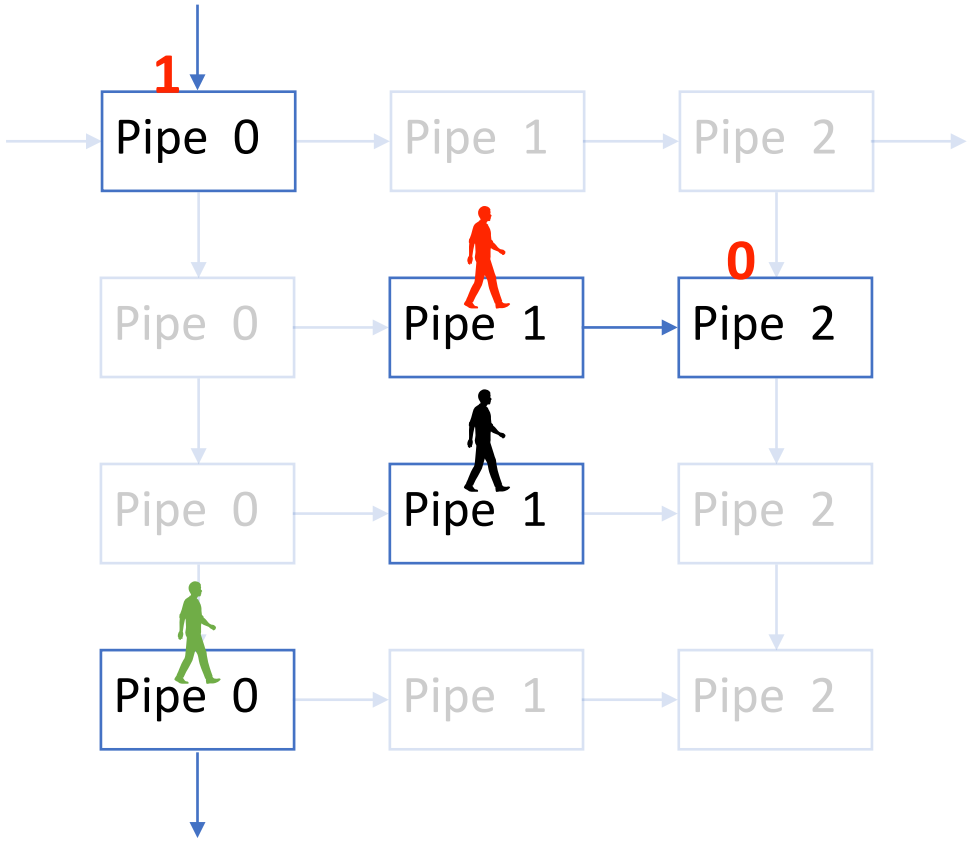
Scheduling Algorithm Walkthrough

- Case 2 : **Blue** thread decrements the counter of below pipe first



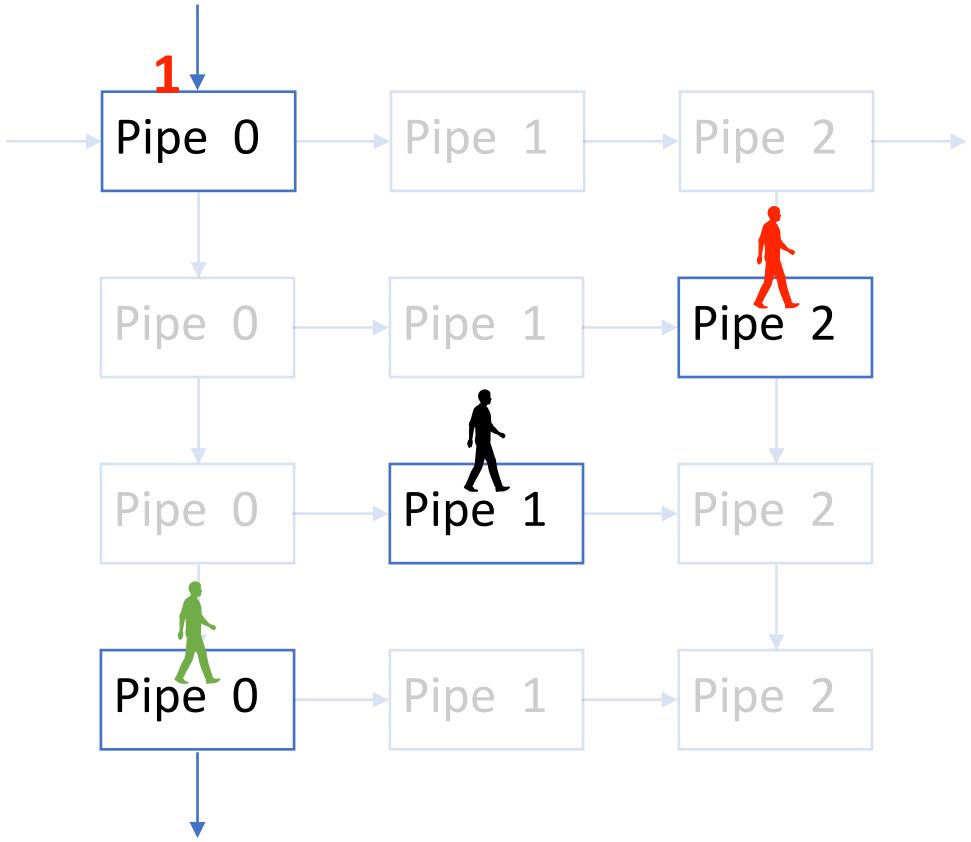
Scheduling Algorithm Walkthrough

- Case 2 : **Blue** thread decrements the counter of below pipe first



Scheduling Algorithm Walkthrough

- Case 2 : **Blue** thread decrements the counter of below pipe first



Conclusion

- We have presented the motivations behind Pipeflow
- We have presented a text processing application and compared the performance between Pipeflow, oneTBB, and a sequential version
- We have presented the Pipeflow scheduling algorithm
- We have integrated Pipeflow in Taskflow project (https://github.com/taskflow/taskflow)

References

- Text processing pipeline application
  - https://godbolt.org/z/KbTM3Kn5c
- More examples
  - https://taskflow.github.io/taskflow/TaskParallelPipeline.html

Thank You

https://cheng-hsiang-chiu.github.io