Ohua as STM alternative for shared state applications

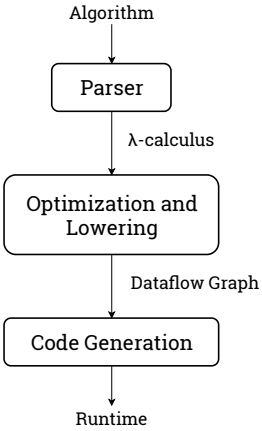
Master Midway Defense

Felix Wittwer

29. January 2020

Ohua[1]

Framework for implicit parallel programming:

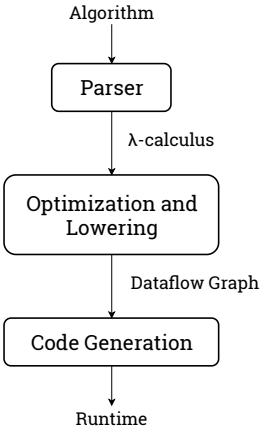


[1] Ertel et al. "Towards Implicit Parallel Programming for Systems." dissertation, 2019.

Ohua[1]

Framework for implicit parallel programming:

- Derives dataflow graph from algorithm file

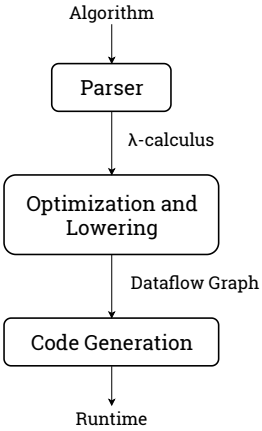


[1] Ertel et al. "Towards Implicit Parallel Programming for Systems." dissertation, 2019.

Ohua[1]

Framework for implicit parallel programming:

- Derives dataflow graph from algorithm file
- Runs optimizations on graph to exploit parallelism at compile time

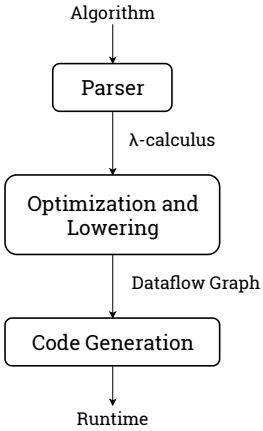


[1] Ertel et al. "Towards Implicit Parallel Programming for Systems." dissertation, 2019.

Ohua[1]

Framework for implicit parallel programming:

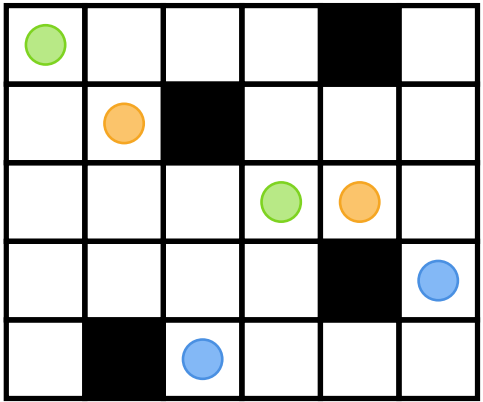
- Derives dataflow graph from algorithm file
- Runs optimizations on graph to exploit parallelism at compile time
- Generates native runtime code



[1] Ertel et al. "Towards Implicit Parallel Programming for Systems." dissertation, 2019.

Throwback: Labyrinth Benchmark

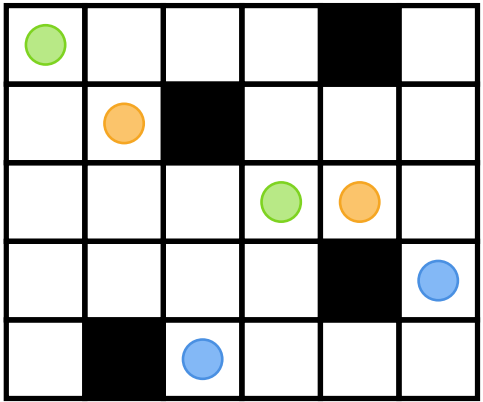
Given: 3D maze, pairs of points



Throwback: Labyrinth Benchmark

Given: 3D maze, pairs of points

Goal: Map a path between each pair of points



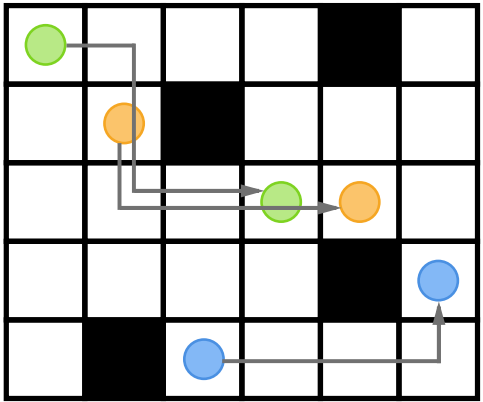
Throwback: Labyrinth Benchmark

Given: 3D maze, pairs of points

Goal: Map a path between each pair of points

Implementation:

- parallel search for new paths



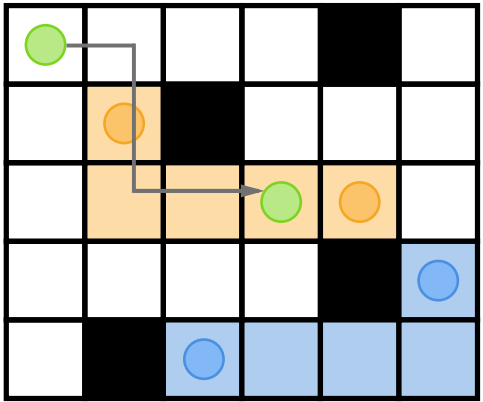
Throwback: Labyrinth Benchmark

Given: 3D maze, pairs of points

Goal: Map a path between each pair of points

Implementation:

- parallel search for new paths
- merge paths into the maze



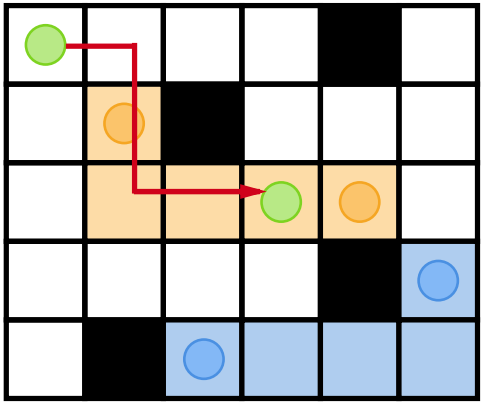
Throwback: Labyrinth Benchmark

Given: 3D maze, pairs of points

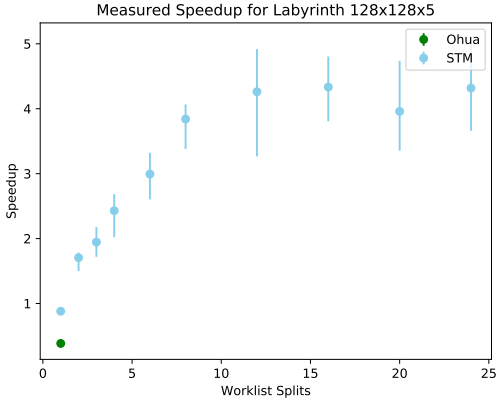
Goal: Map a path between each pair of points

Implementation:

- parallel search for new paths
- merge paths into the maze
  → retry if path crosses other paths

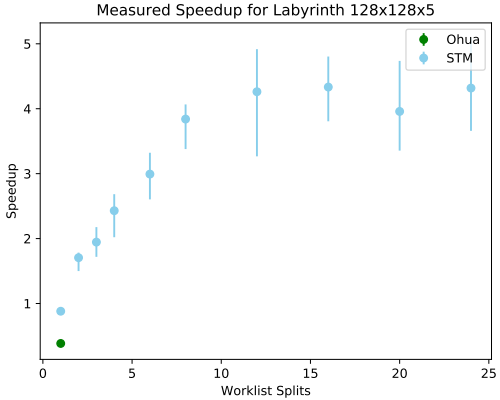


Throwback: Early Results



- Ohua: one measuring point
- Speedup < 0.5

Throwback: Early Results



- Ohua: one measuring point
- Speedup < 0.5

- parallelizable problem that would be solved with locks
- Ohua's aim: offer lock-free alternative

Improving the Benchmark Performance

Previousy envisioned steps:

1. add a simple scheduler to the runtime to improve measurability

Improving the Benchmark Performance

Previousy envisioned steps:

1. ~~add a simple scheduler to the runtime to improve measurability~~

Improving the Benchmark Performance

Previousy envisioned steps:

1. ~~add a simple scheduler to the runtime to improve measurability~~
2. implement data parallelism in `smap` operator

Improving the Benchmark Performance

Previousy envisioned steps:

1. ~~add a simple scheduler to the runtime to improve measurability~~
2. implement data parallelism ~~in `smap` operator~~ manually

Improving the Benchmark Performance

Previousy envisioned steps:

1. ~~add a simple scheduler to the runtime to improve measurability~~
2. implement data parallelism ~~in `smap` operator~~ manually
3. reduce the number of retries by moving the state updates
   - requires thinking about how state sharing should work

Improving the Benchmark Performance

Previousy envisioned steps:

1. ~~add a simple scheduler to the runtime to improve measurability~~
2. implement data parallelism ~~in `smap` operator~~ manually
3. reduce the number of retries by moving the state updates
   - ~~requires thinking about how state sharing should work~~

Improving the Benchmark Performance

Previousy envisioned steps:

1. ~~add a simple scheduler to the runtime to improve measurability~~
2. implement data parallelism ~~in `smap` operator~~ manually
3. reduce the number of retries by moving the state updates
   - ~~requires thinking about how state sharing should work~~

Goal: Finding generalized dataflow transformations

Improving the Benchmark Performance

Previousy envisioned steps:

1. ~~add a simple scheduler to the runtime to improve measurability~~
2. implement data parallelism ~~in `smap` operator~~ manually
3. reduce the number of retries by moving the state updates
   - ~~requires thinking about how state sharing should work~~

Goal: Finding generalized dataflow transformations
→ develop proof of concept implementations

Transformation 1: Worklist Splits

```
fn fill(maze: Maze, to_map: Vec<(Point, Point)>) -> Maze {

    let paths = for pair in to_map {
        find_path(maze, pair)
    };

    let (remap_paths, new_maze) = update_maze(maze, paths);

    // recursively call `fill` as necessary
}
```

Transformation 1: Worklist Splits

```
fn fill(maze: Maze, to_map: Vec<(Point, Point)>) -> Maze {

    let paths = for pair in to_map {
        find_path(maze, pair)
    };

    let (remap_paths, new_maze) = update_maze(maze, paths);

    // recursively call `fill` as necessary
}
```

- Split worklist into *m* equally large parts and run path-finding data-parallel

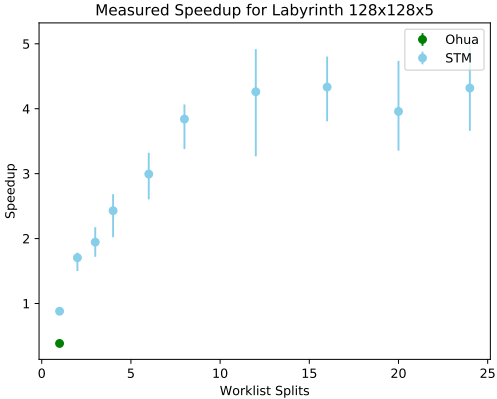
Transformation 1: Worklist Splits

```
fn fill(maze: Maze, to_map: Vec<(Point, Point)>) -> Maze {


    let paths = for pair in to_map {
        find_path(maze, pair)
    };









    let (remap_paths, new_maze) = update_maze(maze, paths);


    // recursively call `fill` as necessary
}
```

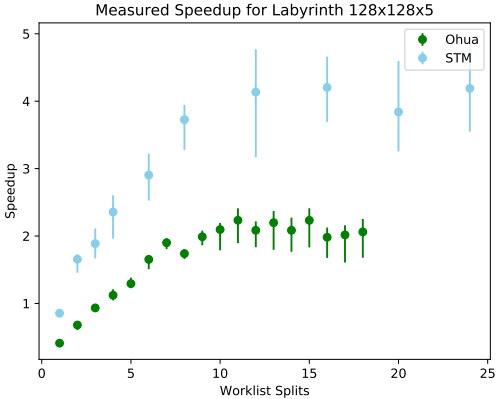
```
fn fill(maze: Maze, to_map: Vec<(Point, Point)>) -> Maze {

    let (tm0, tm1) = splitup(to_map);
    let part0 = for pair in tm0 {
        find_path(maze, pair)
    };
    let part1 = for pair in tm1 {
        find_path(maze, pair)
    };

    let paths = join(part0, part1);

    let (remap_paths, new_maze) = update_maze(maze, paths);


    // recursively call `fill` as necessary
}
```

- Split worklist into *m* equally large parts and run path-finding data-parallel

Transformation 1: Worklist Splits

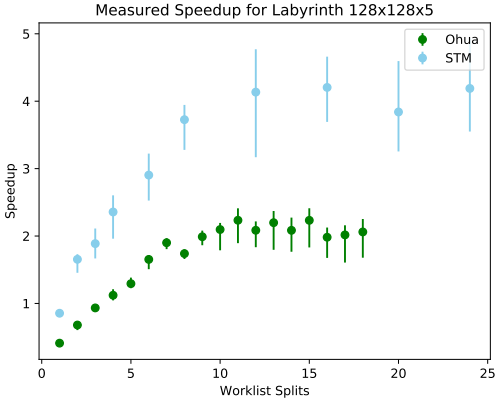


Transformation 1: Worklist Splits



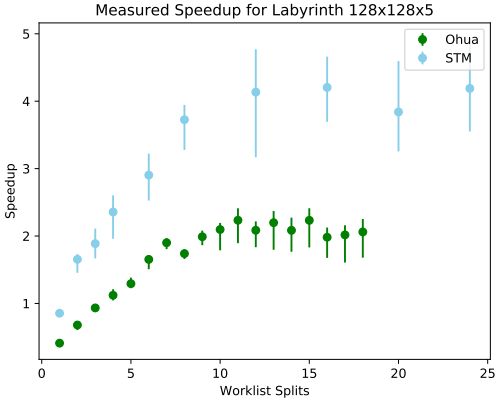
Transformation 1: Worklist Splits

- Granular parallelism control



Transformation 1: Worklist Splits

- Granular parallelism control
- Still only about half the speedup `stm` shows

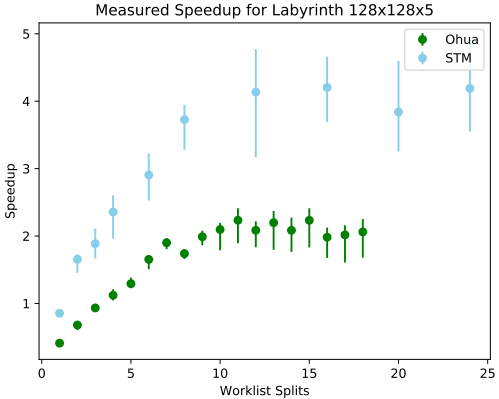


Transformation 1: Worklist Splits

- Granular parallelism control
- Still only about half the speedup `stm` shows

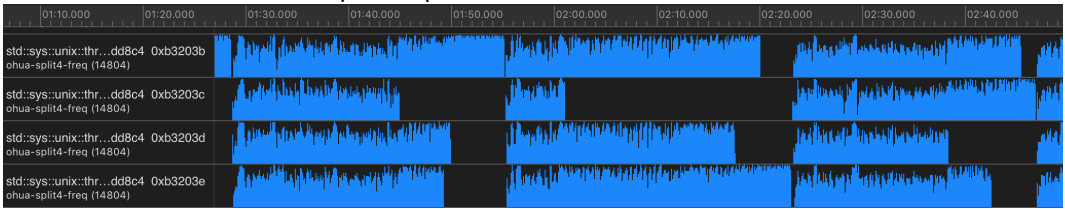
Reason: Straggler Problem

- Synchronization points force all threads to wait



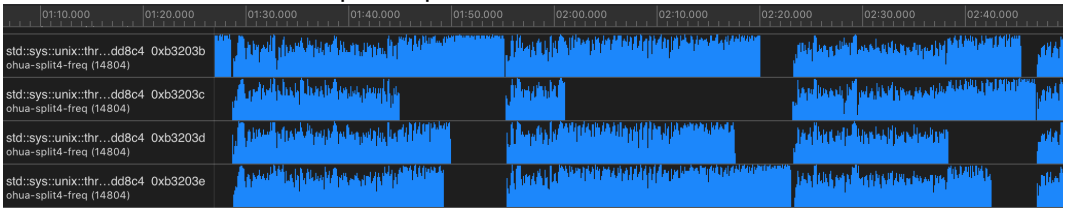
Straggler Problem with Splitting

Instruments trace for 4-split implementation:



Straggler Problem with Splitting

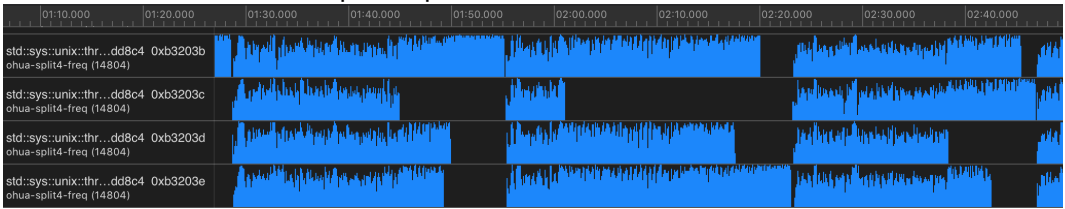
Instruments trace for 4-split implementation:



- Worklists don’t take equally long to process

Straggler Problem with Splitting

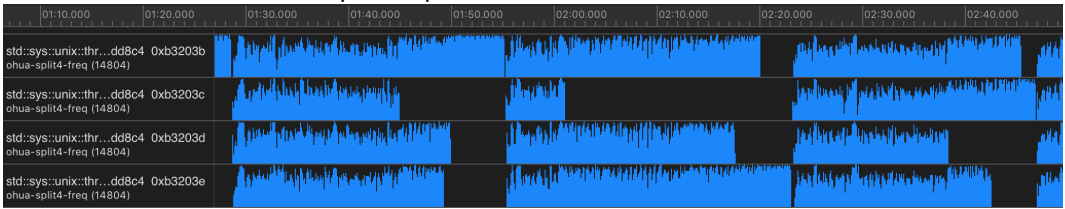
Instruments trace for 4-split implementation:



- Worklists don’t take equally long to process
- All threads are forced to wait at barrier

Straggler Problem with Splitting

Instruments trace for 4-split implementation:



- Worklists don’t take equally long to process
- All threads are forced to wait at barrier
  - each recomputation affects execution time twice
  - uses more processing time and causes slack time for other cores

Transformation 2: Update Frequency

```rust
fn fill(maze: Maze, to_map: Vec<(Point, Point)>) -> Maze {

    let (tm0, tm1) = splitup(to_map);
    let part0 = for pair in tm0 {
        find_path(maze, pair)
    };
    let part1 = for pair in tm1 {
        find_path(maze, pair)
    };

    let paths = join(part0, part1);

    let (remap_paths, new_maze) = update_maze(maze, paths);


    // recursively call `fill` as necessary
}
```

Transformation 2: Update Frequency

```
fn fill(maze: Maze, to_map: Vec<(Point, Point)>) -> Maze {

    let (tm0, tm1) = splitup(to_map);
    let part0 = for pair in tm0 {
        find_path(maze, pair)
    };
    let part1 = for pair in tm1 {
        find_path(maze, pair)
    };

    let paths = join(part0, part1);

    let (remap_paths, new_maze) = update_maze(maze, paths);


    // recursively call `fill` as necessary
}
```

- Update shared state more often by only processing *n* elements at once

Transformation 2: Update Frequency

```
fn fill(maze: Maze, to_map: Vec<(Point, Point)>) -> Maze {

    let (tm0, tm1) = splitup(to_map);
    let part0 = for pair in tm0 {
        find_path(maze, pair)
    };
    let part1 = for pair in tm1 {
        find_path(maze, pair)
    };

    let paths = join(part0, part1);

    let (remap_paths, new_maze) = update_maze(maze, paths);


    // recursively call `fill` as necessary
}
```

```
fn fill(maze: Maze, to_map: Vec<(Point, Point)>) -> Maze {
    let (points, still_to_map) = take_n(to_map, frequency);
    let (tm0, tm1) = splitup(points);
    let part0 = for pair in tm0 {
        find_path(maze, pair)
    };
    let part1 = for pair in tm1 {
        find_path(maze, pair)
    };

    let paths = join(part0, part1);

    let (remap_paths, new_maze) = update_maze(maze, paths);
    let to_remap = join(remap_paths, still_to_map);

    // recursively call `fill` as necessary
}
```

- Update shared state more often by only processing *n* elements at once

Transformation 3: Improve Resource Utilization

- further improve performance by utilizing work-stealing algorithms
- execute parallel computations on top of `tokio` framework

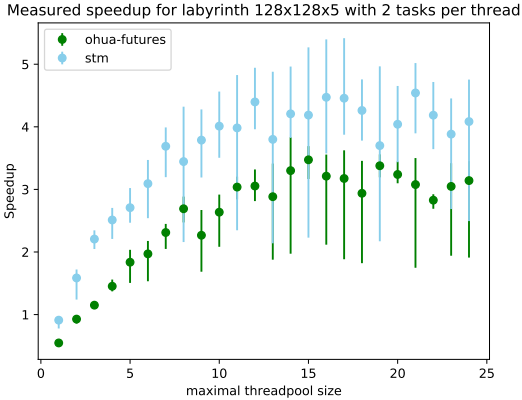
Transformation 3: Improve Resource Utilization

- further improve performance by utilizing work-stealing algorithms
- execute parallel computations on top of `tokio` framework
  - schedule computations as tasks on threadpool

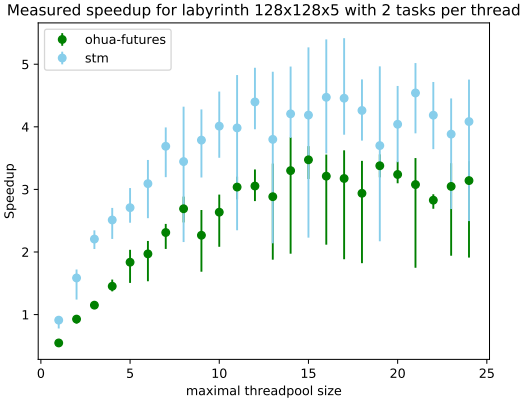
Transformation 3: Improve Resource Utilization

- further improve performance by utilizing work-stealing algorithms
- execute parallel computations on top of `tokio` framework
  - schedule computations as tasks on threadpool
  - when thread is done it can steal work from other threads

Result: Significant Performance Improvements

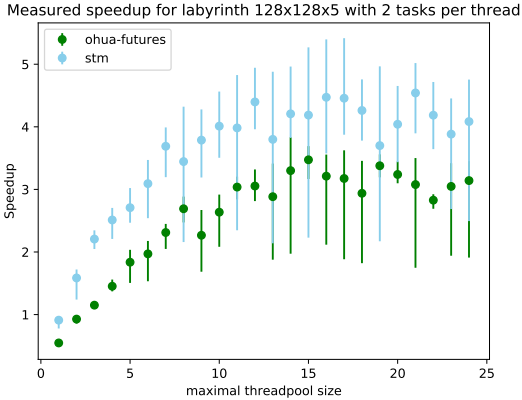


Result: Significant Performance Improvements



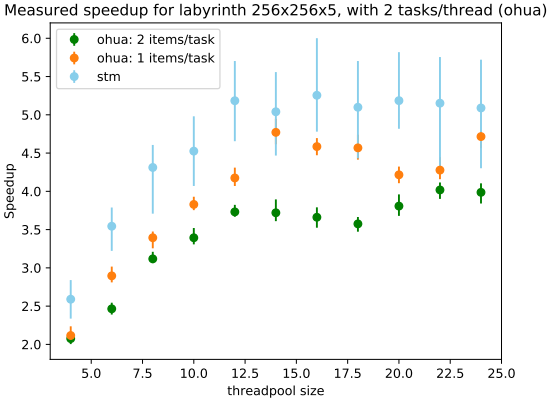
- Proof of concept: Ohua almost on par with `stm`

Result: Significant Performance Improvements



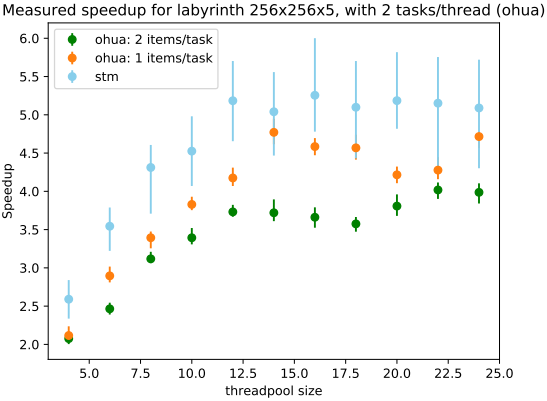
- Proof of concept: Ohua almost on par with `stm`
- Advantage: deterministic execution model
  - allows better debugging

Result: Significant Performance Improvements



- Proof of concept: Ohua almost on par with `stm`
- Advantage: deterministic execution model
  - allows better debugging

Result: Significant Performance Improvements



- Proof of concept: Ohua almost on par with `stm`
- Advantage: deterministic execution model
  - allows better debugging
  - also reflected by execution times

Transformations: Overview

General-purpose transformations applied:

Transformations: Overview

General-purpose transformations applied:

1. Data parallelism for stateless loop computations
   - improve execution times

Transformations: Overview

General-purpose transformations applied:

1. Data parallelism for stateless loop computations
   - improve execution times
2. Batch write accesses to shared state
   - state updates take effect earlier

Transformations: Overview

General-purpose transformations applied:

1. Data parallelism for stateless loop computations
   - improve execution times
2. Batch write accesses to shared state
   - state updates take effect earlier
3. Use work-stealing algorithm
   - lessens effect of the straggler problem

Other benchmarks

- According to Minh et al.[2] applications for `stm` are manifold

[2] Minh, Chi Cao, et al. "STAMP: Stanford transactional applications for multi-processing." 2008 IEEE International Symposium on Workload Characterization. IEEE, 2008.

Other benchmarks

- According to Minh et al.[2] applications for `stm` are manifold
- Each has own behavioral patterns
  $\rightarrow$ test transformations on representative application range

[2] Minh, Chi Cao, et al. "STAMP: Stanford transactional applications for multi-processing." 2008 IEEE International Symposium on Workload Characterization. IEEE, 2008.

Benchmark 2: Intruder

- Simulates intrusion detection system

Benchmark 2: Intruder

- Simulates intrusion detection system
  - reassembles & inspects network packets

Benchmark 2: Intruder

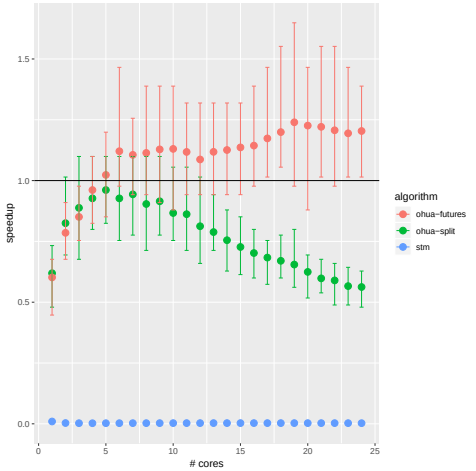
- Simulates intrusion detection system
  - reassembles & inspects network packets
  - hard to parallelize

Benchmark 2: Intruder

- Simulates intrusion detection system
  - reassembles & inspects network packets
  - hard to parallelize
  - designed to show bad performance of STM

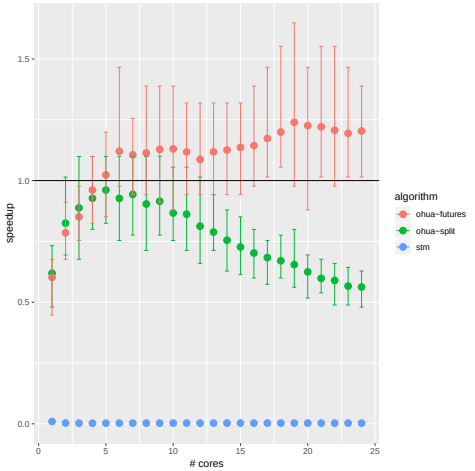
Benchmark 2: Intruder

- Simulates intrusion detection system
  - reassembles & inspects network packets
  - hard to parallelize
  - designed to show bad performance of STM



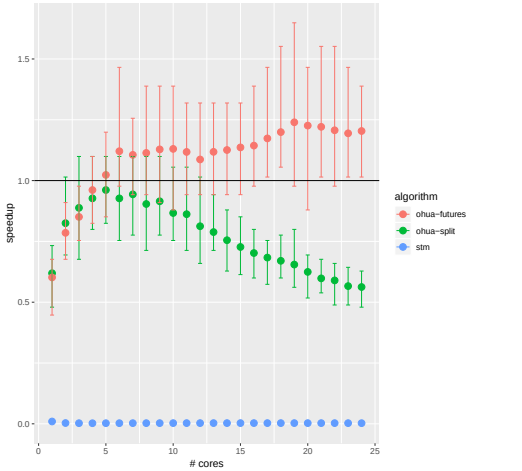
Benchmark 2: Intruder

- Simulates intrusion detection system
  - reassembles & inspects network packets
  - hard to parallelize
  - designed to show bad performance of STM
- Ohua: Speedups of up to 1.25



Benchmark 2: Intruder

- Simulates intrusion detection system
  - reassembles & inspects network packets
  - hard to parallelize
  - designed to show bad performance of STM
- Ohua: Speedups of up to 1.25
- No contention due to its execution model



Next Steps

1. Port 2-3 more benchmarks
   - verify that found optimizations are applicable to other problem classes normally solved wirh STM

Next Steps

1. Port 2-3 more benchmarks
   - verify that found optimizations are applicable to other problem classes normally solved wirh STM
2. Implement compiler optimizations
   - add dataflow transformations to the Ohua compiler

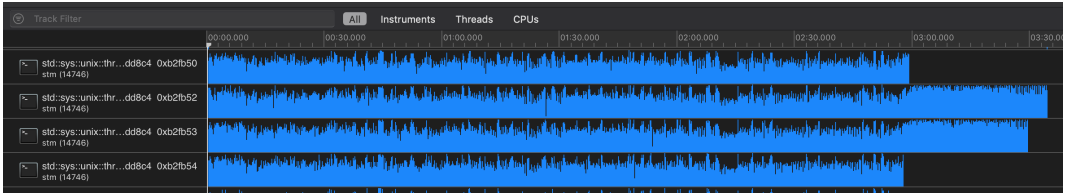
Next Steps

1. Port 2-3 more benchmarks
   - verify that found optimizations are applicable to other problem classes normally solved wirh STM
2. Implement compiler optimizations
   - add dataflow transformations to the Ohua compiler
3. Verify results
   - using previously examined benchmarks

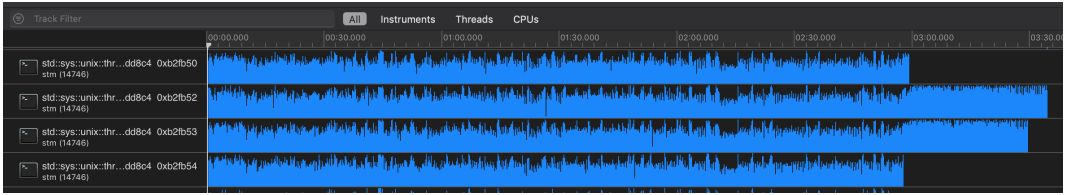
Thank you for your attention.

Backup

Backup: Straggler Problem in STM

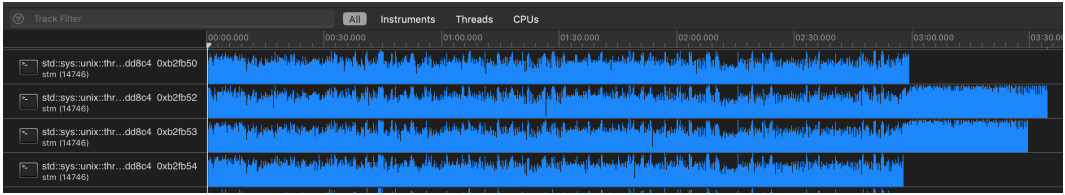


Backup: Straggler Problem in STM



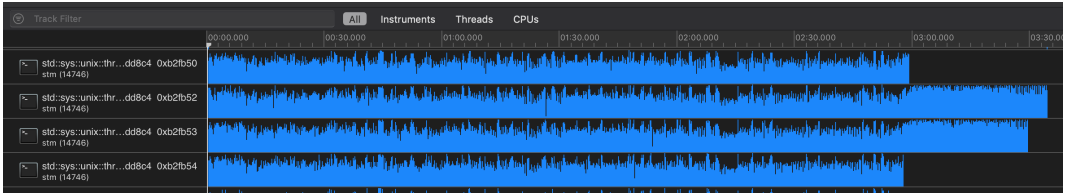
- STM interleaves reads and writes on shared data

Backup: Straggler Problem in STM



- STM interleaves reads and writes on shared data
- trade-off: no synchronization barriers

Backup: Straggler Problem in STM



- STM interleaves reads and writes on shared data
- trade-off: no synchronization barriers but
  - non-deterministic execution model
  - more recomputations due to state invalidation

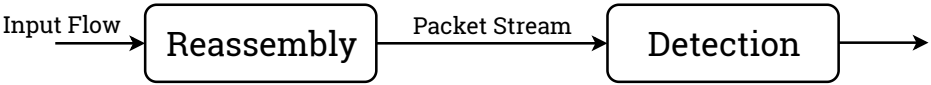
Backup: Benchmark Classification

Classification of STAMP[2] suite benchmarks:

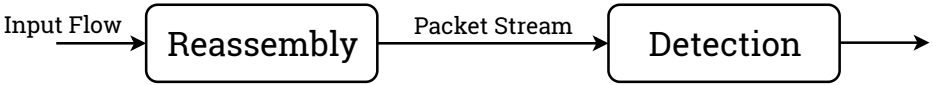
application	tx length	r/w set	tx time	contention
labyrinth	long	large	high	high
bayes	long	large	high	high
yada	long	large	high	medium
vacation	medium	medium	high	low/medium
genome	medium	medium	high	low
intruder	short	medium	medium	high
kmeans	short	small	low	low
ssca2	short	small	low	low

[2] Minh, Chi Cao, et al. "STAMP: Stanford transactional applications for multi-processing." 2008 IEEE International Symposium on Workload Characterization. IEEE, 2008.

Backup: Intruder Benchmark

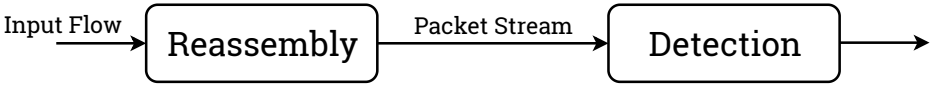


Backup: Intruder Benchmark



- Reconstructed packets stored in shared hash map

Backup: Intruder Benchmark



- Reconstructed packets stored in shared hash map
- Parallel write accesses in STM provoke recomputations