Ohua as STM Alternative for Shared State Applications

Master Defense

Felix Wittwer

25th of August, 2020

Motivation

Parallelization approaches for shared state applications:

Motivation

Parallelization approaches for shared state applications:

- Compiler analyses

Motivation

Parallelization approaches for shared state applications:

- Compiler analyses
  - struggle to uncover parallelism (no alias analysis etc)

Motivation

Parallelization approaches for shared state applications:

- Compiler analyses
  - struggle to uncover parallelism (no alias analysis etc)
- Locking

Motivation

Parallelization approaches for shared state applications:

- Compiler analyses
  - struggle to uncover parallelism (no alias analysis etc)
- Locking
  - coarse grain locking doesn't scale

Motivation

Parallelization approaches for shared state applications:

- Compiler analyses
  - struggle to uncover parallelism (no alias analysis etc)
- Locking
  - coarse grain locking doesn't scale
  - fine grained locking is error-prone

Motivation

Parallelization approaches for shared state applications:

- Compiler analyses
  - struggle to uncover parallelism (no alias analysis etc)
- Locking
  - coarse grain locking doesn't scale
  - fine grained locking is error-prone
  - resulting program is non-deterministic

Motivation

Parallelization approaches for shared state applications:

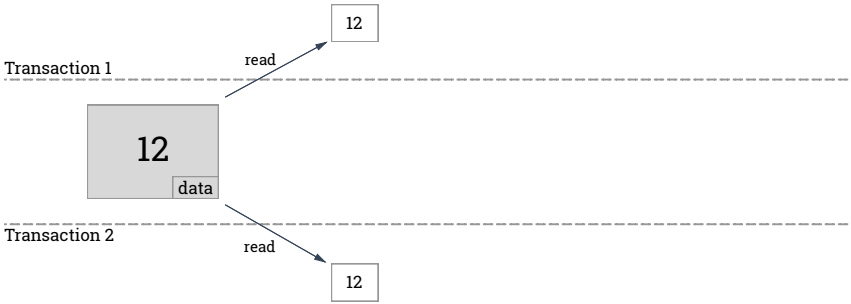
- Compiler analyses
  - struggle to uncover parallelism (no alias analysis etc)
- Locking
  - coarse grain locking doesn't scale
  - fine grained locking is error-prone
  - resulting program is non-deterministic
- Software Transactional Memory

Software Transactional Memory[1]

- uses *transactions* to guard access to shared data

[1] Shavit, et al. "Software transactional memory." Distributed Computing 10.2 (1997): 99-116.

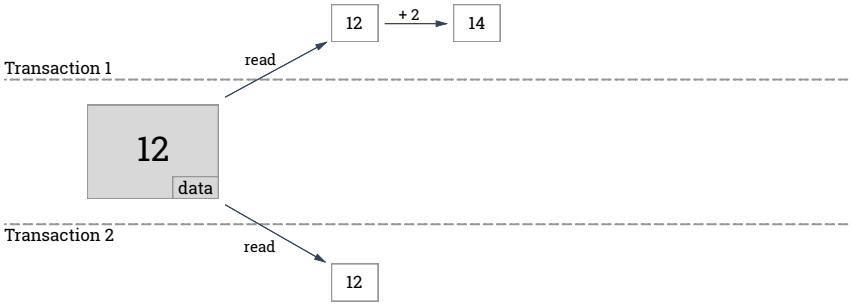
Software Transactional Memory[1]



- uses *transactions* to guard access to shared data

[1] Shavit, et al. "Software transactional memory." Distributed Computing 10.2 (1997): 99-116.

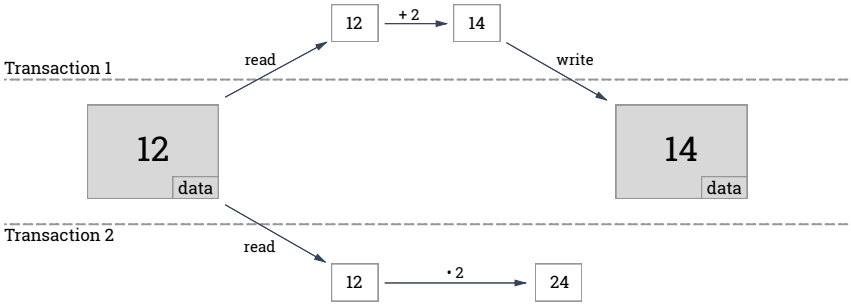
Software Transactional Memory[1]



- uses *transactions* to guard access to shared data

[1] Shavit, et al. "Software transactional memory." Distributed Computing 10.2 (1997): 99-116.

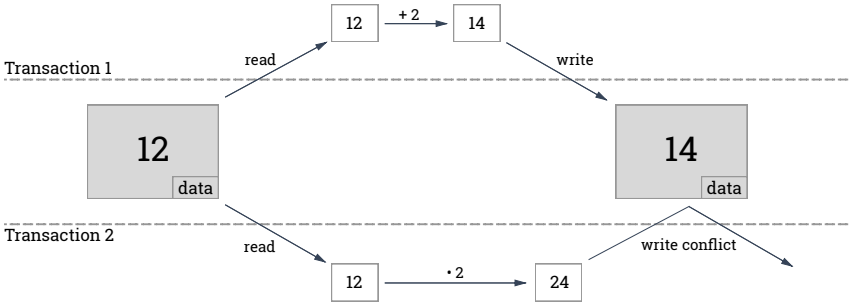
Software Transactional Memory[1]



- uses *transactions* to guard access to shared data

[1] Shavit, et al. "Software transactional memory." Distributed Computing 10.2 (1997): 99-116.

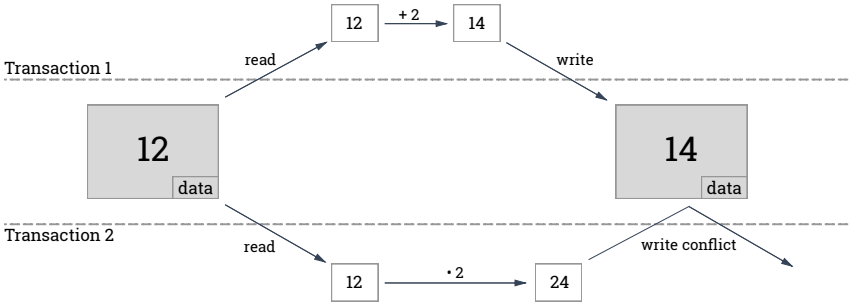
Software Transactional Memory[1]



- uses *transactions* to guard access to shared data

[1] Shavit, et al. "Software transactional memory." Distributed Computing 10.2 (1997): 99-116.

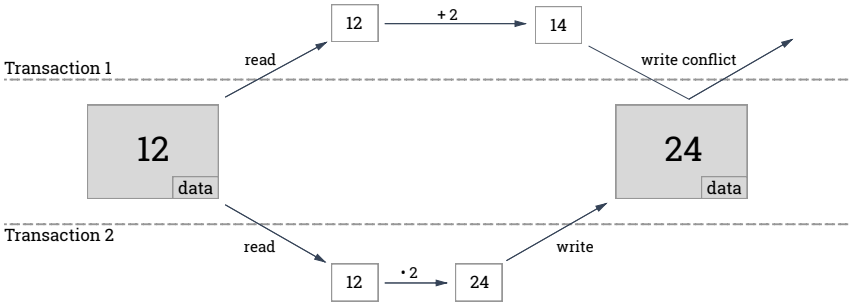
Software Transactional Memory[1]



- uses *transactions* to guard access to shared data
- allows shared data access without locking

[1] Shavit, et al. "Software transactional memory." Distributed Computing 10.2 (1997): 99-116.

Software Transactional Memory[1]



- uses *transactions* to guard access to shared data
- allows shared data access without locking

[1] Shavit, et al. "Software transactional memory." Distributed Computing 10.2 (1997): 99-116.

Motivation

Parallelization approaches for shared state applications:

- Compiler analyses
  - struggle to uncover parallelism (no alias analysis etc)
- Locking
  - coarse grain locking doesn't scale
  - fine grained locking is error-prone
  - resulting program is non-deterministic
- Software Transactional Memory

Motivation

Parallelization approaches for shared state applications:

- Compiler analyses
  - struggle to uncover parallelism (no alias analysis etc)
- Locking
  - coarse grain locking doesn't scale
  - fine grained locking is error-prone
  - resulting program is non-deterministic
- Software Transactional Memory
  - solves composability problem

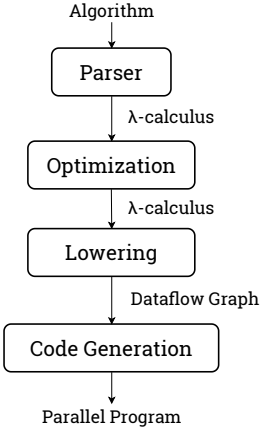
Motivation

Parallelization approaches for shared state applications:

- Compiler analyses
  - struggle to uncover parallelism (no alias analysis etc)
- Locking
  - coarse grain locking doesn't scale
  - fine grained locking is error-prone
  - resulting program is non-deterministic
- Software Transactional Memory
  - solves composability problem
  - programs are still non-deterministic

Ohua[2]

Framework for implicit parallel programming:

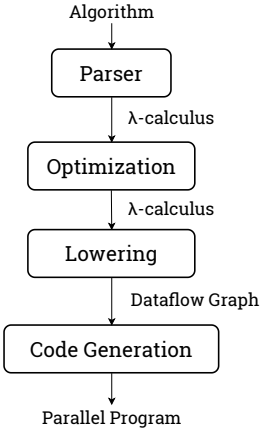


[2] Ertel et al. "Towards Implicit Parallel Programming for Systems." dissertation, 2019.

Ohua[2]

Framework for implicit parallel programming:

- Derives dataflow graph from algorithm file

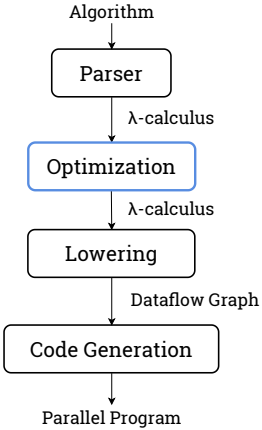


[2] Ertel et al. "Towards Implicit Parallel Programming for Systems." dissertation, 2019.

Ohua[2]

Framework for implicit parallel programming:

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

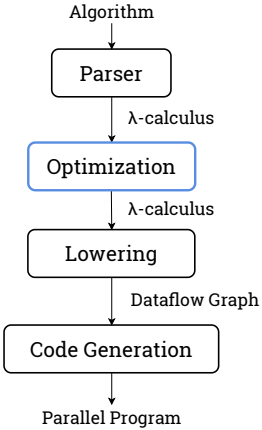


[2] Ertel et al. "Towards Implicit Parallel Programming for Systems." dissertation, 2019.

Ohua[2]

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

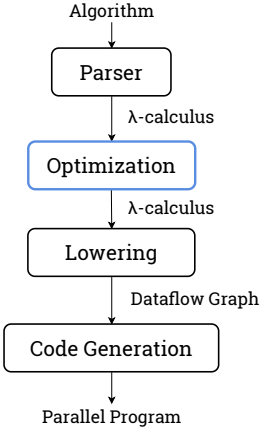


[2] Ertel et al. "Towards Implicit Parallel Programming for Systems." dissertation, 2019.

Ohua[2]

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
- **Result:** Deterministic parallel program



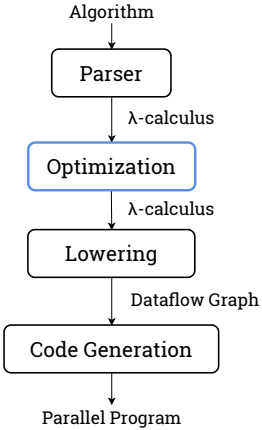
[2] Ertel et al. "Towards Implicit Parallel Programming for Systems." dissertation, 2019.

Ohua[2]

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
- **Result:** Deterministic parallel program

Is Ohua a possible alternative to STM?



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

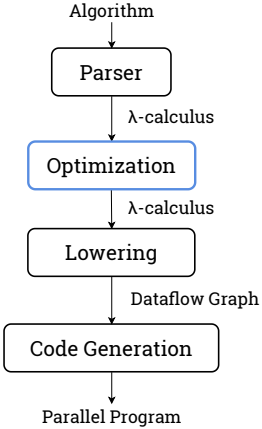
Ohua[2]

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
- **Result:** Deterministic parallel program

Is Ohua a possible alternative to STM?

- Problem: Ohua only fosters local state

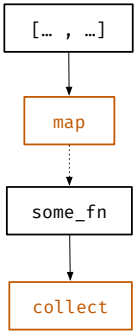


[2] Ertel et al. "Towards Implicit Parallel Programming for Systems." dissertation, 2019.

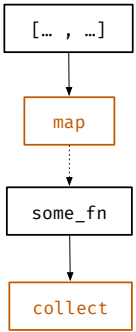
Compiler Transformations

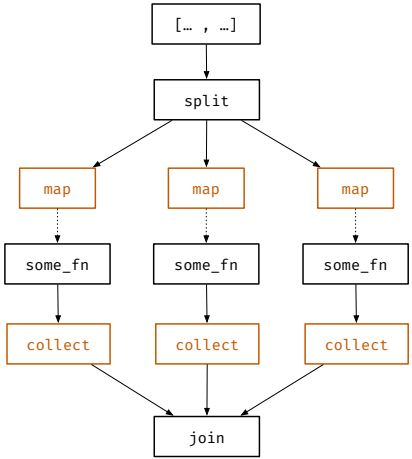
- Transformation 1: Map parallelization

Transformation 1



Transformation 1





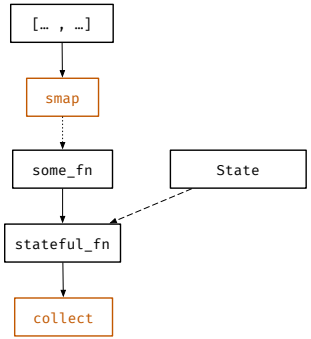
Compiler Transformations

- Transformation 1: Map parallelization

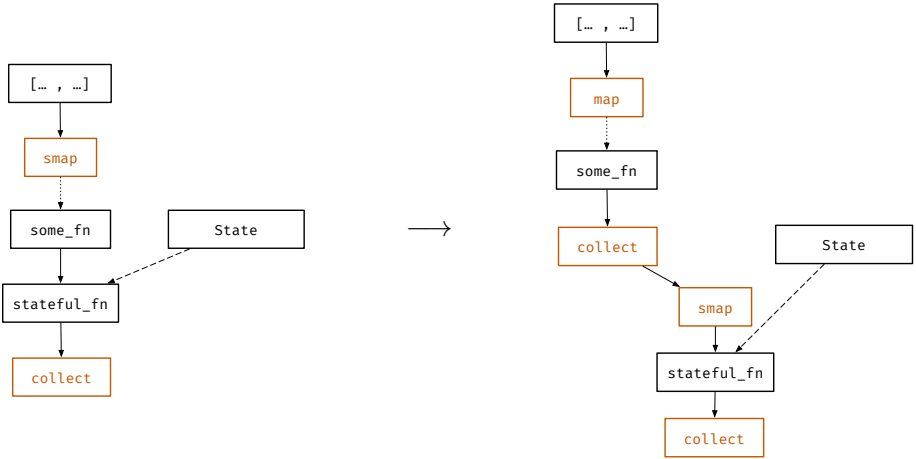
Compiler Transformations

- Transformation 1: Map parallelization
- Transformation 2: State Decoupling

Transformation 2



Transformation 2



Compiler Transformations

- Transformation 1: Map parallelization
- Transformation 2: State Decoupling

Compiler Transformations

- Transformation 1: Map parallelization
- Transformation 2: State Decoupling
- Transformation 3: Batch Updates

Compiler Transformations

- Transformation 1: Map parallelization
- Transformation 2: State Decoupling
- Transformation 3: Batch Updates
  - process only *n* elements from the input list

Compiler Transformations

- Transformation 1: Map parallelization
- Transformation 2: State Decoupling
- Transformation 3: Batch Updates
  - process only *n* elements from the input list
- Transformation 4: Optimize Map parallelization

Compiler Transformations

- Transformation 1: Map parallelization
- Transformation 2: State Decoupling
- Transformation 3: Batch Updates
  - process only *n* elements from the input list
- Transformation 4: Optimize Map parallelization
  - reduce stragglers by implementing work stealing into the runtime

Performance Comparison

- selected 4 representative benchmarks from STAMP[3] suite

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

Performance Comparison

- selected 4 representative benchmarks from STAMP[3] suite
- STAMP benchmarks are written in C using `t12` library

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

Performance Comparison

- selected 4 representative benchmarks from STAMP[3] suite
- STAMP benchmarks are written in C using `tl2` library
- had to be ported to *safe* Rust
  - using library `rust-stm`

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

Performance Comparison

- selected 4 representative benchmarks from STAMP[3] suite
- STAMP benchmarks are written in C using `tl2` library
- had to be ported to *safe* Rust
  - using library `rust-stm`
  - based on algorithm description (no 1:1 translation)

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

Performance Comparison

- selected 4 representative benchmarks from STAMP[3] suite
- STAMP benchmarks are written in C using `tl2` library
- had to be ported to *safe* Rust
  - using library `rust-stm`
  - based on algorithm description (no 1:1 translation)
  - re-implemented specialized data structures

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

Performance Comparison

- verified Rust port by comparing STAMP and Rust results

Performance Comparison

- verified Rust port by comparing STAMP and Rust results
  - **Verification Criterion:** similar scaling behavior

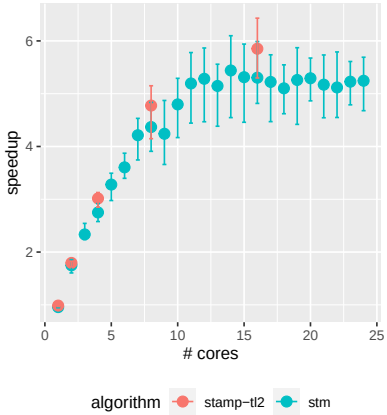
Performance Comparison

- verified Rust port by comparing STAMP and Rust results
  - **Verification Criterion:** similar scaling behavior
- additionally developed Ohua implementations
  - using newly developed transformations

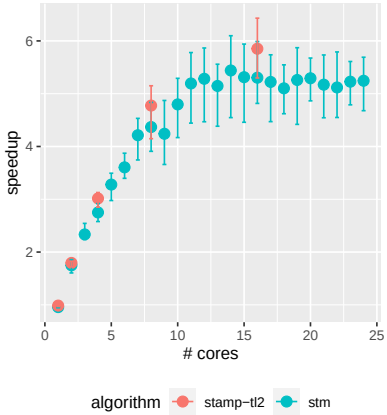
Performance Comparison

- verified Rust port by comparing STAMP and Rust results
  - **Verification Criterion:** similar scaling behavior
- additionally developed Ohua implementations
  - using newly developed transformations
- compared speedups & CPU use for STM and Ohua

Results: Labyrinth

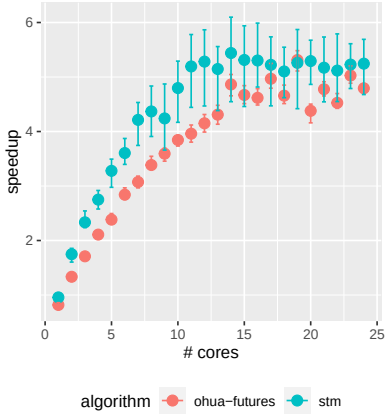


Results: Labyrinth

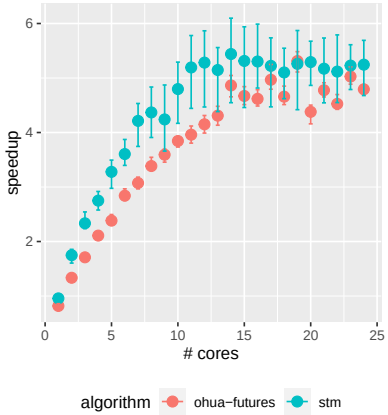


☑ similar scaling behavior

Results: Labyrinth

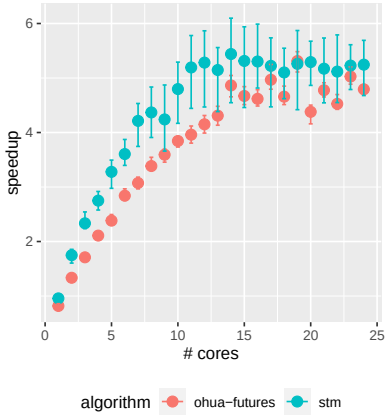


Results: Labyrinth



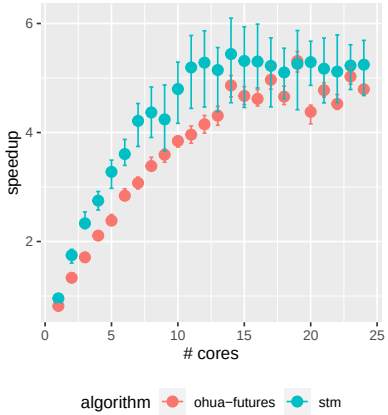
- Ohua almost on par with STM

Results: Labyrinth



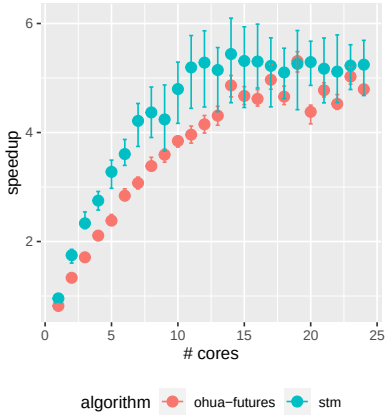
- Ohua almost on par with STM
  - less variance in measured values

Results: Labyrinth



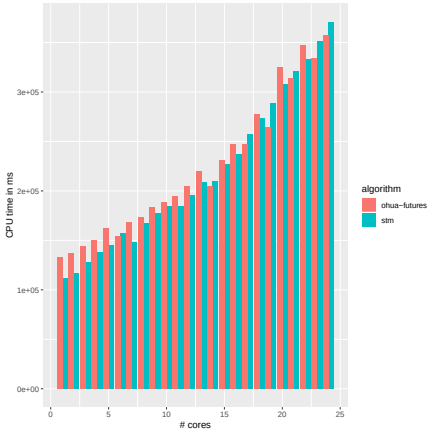
- Ohua almost on par with STM
  - less variance in measured values
- STM: non-deterministic

Results: Labyrinth



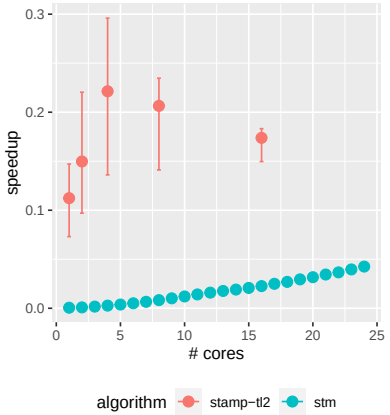
- Ohua almost on par with STM
  - less variance in measured values
- STM: non-deterministic
  - varying execution order

Results: Labyrinth

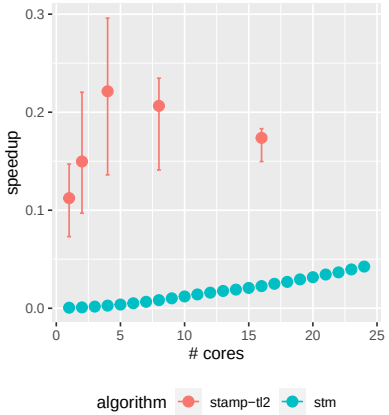


- Ohua almost on par with STM
  - less variance in measured values
- STM: non-deterministic
  - varying execution order
- Ohua uses equally much CPU time as STM

Results: Intruder

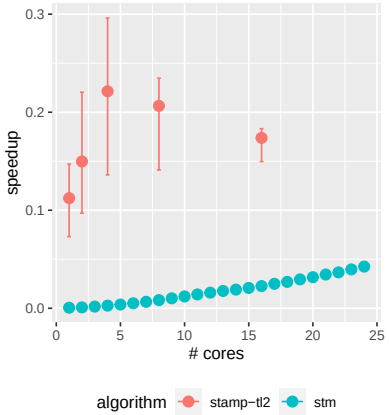


Results: Intruder



- worse scaling behavior

Results: Intruder

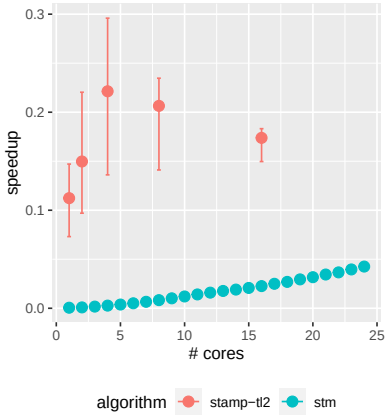


- ☒ worse scaling behavior

Possible Reasons:

- ☐ STAMP is heavily optimized (e.g., own HashMap)

Results: Intruder

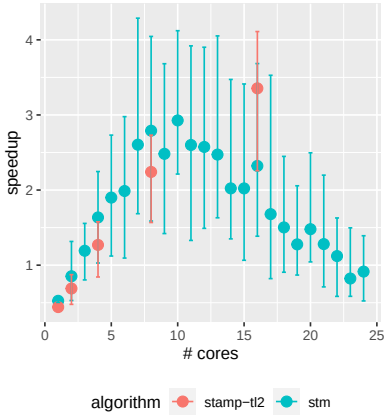


☒ worse scaling behavior

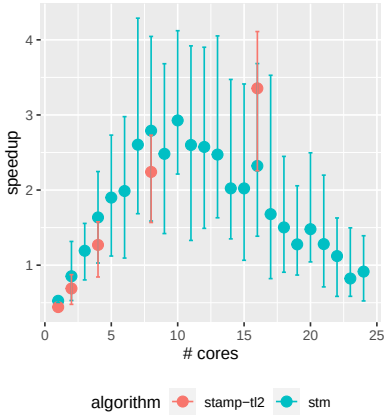
Possible Reasons:

- STAMP is heavily optimized (e.g., own HashMap)
- larger framework overhead in rust-stm

Results: k-means (low contention)

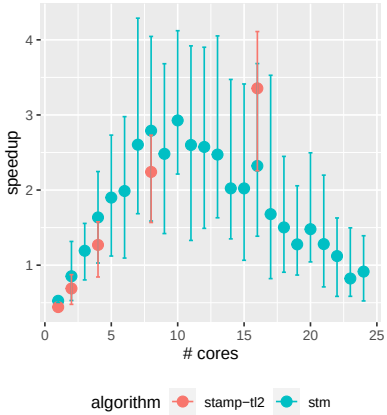


Results: k-means (low contention)



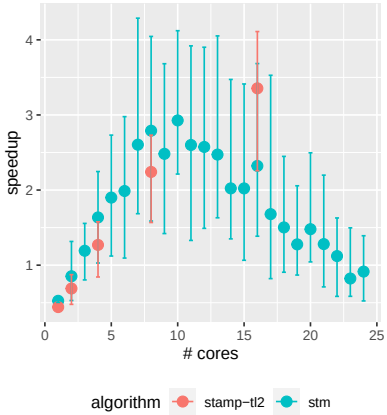
☑ nearly similar scaling behavior

Results: k-means (low contention)



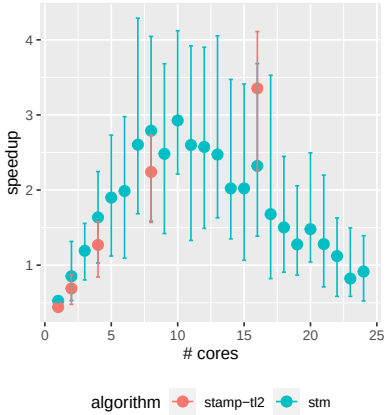
- ☑ nearly similar scaling behavior
- ☐ performance drop for large thread counts

Results: k-means (low contention)



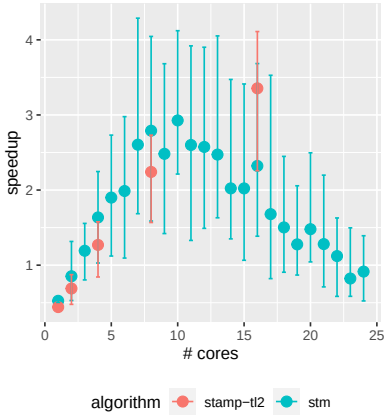
- ☑ nearly similar scaling behavior
- ☐ performance drop for large thread counts
- ☐ **Reason:** differences in memory sharing

Results: k-means (low contention)



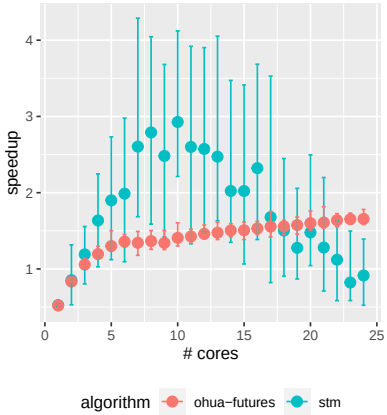
- ☑ nearly similar scaling behavior
- ☐ performance drop for large thread counts
- ☐ **Reason:** differences in memory sharing
  - ☐ C version uses single copy of data structure

Results: k-means (low contention)

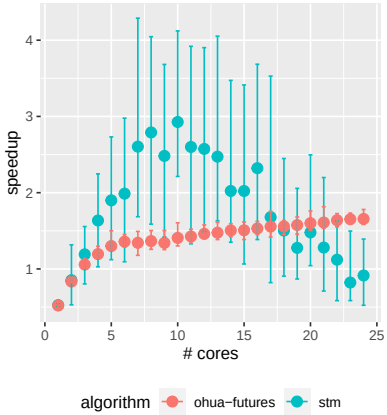


- ☑ nearly similar scaling behavior
- ☐ performance drop for large thread counts
- ☐ **Reason:** differences in memory sharing
  - ☐ C version uses single copy of data structure
  - ☐ safe Rust requires copying of data for each thread

Results: k-means (low contention)

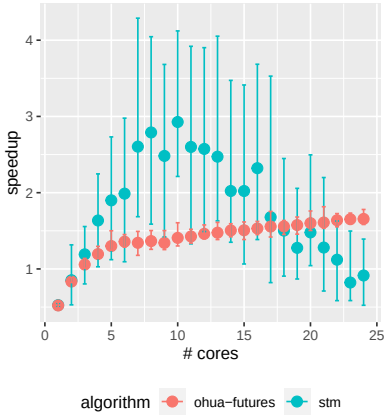


Results: k-means (low contention)



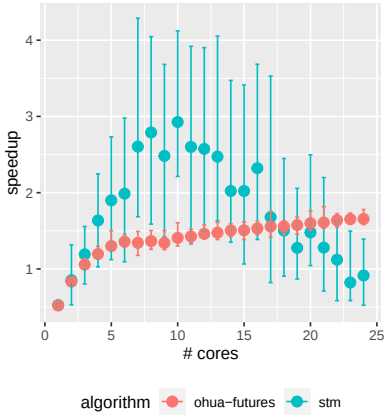
- Ohua does not perform as good as STM

Results: k-means (low contention)



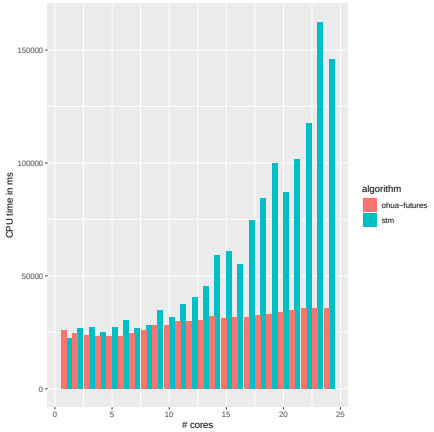
- Ohua does not perform as good as STM
- performance reaches ceiling

Results: k-means (low contention)



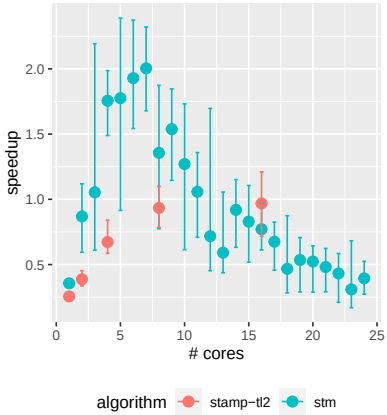
- Ohua does not perform as good as STM
- performance reaches ceiling
  - indicates that not all available parallelism has been exploited

Results: k-means (low contention)

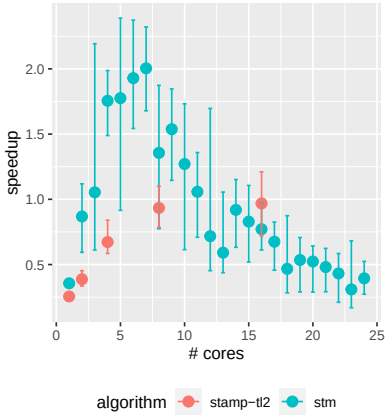


- Ohua does not perform as good as STM
- performance reaches ceiling
  - indicates that not all available parallelism has been exploited
- uses nearly constant, relatively low amount of CPU time

Results: k-means (high contention)

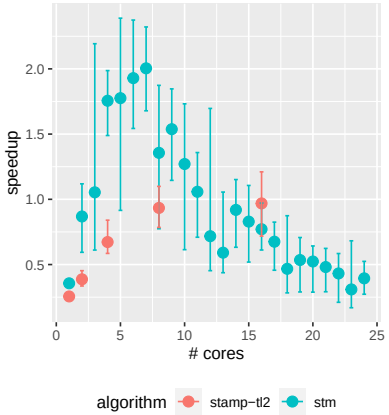


Results: k-means (high contention)



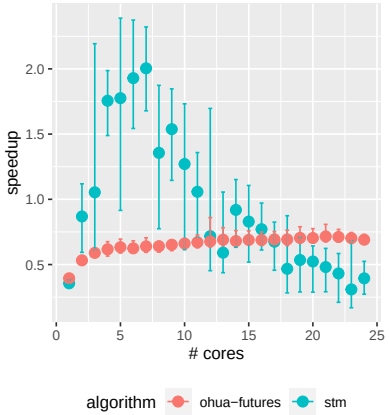
☑ Rust version outperforms STAMP

Results: k-means (high contention)

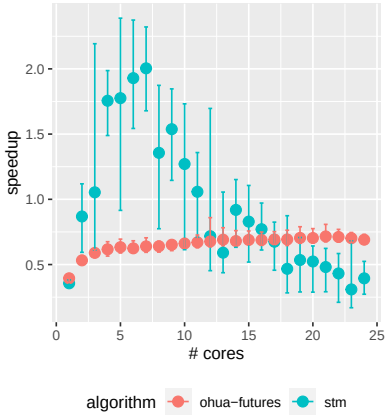


- ☑ Rust version outperforms STAMP
- ☐ same behavior as in low contention run

Results: k-means (high contention)

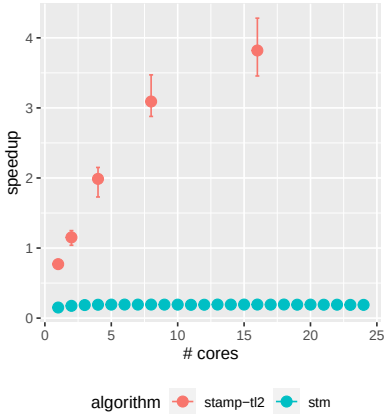


Results: k-means (high contention)

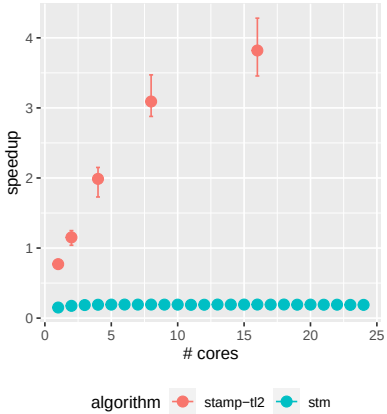


- Ohua again reaches performance ceiling

Results: Genome

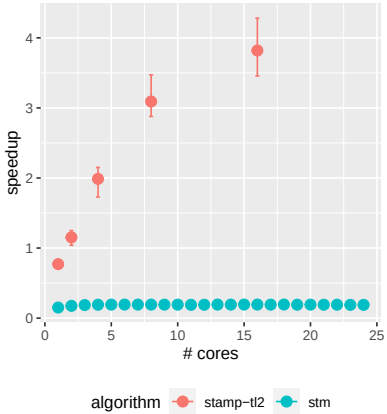


Results: Genome



- completely different curve shapes

Results: Genome

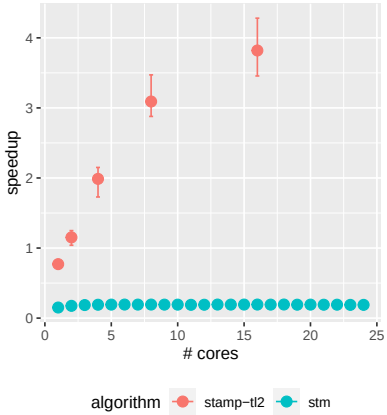


- ☒ completely different curve shapes

Possible Reasons:

- ☐ STAMP is more optimized (e.g., own HashSet)

Results: Genome



- ☒ completely different curve shapes

Possible Reasons:

- ☐ STAMP is more optimized (e.g., own HashSet)
- ☐ high overhead for data duplication (24 % of total time used)

Results: Remarks

- Transformations 2 & 3 proved useful for parallelization

Results: Remarks

- Transformations 2 & 3 proved useful for parallelization
  - used for `labyrinth`

Results: Remarks

- Transformations 2 & 3 proved useful for parallelization
  - used for `labyrinth`
  - only applicable for amorphous data parallel applications

Results: Remarks

- Transformations 2 & 3 proved useful for parallelization
  - used for `labyrinth`
  - only applicable for amorphous data parallel applications
- optimizing other applications requires deeper insight into shared state usage

Results: Remarks

- Transformations 2 & 3 proved useful for parallelization
  - used for `labyrinth`
  - only applicable for amorphous data parallel applications
- optimizing other applications requires deeper insight into shared state usage
- Ohua uses often less or equally as much CPU time as STM

Conclusions

Could Ohua be an alternative to STM for shared state applications?

Conclusions

Could Ohua be an alternative to STM for shared state applications?

- yes, for *some* applications

Conclusions

Could Ohua be an alternative to STM for shared state applications?

- yes, for *some* applications
- can not yet uncover all available parallelism in other applications

Conclusions

Could Ohua be an alternative to STM for shared state applications?

- yes, for *some* applications
- can not yet uncover all available parallelism in other applications

Future Work:

- understand, how shared data structures could be partitioned for parallel accesses

Conclusions

Could Ohua be an alternative to STM for shared state applications?

- yes, for *some* applications
- can not yet uncover all available parallelism in other applications

Future Work:

- understand, how shared data structures could be partitioned for parallel accesses
  - identifiable using certain functions defined on a type?

Conclusions

Could Ohua be an alternative to STM for shared state applications?

- yes, for *some* applications
- can not yet uncover all available parallelism in other applications

Future Work:

- understand, how shared data structures could be partitioned for parallel accesses
  - identifiable using certain functions defined on a type?
  - could a special data type be defined to allow better compiler insight?

Thank you for your attention.

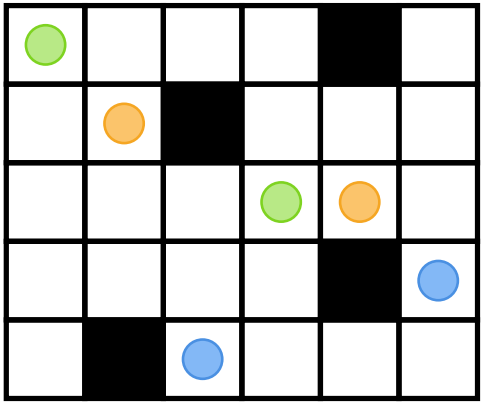
Backup

Backup: Representative Benchmark Selection

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

Backup: Labyrinth Benchmark

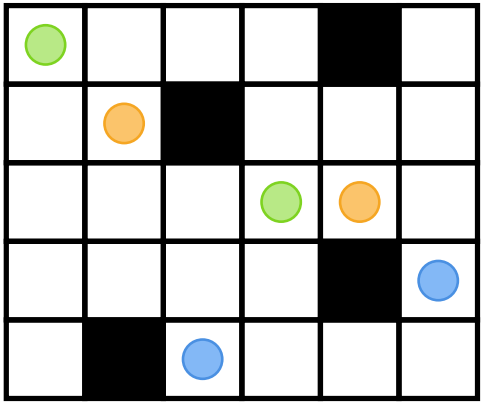
Given: 3D maze, pairs of points



Backup: Labyrinth Benchmark

Given: 3D maze, pairs of points

Goal: Map a path between each pair of points



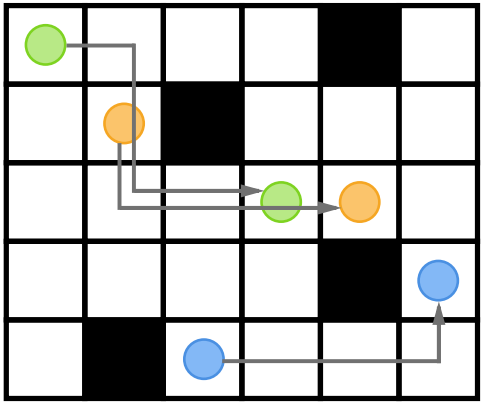
Backup: Labyrinth Benchmark

Given: 3D maze, pairs of points

Goal: Map a path between each pair of points

Implementation:

- parallel search for new paths



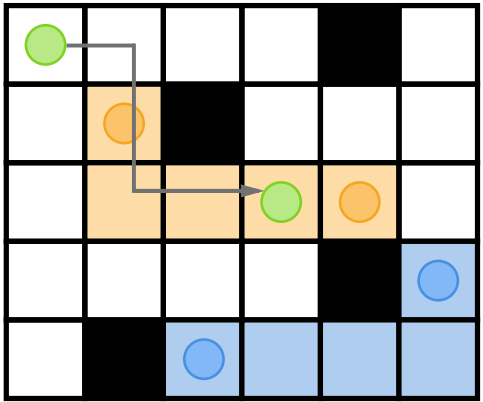
Backup: Labyrinth Benchmark

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



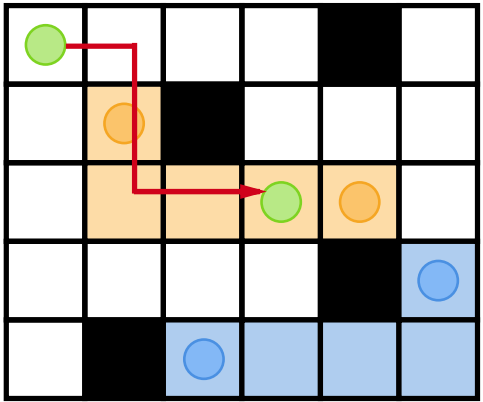
Backup: Labyrinth Benchmark

Given: 3D maze, pairs of points

Goal: Map a path between each pair of points

Implementation:

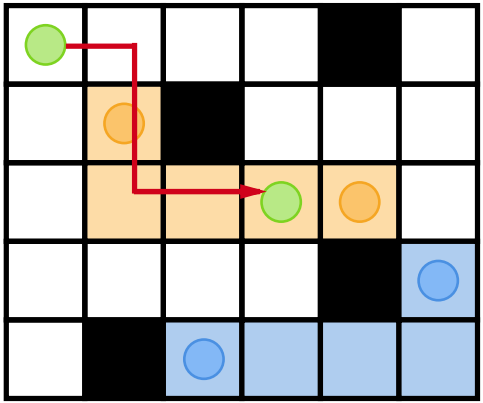
- parallel search for new paths
- merge paths into the maze
  → retry if path crosses other paths



Shared State Application: Labyrinth Benchmark

Irregular Applications

centered around the manipulation of pointer-based data structures

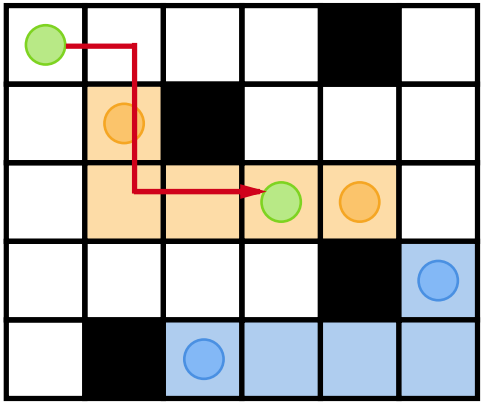


Shared State Application: Labyrinth Benchmark

Irregular Applications

centered around the manipulation of pointer-based data structures

- structure makes identification of safe parallel accesses hard

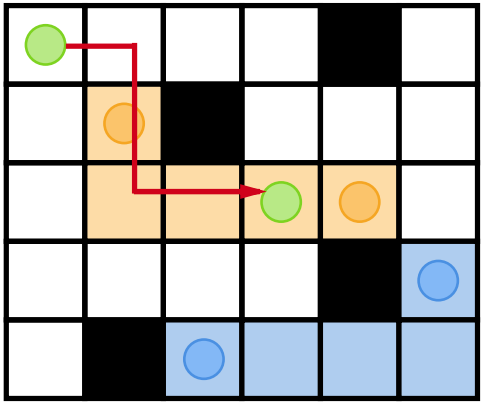


Shared State Application: Labyrinth Benchmark

Irregular Applications

centered around the manipulation of pointer-based data structures

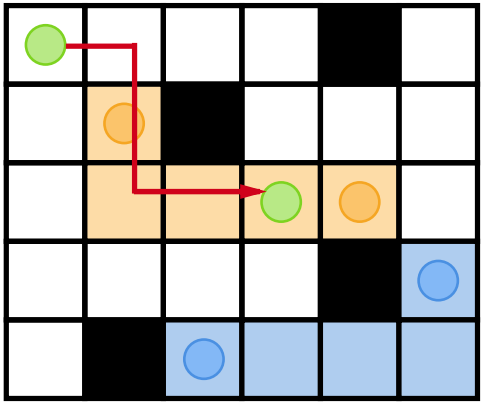
- structure makes identification of safe parallel accesses hard
- compiler analyses struggle to uncover meaningful parallelism



Shared State Application: Labyrinth Benchmark

Amorphous Data Parallelism

behaviour observed in some irregular applications

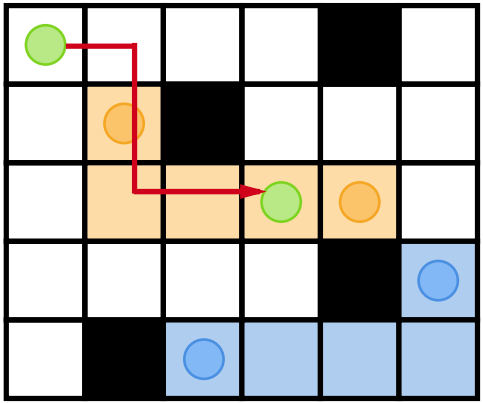


Shared State Application: Labyrinth Benchmark

Amorphous Data Parallelism

behaviour observed in some irregular applications

- processing one element may generate new work items or remove others

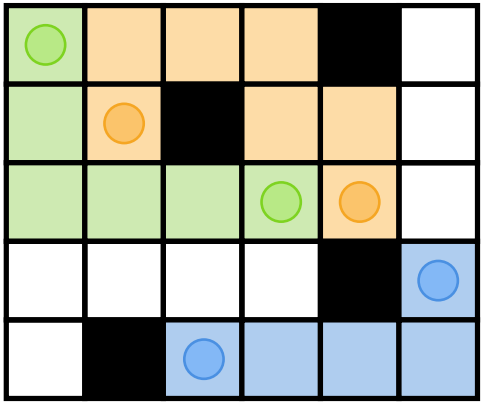


Shared State Application: Labyrinth Benchmark

Amorphous Data Parallelism

behaviour observed in some irregular applications

- processing one element may generate new work items or remove others

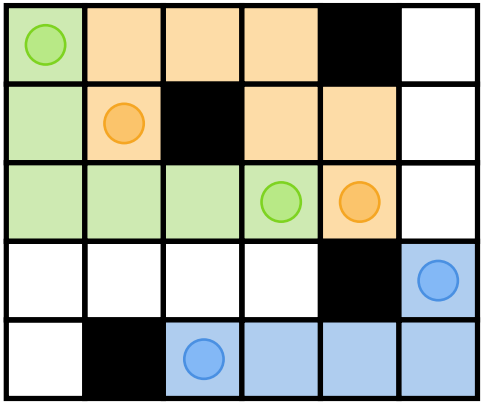


Shared State Application: Labyrinth Benchmark

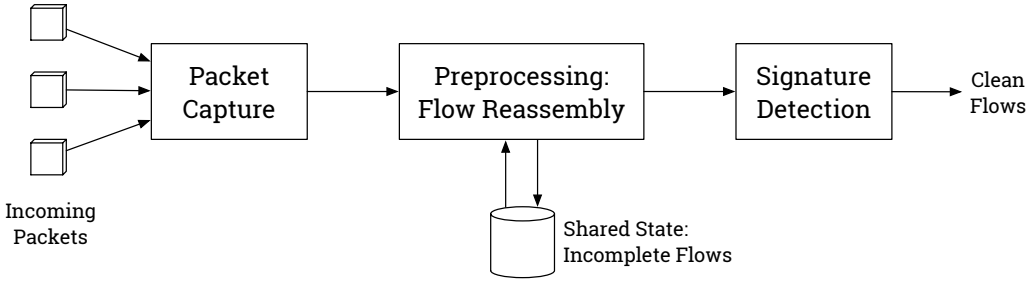
Amorphous Data Parallelism

behaviour observed in some irregular applications

- processing one element may generate new work items or remove others
- some items cannot be processed in parallel due to conflicts



Backup: Intruder Benchmark

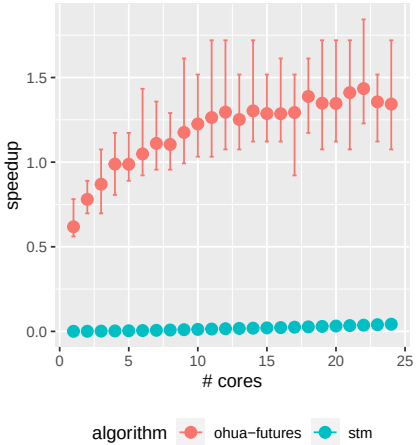


Backup: Intruder Benchmark

```
let mut flows = State::new();
for packet in input {
    flows.add(packet);
}

for flow in flows {
    analyze(flow);
}
```

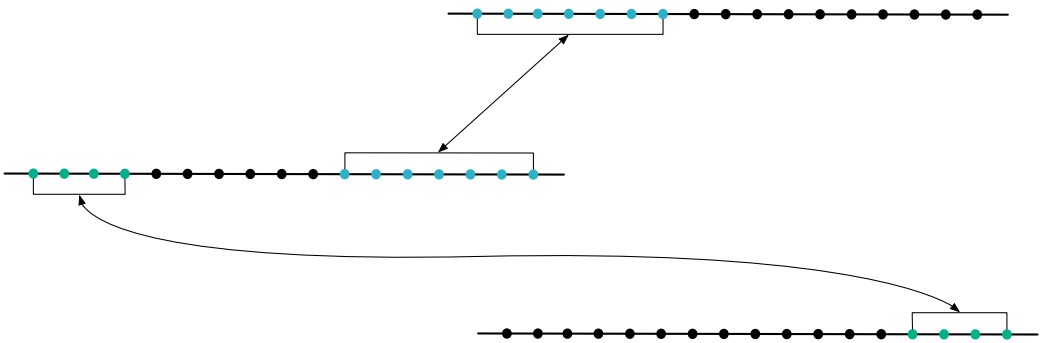
Backup: Intruder Benchmark



Backup: k-means Benchmark

```
let mut centers = initialize(input);
loop {
    // data parallelism
    for item in input {
        item.find_center(centers);
    }
    // fold
    centers = recompute(input);
}
```

Backup: Genome Benchmark - Overlap Matching

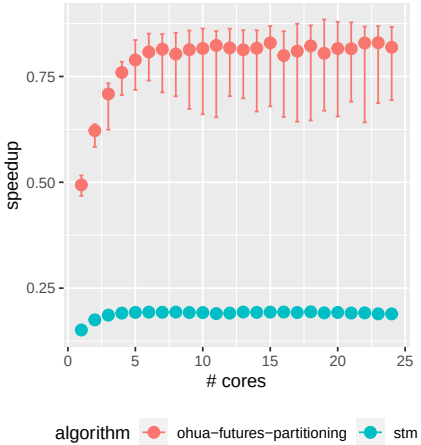


Backup: Genome Benchmark

```
let mut nucleotides = Hashset::new();
for segment in input {
    // deduplication
    nucleotides.insert(segment);
}

loop {
    for item in nucleotides {
        item.find_neighbor(nucleotides);
    }
}
```

Backup: Genome Benchmark



Backup: Differences in Memory Sharing

```
unsigned long data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

pid_t pid = fork();
if (pid != 0) {
    int lower = 0;
    int upper = 4;
    // changes elements at indices 0 to 4
    modify_elements(data, lower, upper);
} else {
    int lower = 5;
    int upper = 9;
    // changes elements at indices 5 to 9
    modify_elements(data, lower, upper);
}
```