

Parallel Fringe Search

Lukas Mosimann & Christian Zeman

ETH Zürich

Design of Parallel and High-Performance Computing

December 15, 2013

Overview

1 Fringe Search

2 What we have done

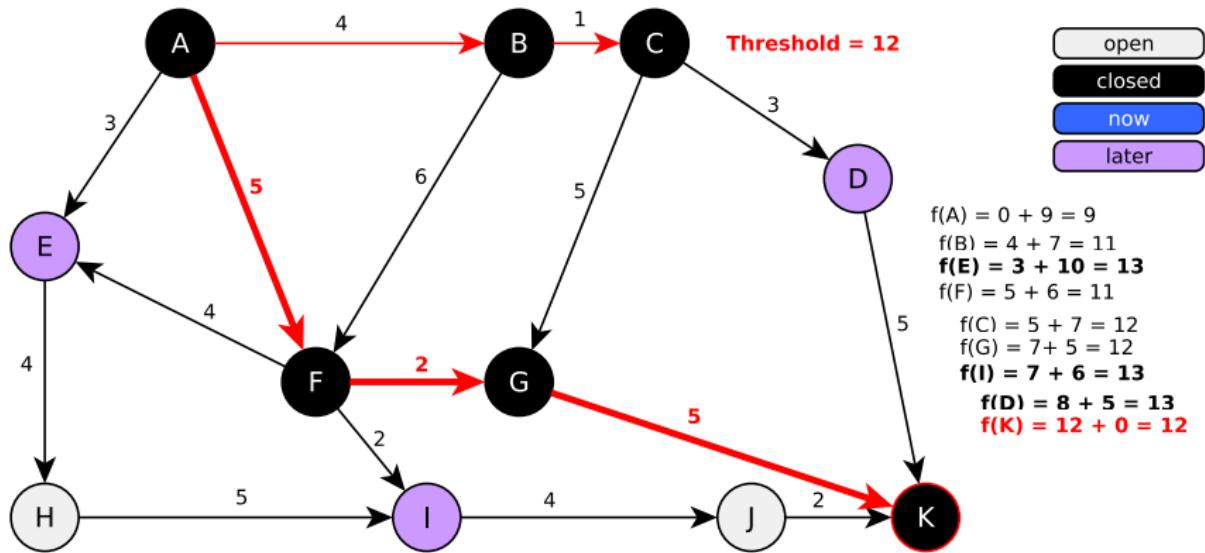
3 Locking concepts

4 Benchmarks

- Locks
- Strong scaling
- Weak scaling
- Path length
- Threshold handling

5 Conclusions

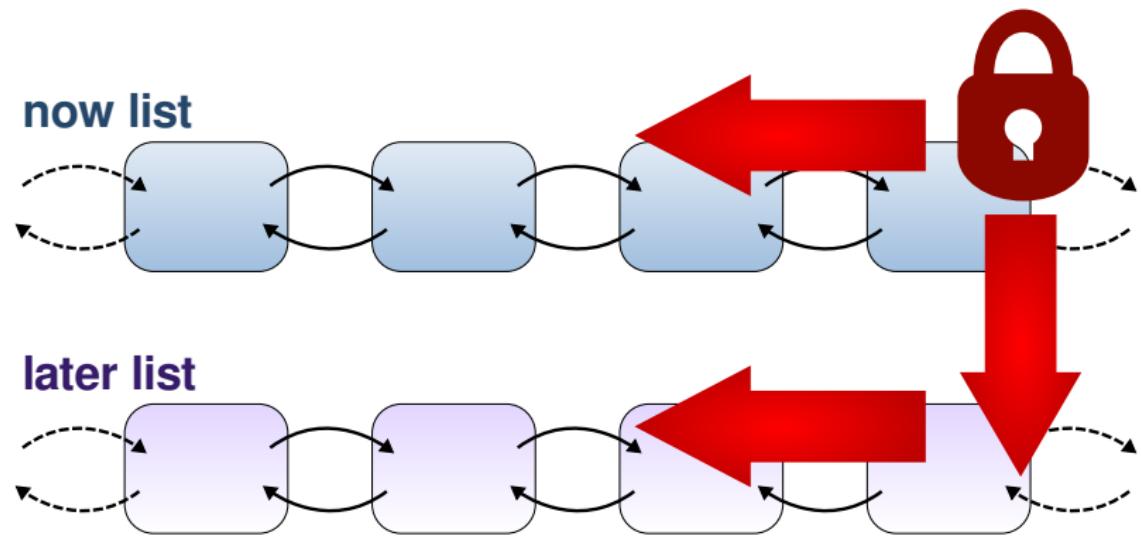
Fringe Search



What we have done

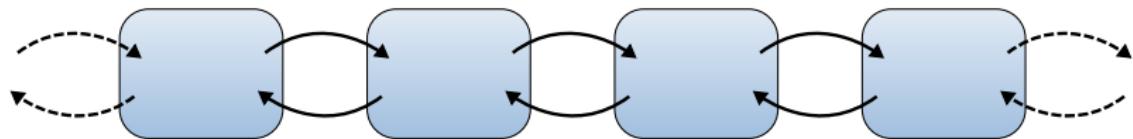
- Serial implementation of fringe search (much faster than Boost A*)
- Parallel implementation with Open MP
 - 2 different locking concepts
 - Locks implemented using inline assembly (faster than Open MP locks)
- Benchmarking
 - Strong scaling
 - Weak scaling
 - Path quality

Locking concept: Deadlock prevention

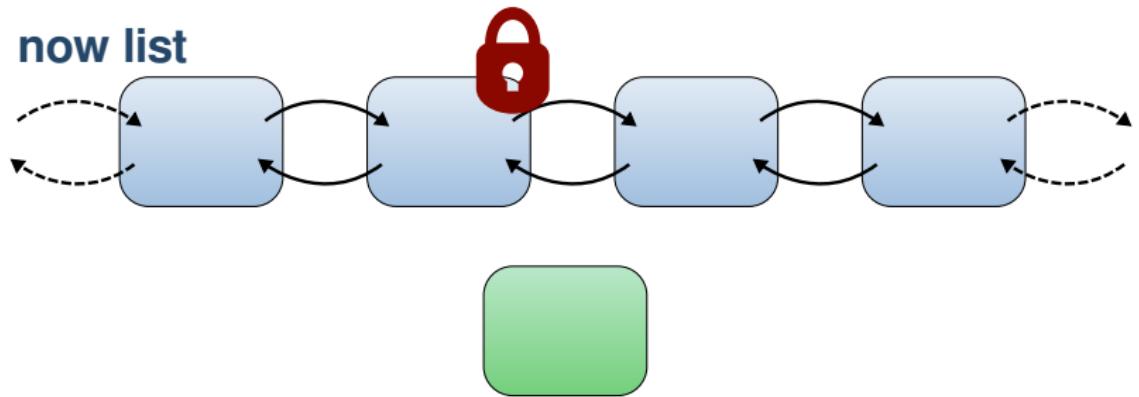


Locking concept: Insert node

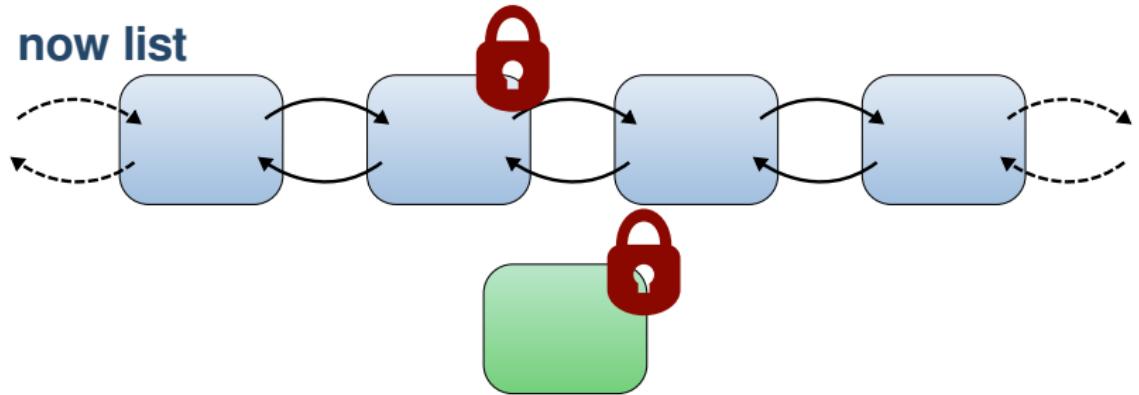
now list



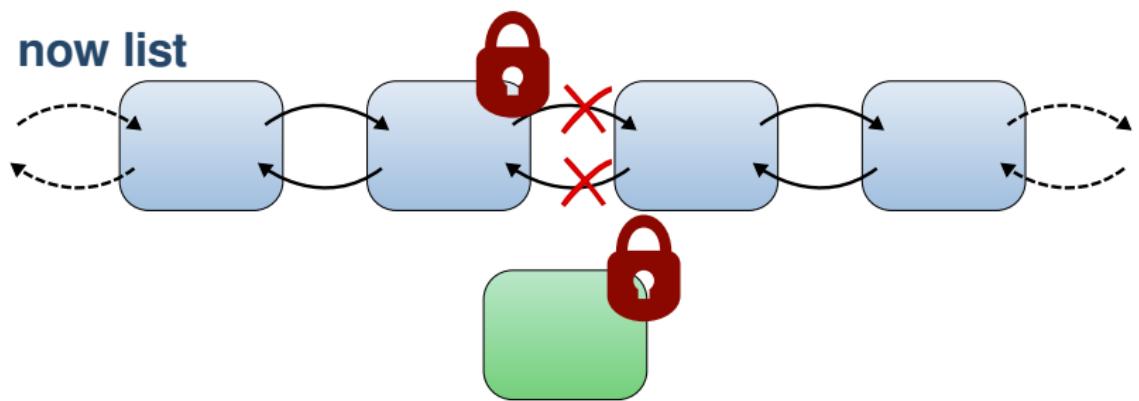
Locking concept: Insert node



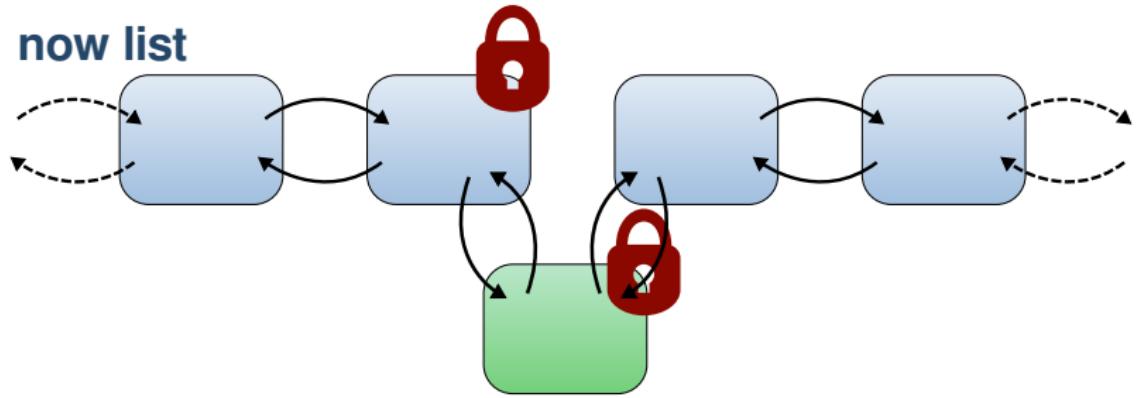
Locking concept: Insert node



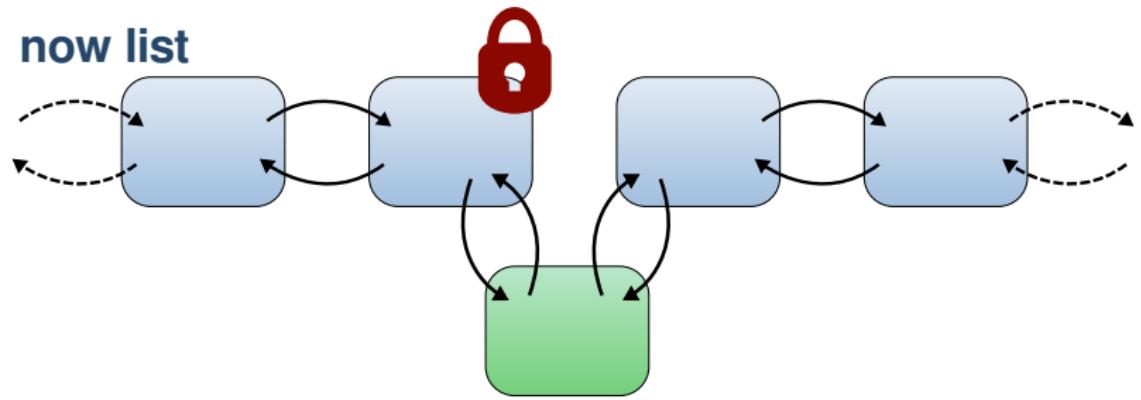
Locking concept: Insert node



Locking concept: Insert node

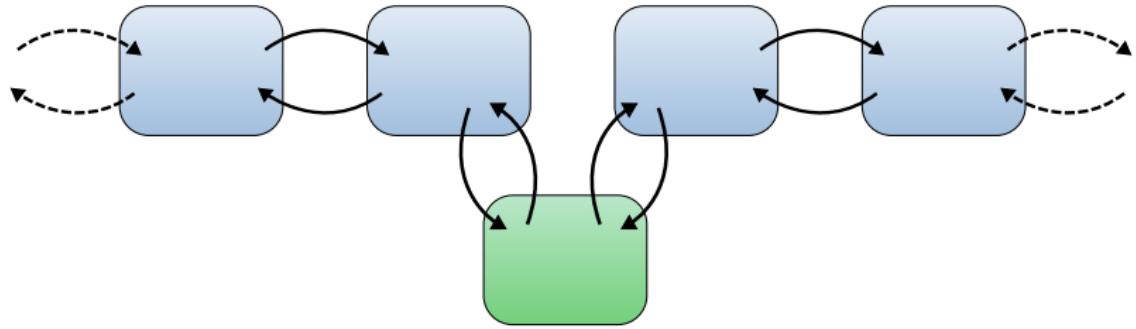


Locking concept: Insert node



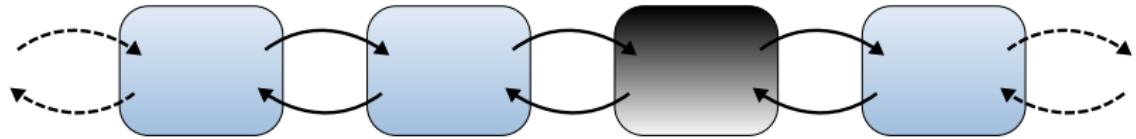
Locking concept: Insert node

now list

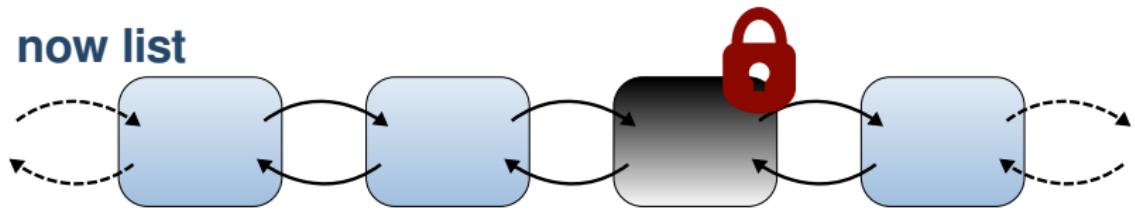


Locking concept: Remove node

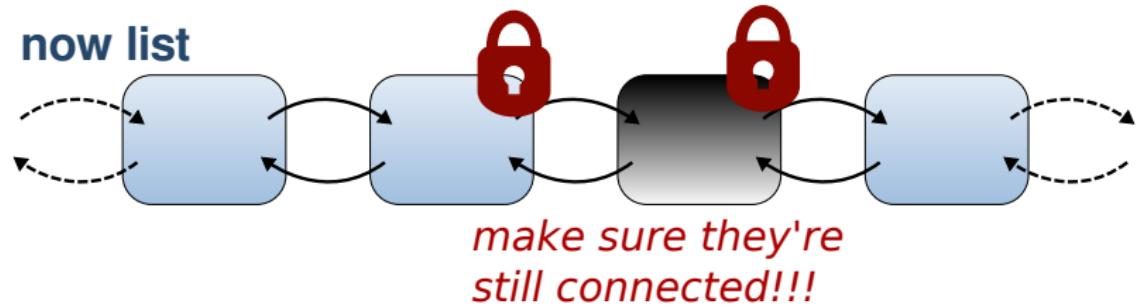
now list



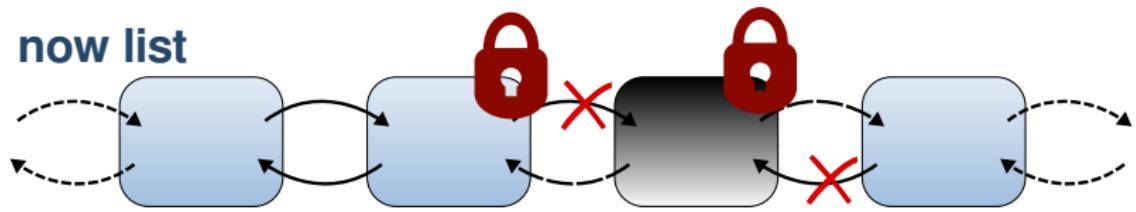
Locking concept: Remove node



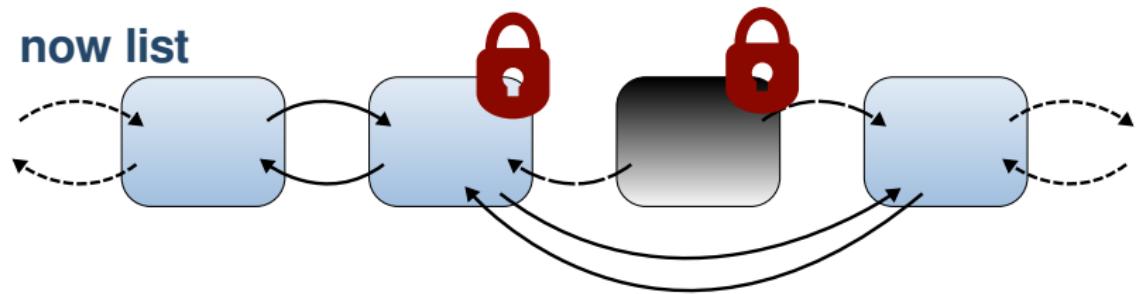
Locking concept: Remove node



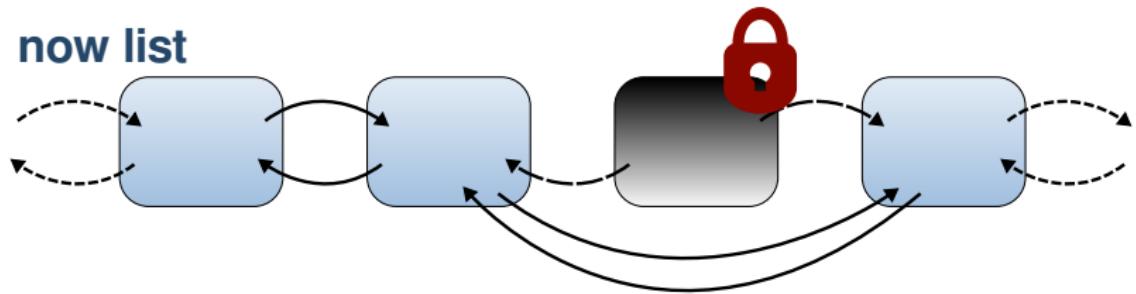
Locking concept: Remove node



Locking concept: Remove node

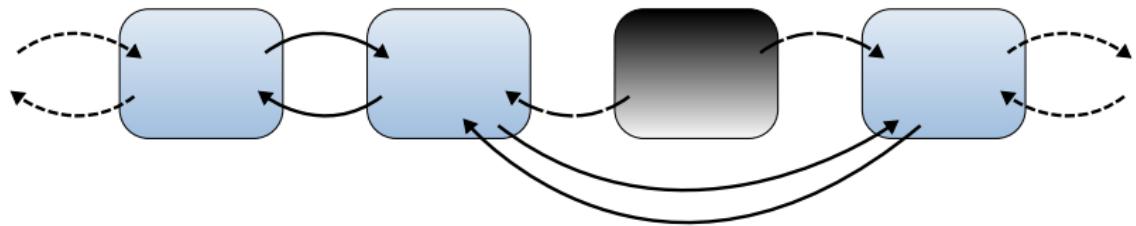


Locking concept: Remove node



Locking concept: Remove node

now list



Locking concept: 2 concepts for removing nodes

Both concepts use optimistic locking for acquiring the locks.

Normal:

- Lock node and predecessor as shown before and remove it right away

Lazy locking:

- Don't lock anything and just mark the node as removed
- Other threads will clean up and remove it later

Benchmarking

Setup:

Each of the following boxplots was generated from data from **50 runs** on **1 node** of kanifushi.inf.ethz.ch.

Specifications of kanifushi.inf.ethz.ch:

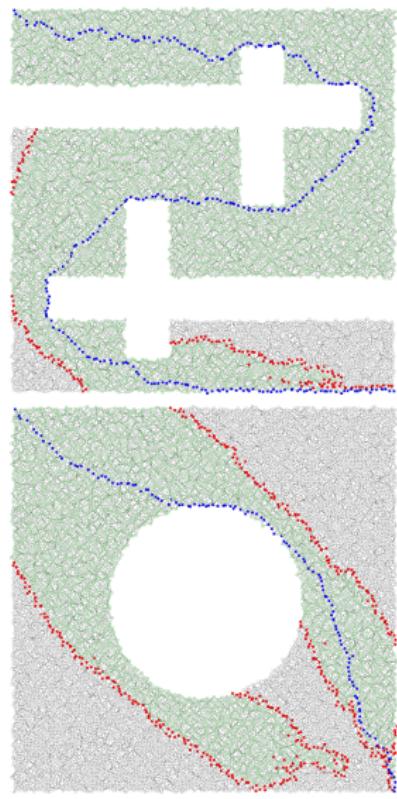
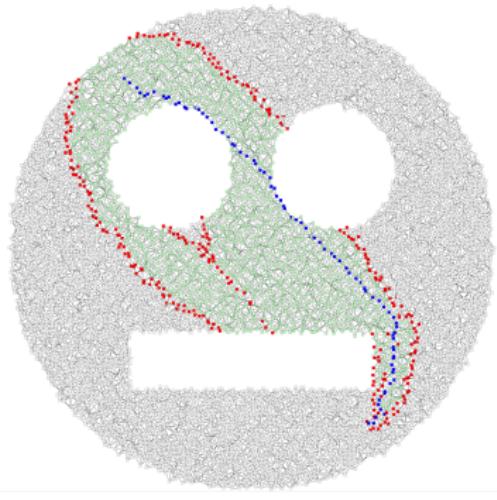
- NUMA model with 32 CPUs on 4 nodes
- 8 CPUs per node
- Intel(R) Xeon(R) CPU E7- 4830 @ 2.13GHz
- per CPU: 32KB L1 cache, 256KB L2 cache
- per node: 24MB L3 cache, 16GB memory

The code is written in C++ / Open MP and it has been compiled with g++ v. 4.6.1 using O1 optimization.

Benchmarking

Graphs

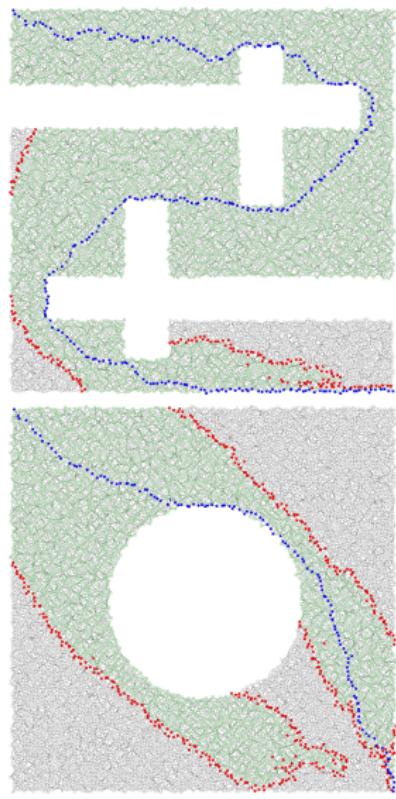
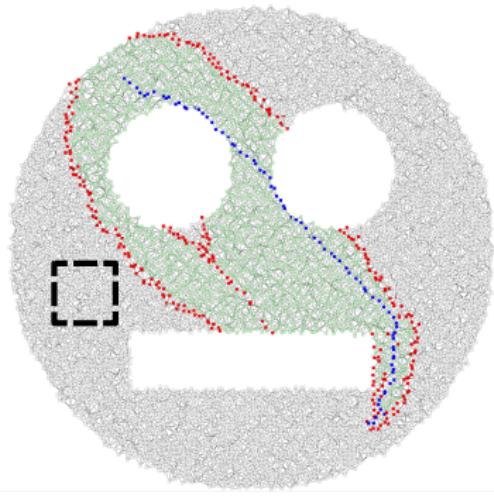
- based on a regular grid with distance 1 and 8 edges per node
- each node has standard deviation of 0.3
- different obstacles (circle, crosses, etc.)



Benchmarking

Graphs

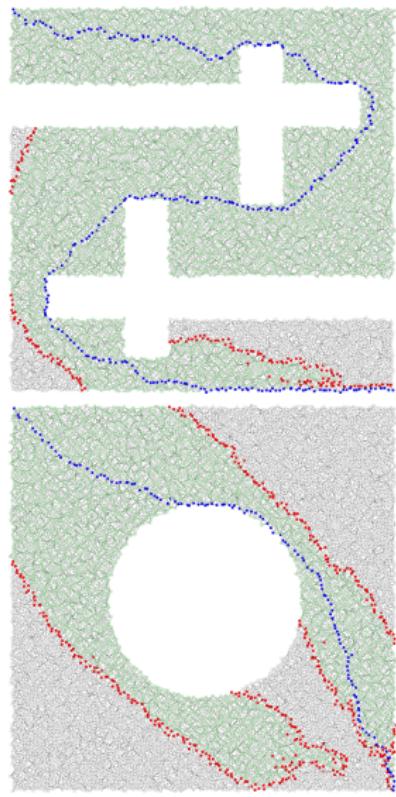
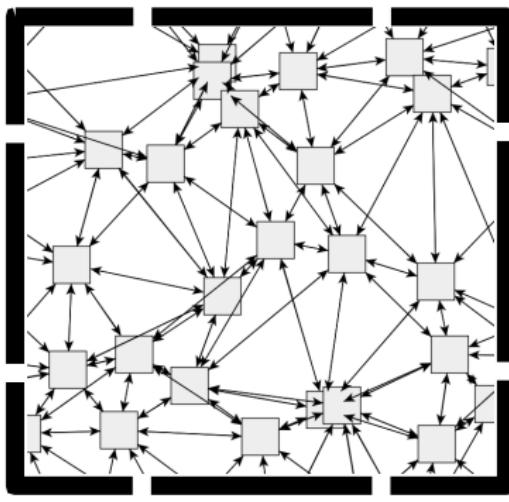
- based on a regular grid with distance 1 and 8 edges per node
- each node has standard deviation of 0.3
- different obstacles (circle, crosses, etc.)



Benchmarking

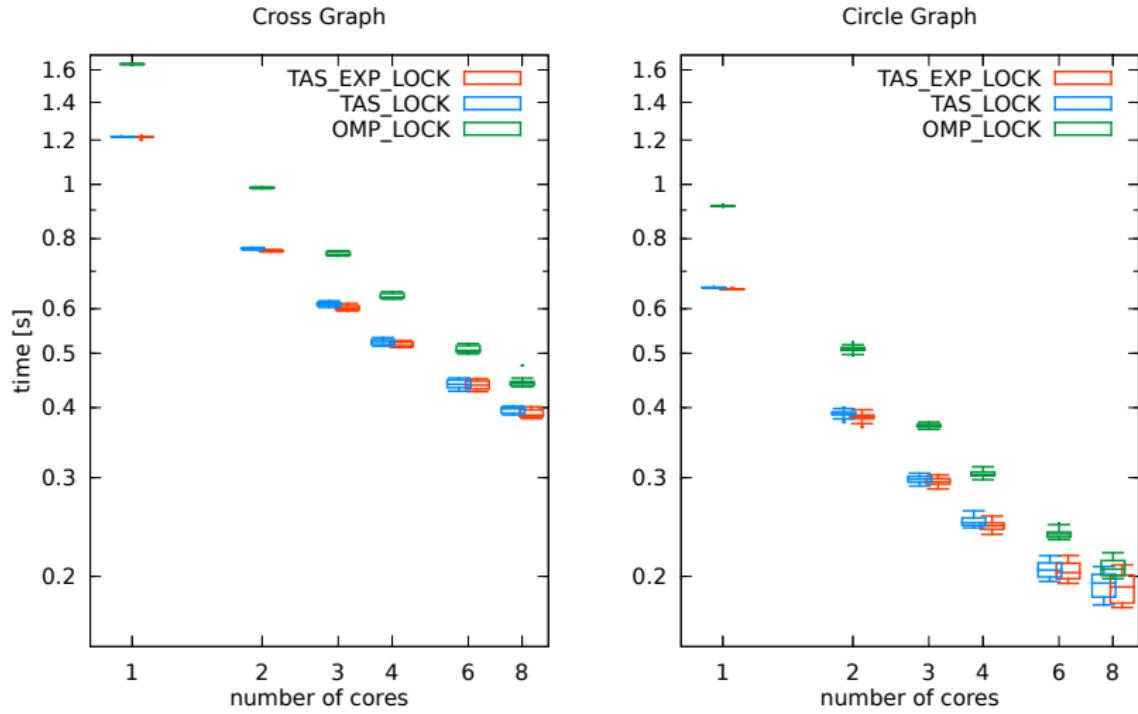
Graphs

- based on a regular grid with distance 1 and 8 edges per node
- each node has standard deviation of 0.3
- different obstacles (circle, crosses, etc.)



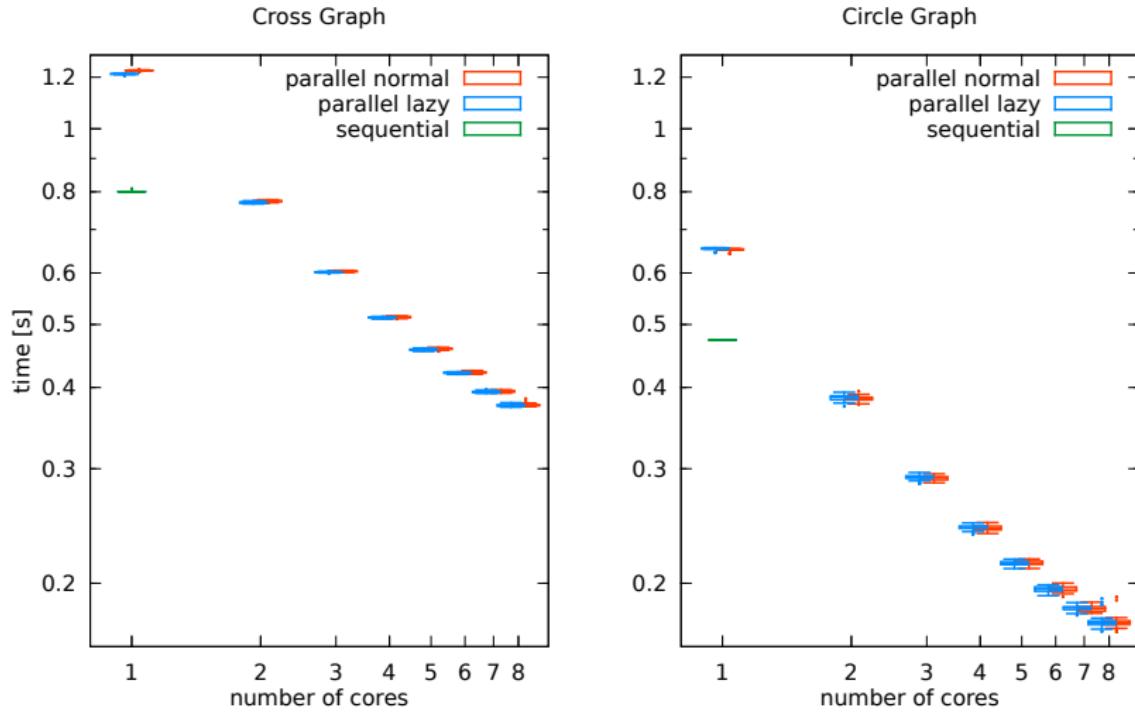
Benchmarking: Locks

Strong scaling 2048 x 2048, threshold=1



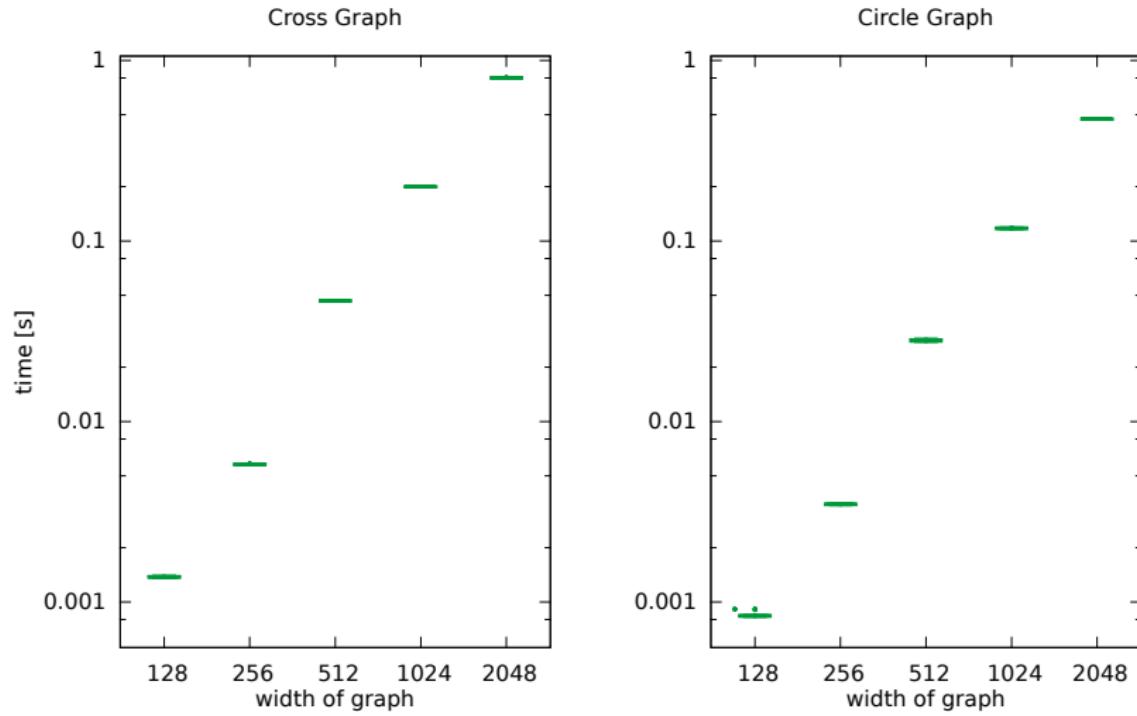
Benchmarking: Strong scaling

Strong scaling 2048 x 2048, threshold=1



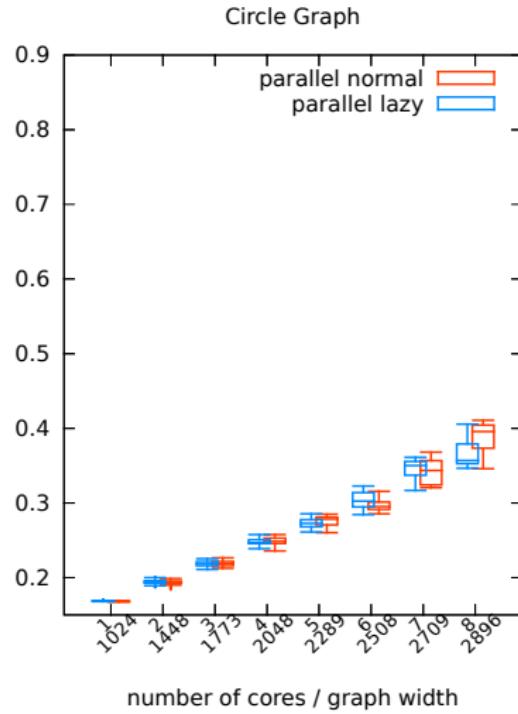
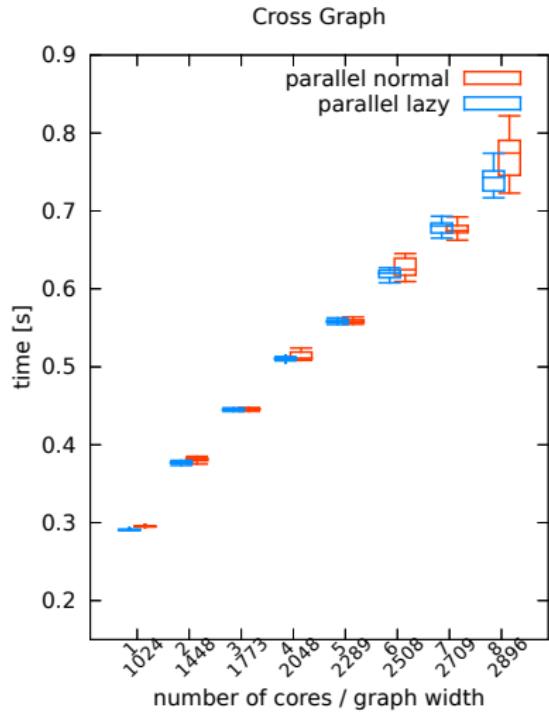
Benchmarking: Sequential scaling

Scaling sequential, threshold=1



Benchmarking: Weak scaling

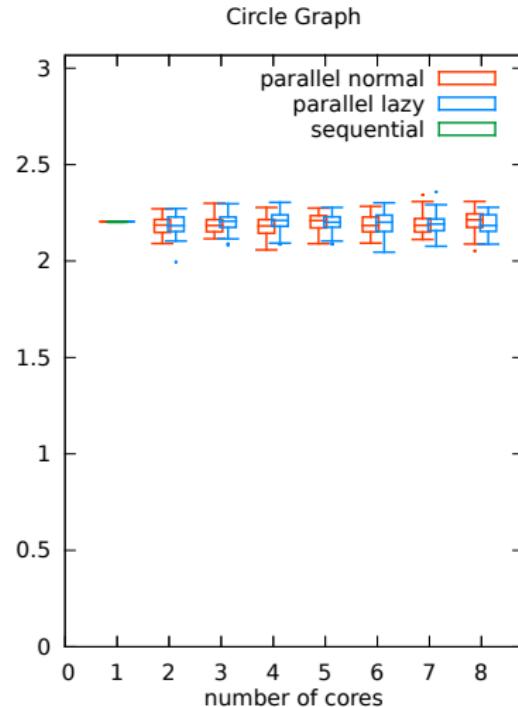
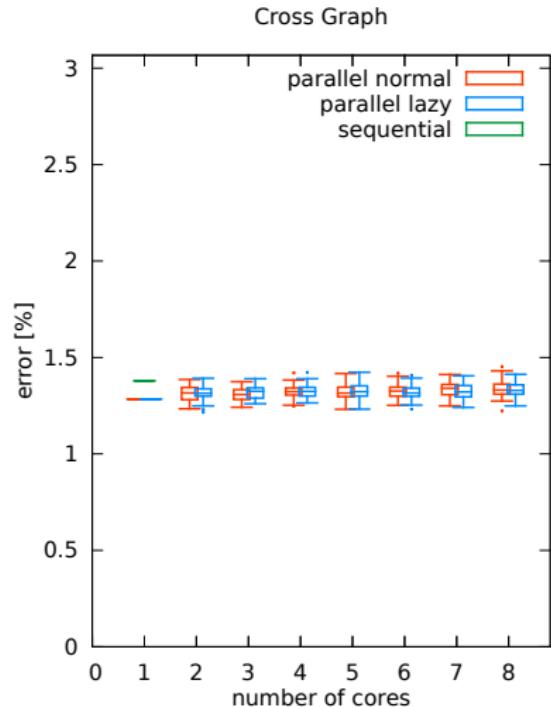
Weak scaling, threshold=1



Benchmarking: Path length \leftrightarrow # cores

compared to A* from Boost Graph Library

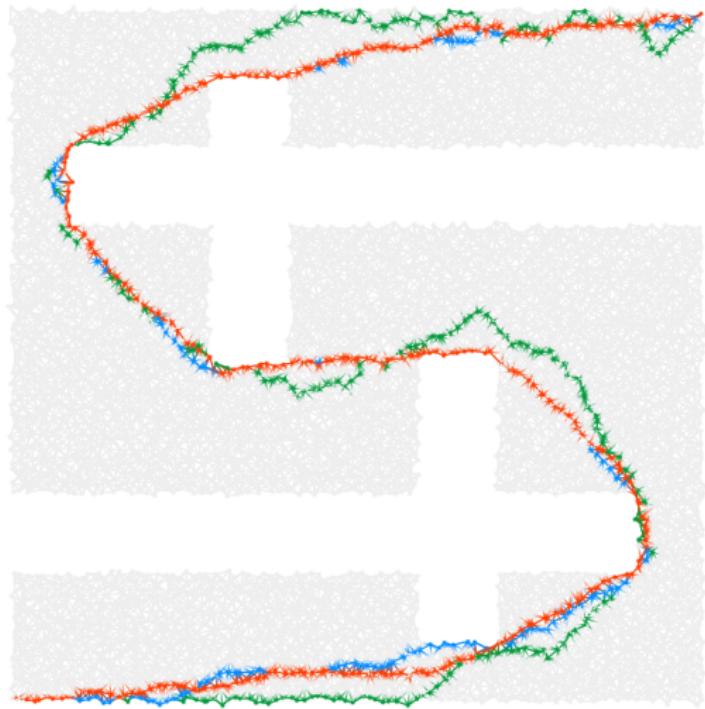
relative Error, 2048 x 2048, threshold=1



Benchmarking: Path length \leftrightarrow Threshold update

Threshold update

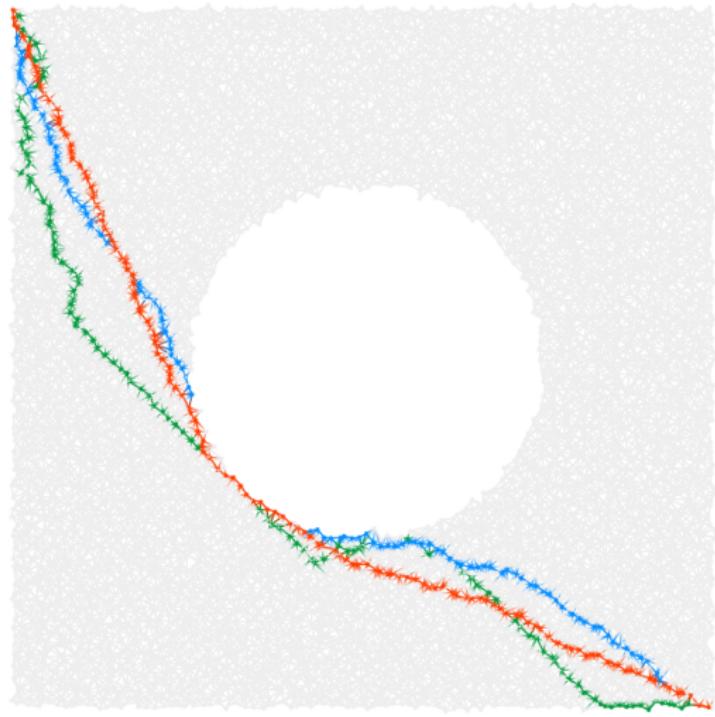
- threshold $+= 0.1$
- threshold $+= 1$
- threshold $+= 10$



Benchmarking: Path length \leftrightarrow Threshold update

Threshold update

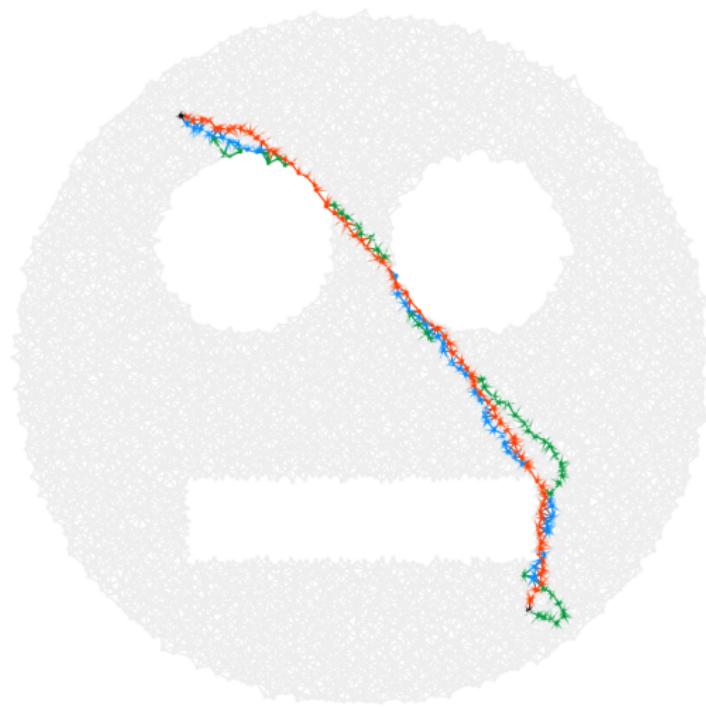
- threshold $+= 0.1$
- threshold $+= 1$
- threshold $+= 10$



Benchmarking: Path length \leftrightarrow Threshold update

Threshold update

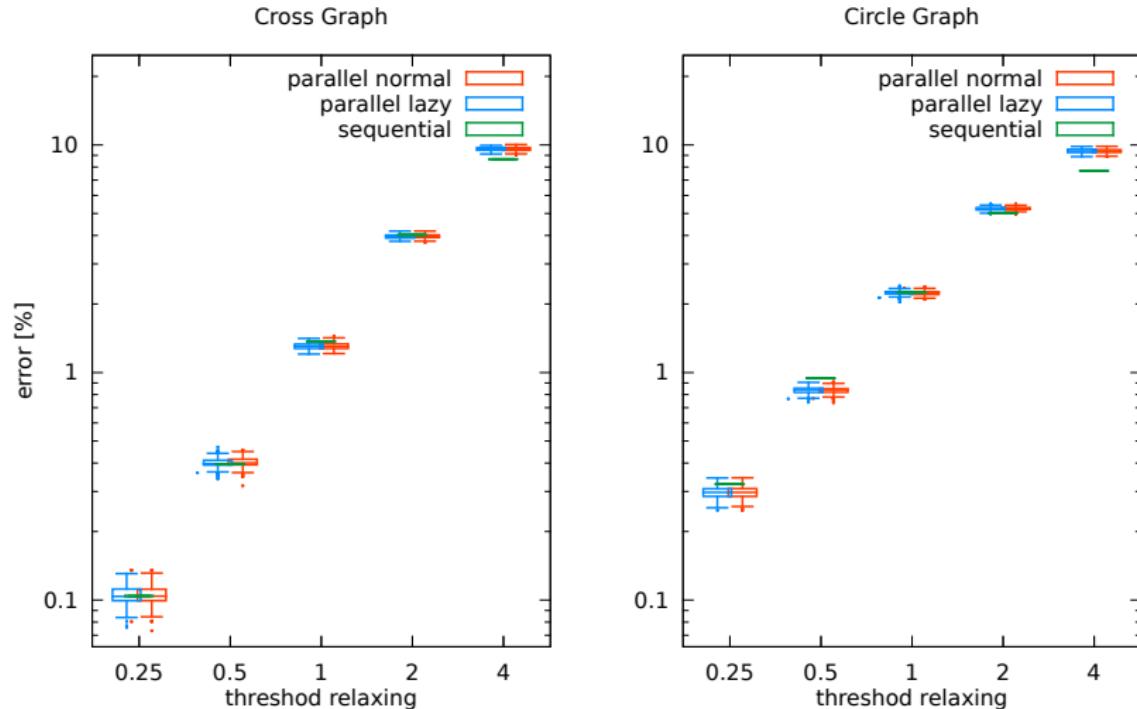
- threshold $+= 0.1$
- threshold $+= 1$
- threshold $+= 10$



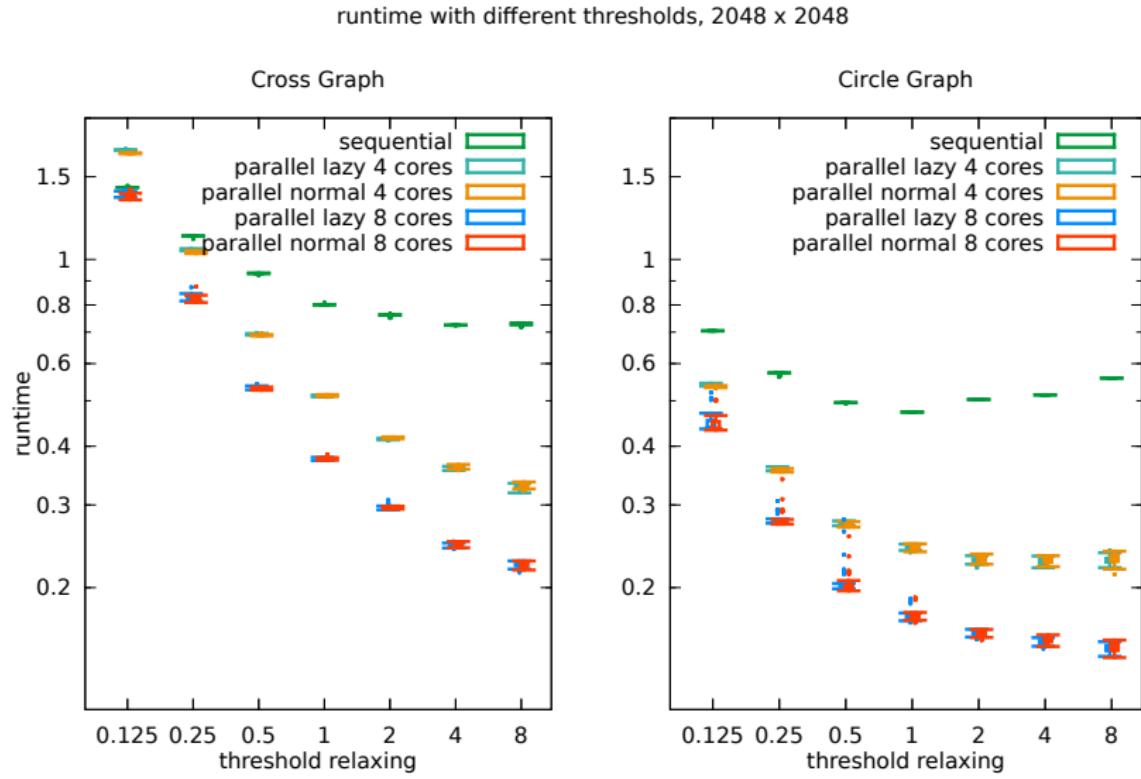
Benchmarking: Path length \leftrightarrow Threshold update

compared to A* from Boost Graph Library

relative Error, 2048 x 2048



Benchmarking: Run time \leftrightarrow Threshold update



Conclusions

In general Fringe Search is a good single source shortest path algorithm, that can be very well implemented in parallel.

- Path quality not dependent of # cores
- Good strong scaling
- Weak scaling is not perfect
- quality \leftrightarrow runtime trade-off can be tuned for desired result

The End