Parallel Fringe Search

Lukas Mosimann & Christian Zeman

ETH Zürich

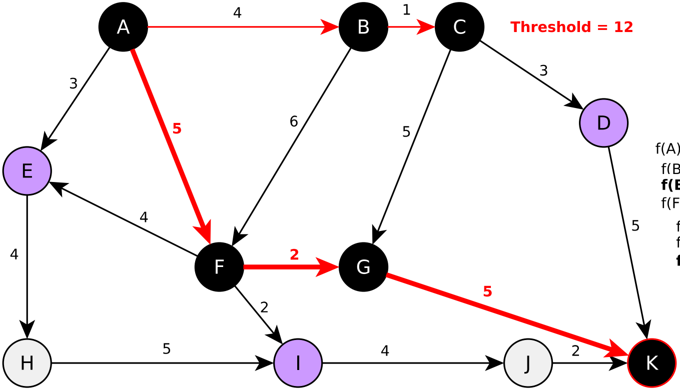
Design of Parallel and High-Performance Computing

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



Threshold = 12

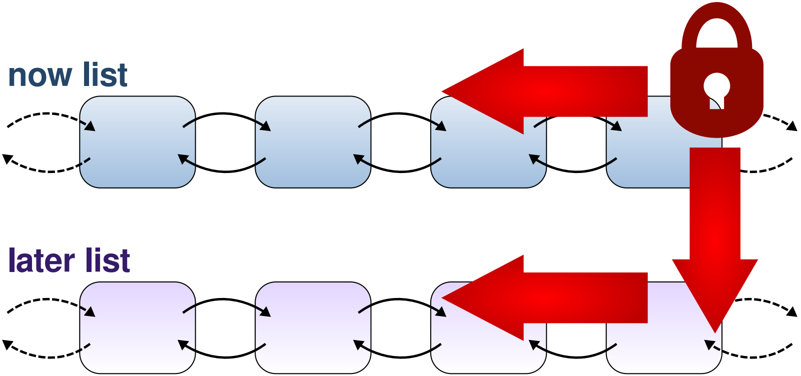
open
closed
now
later

f(A) = 0 + 9 = 9
f(B) = 4 + 7 = 11
f(E) = 3 + 10 = 13
f(F) = 5 + 6 = 11
f(C) = 5 + 7 = 12
f(G) = 7+ 5 = 12
f(I) = 7 + 6 = 13
f(D) = 8 + 5 = 13
f(K) = 12 + 0 = 12

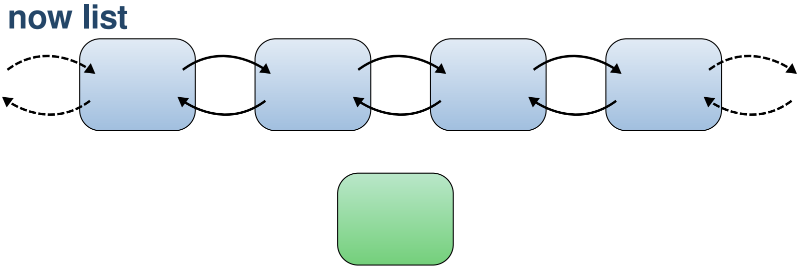
What we have done

- 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

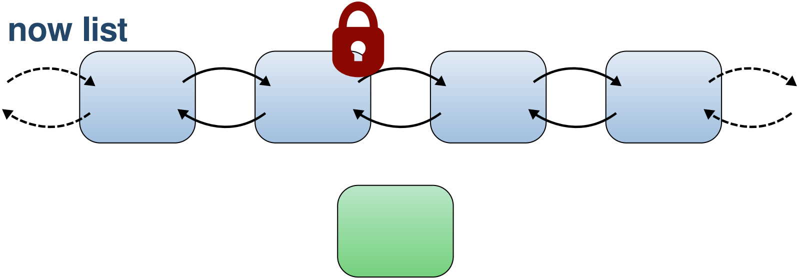
Locking concept: Deadlock prevention



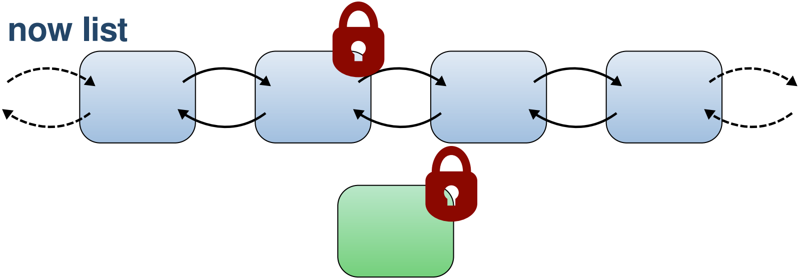
Locking concept: Insert node



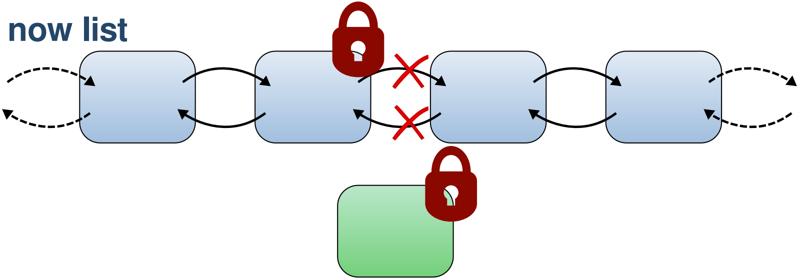
Locking concept: Insert node



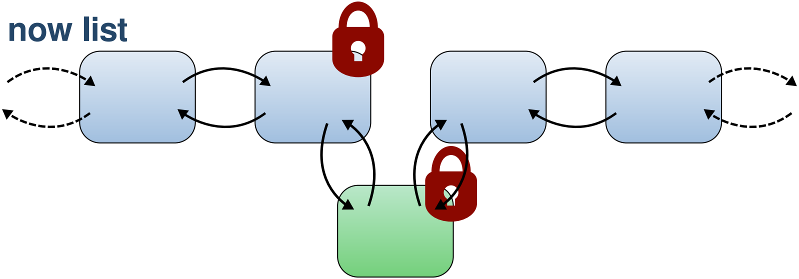
Locking concept: Insert node



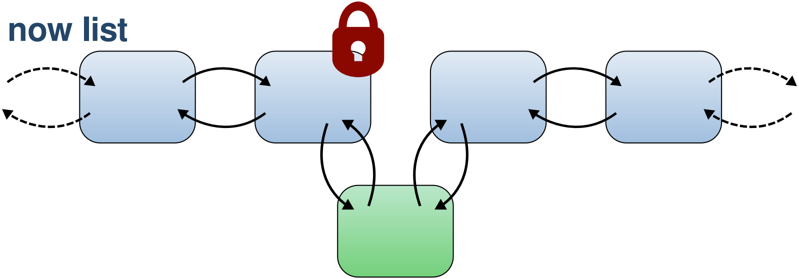
Locking concept: Insert node



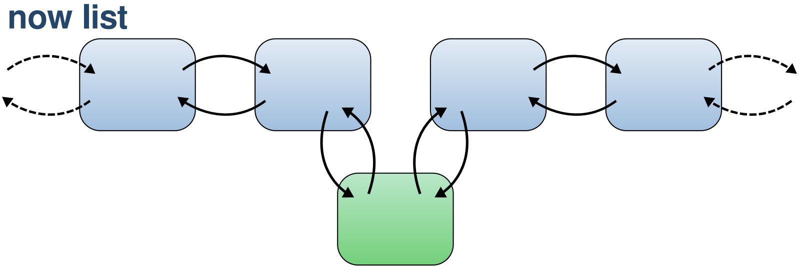
Locking concept: Insert node



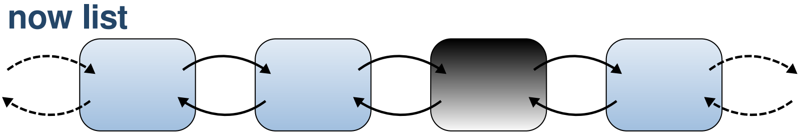
Locking concept: Insert node



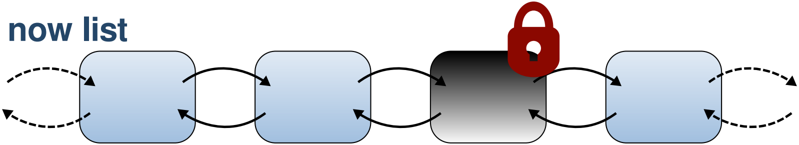
Locking concept: Insert node



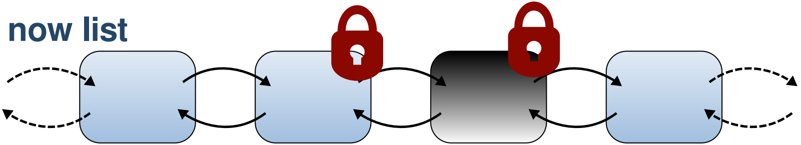
Locking concept: Remove node



Locking concept: Remove node

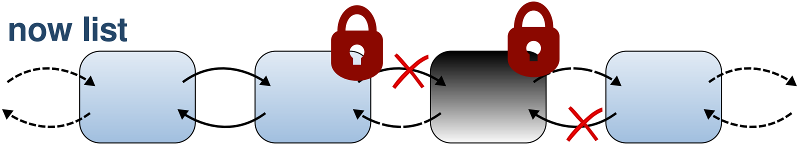


Locking concept: Remove node

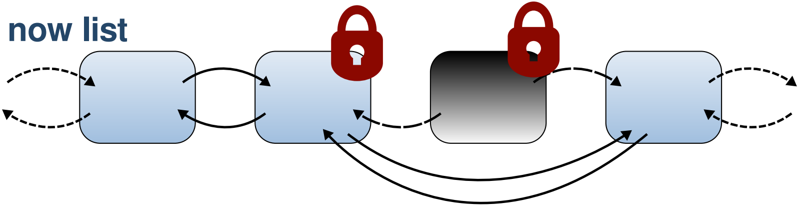


make sure they're still connected!!!

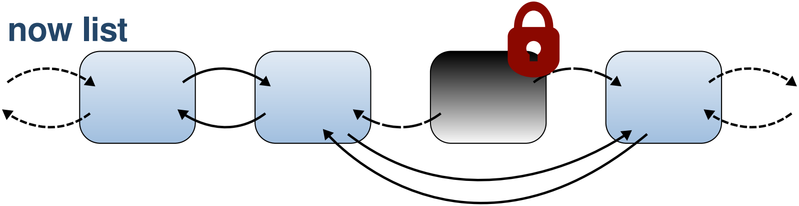
Locking concept: Remove node



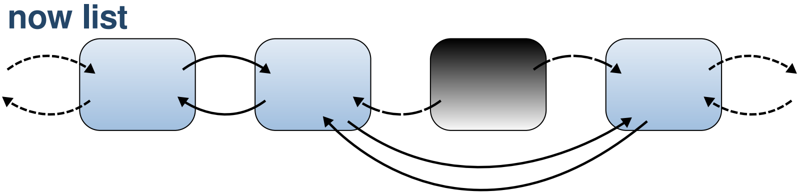
Locking concept: Remove node



Locking concept: Remove node



Locking concept: Remove node



Locking concept: 2 concepts for removing nodes

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:

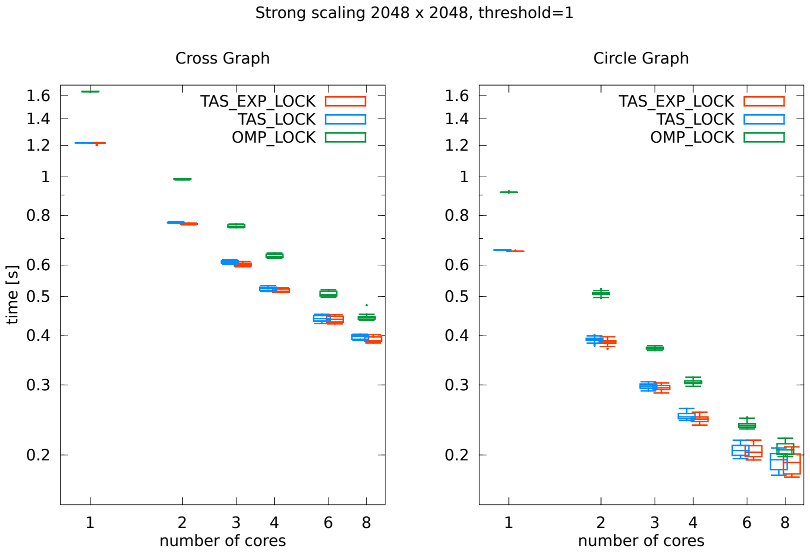
Each of the following boxplots was generated from data from **50 runs** on **1 node** of kanifushi.inf.ethz.ch.

Specifications of kanifushi.inf.ethz.ch:

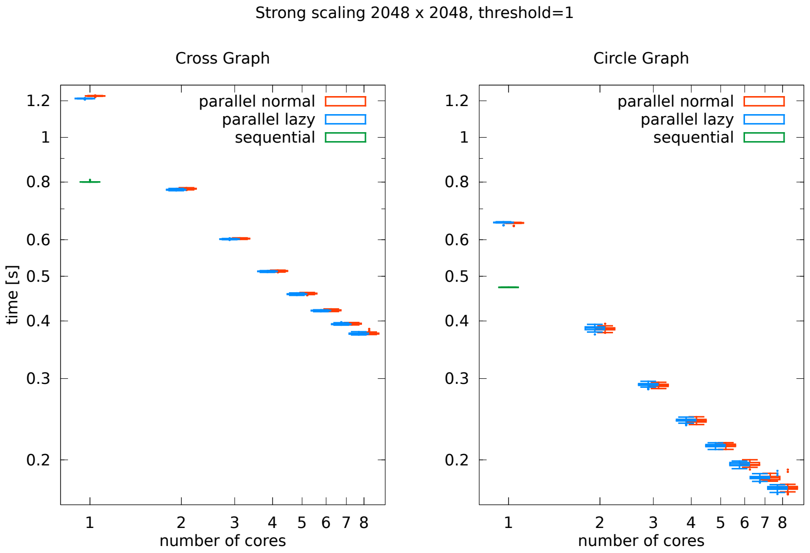
- NUMA model with 32 CPUs on 4 nodes
- 8 CPUs per node
- Intel(R) Xeon(R) CPU E7- 4830 @ 2.13GHz
- per CPU: 32KB L1 cache, 256KB L2 cache
- per node: 24MB L3 cache, 16GB memory

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

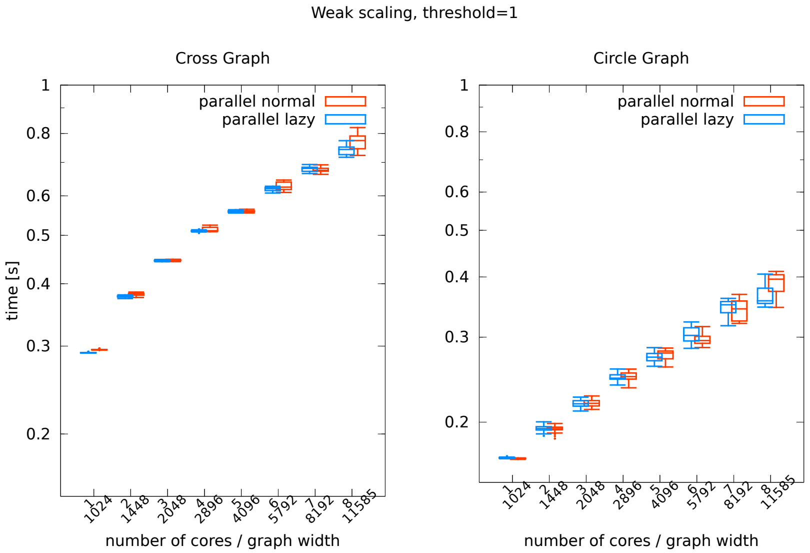
Benchmarking: Locks



Benchmarking: Strong scaling

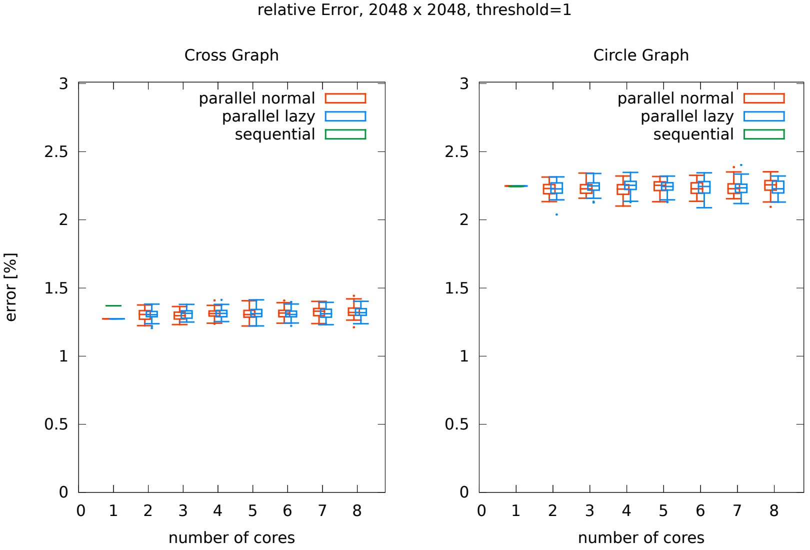


Benchmarking: Weak scaling



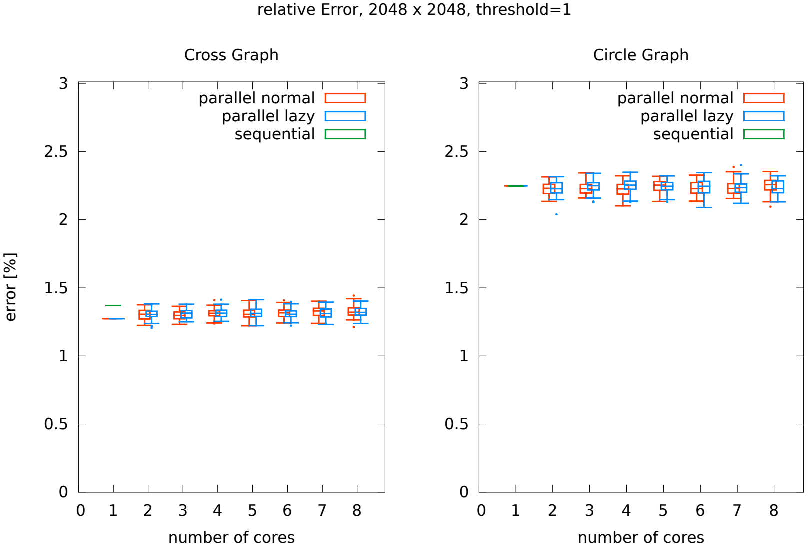
Benchmarking: Path length ↔ # cores

compared to A* from Boost Graph Library

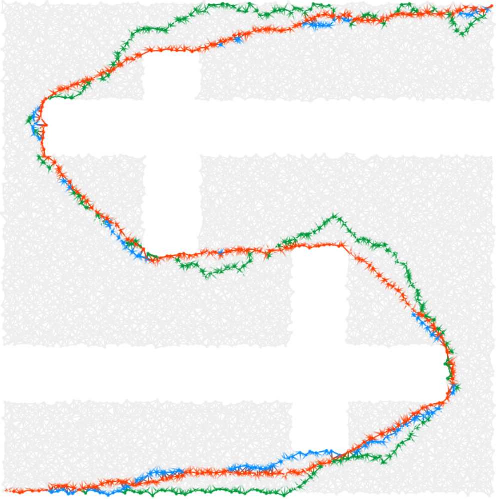


Benchmarking: Path length vs. # cores

compared to A*



Benchmarking: Path length ↔ Threshold update



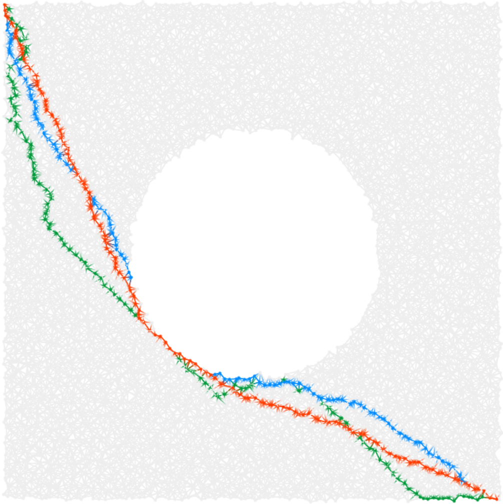
Threshold update

- threshold += **0.1**
- threshold += **1**
- threshold += **10**

Benchmarking: Path length ↔ Threshold update

Threshold update

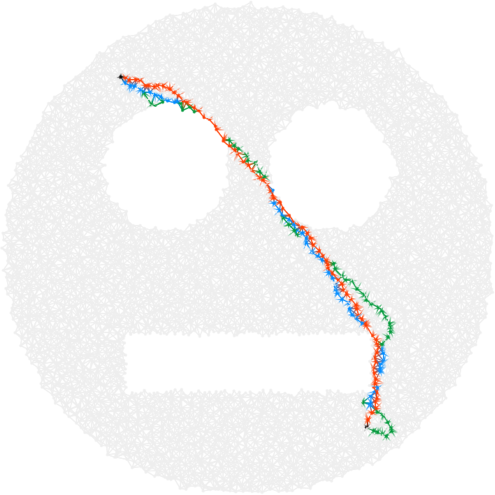
- threshold += **0.1**
- threshold += **1**
- threshold += **10**



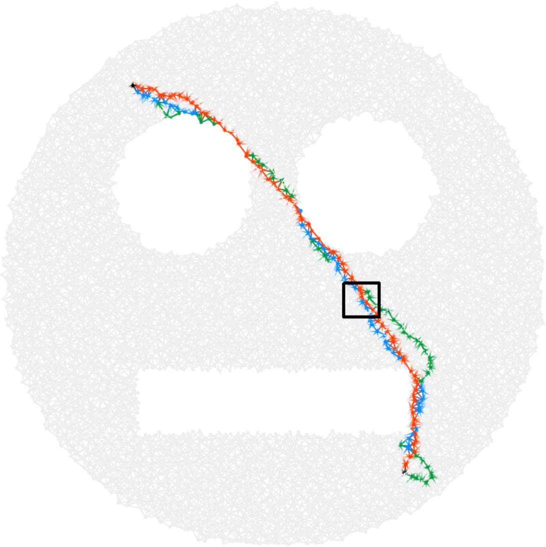
Benchmarking: Path length ↔ Threshold update

Threshold update

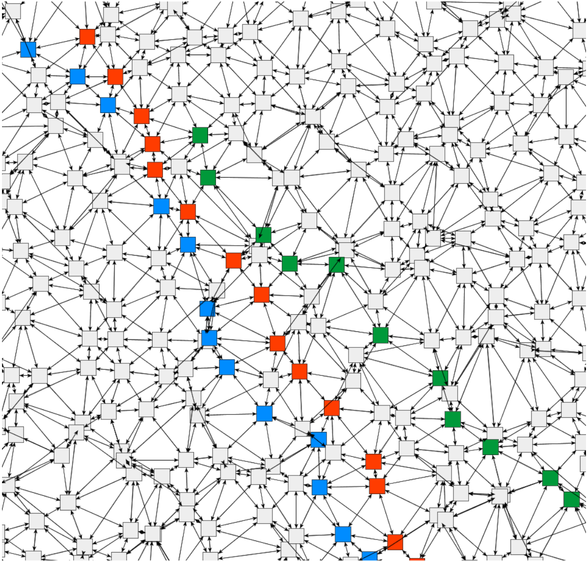
- threshold += **0.1**
- threshold += **1**
- threshold += **10**



Benchmarking: Path length ↔ Threshold update

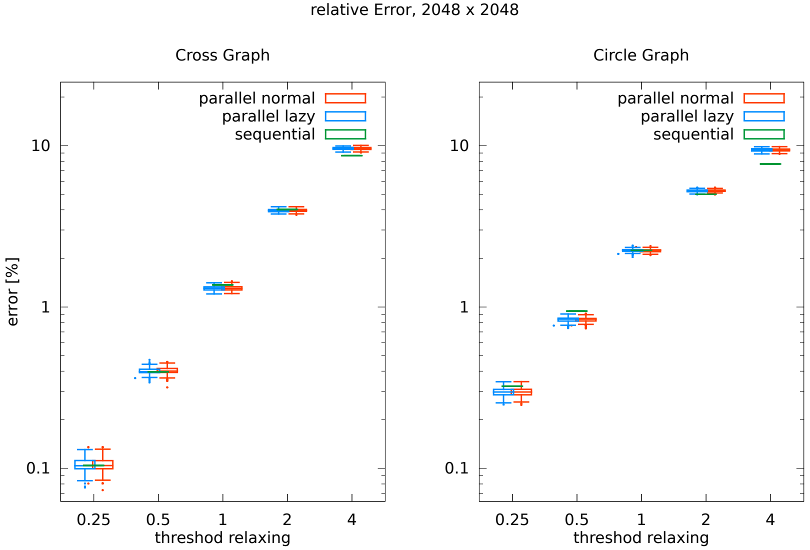


Benchmarking: Path length ↔ Threshold update

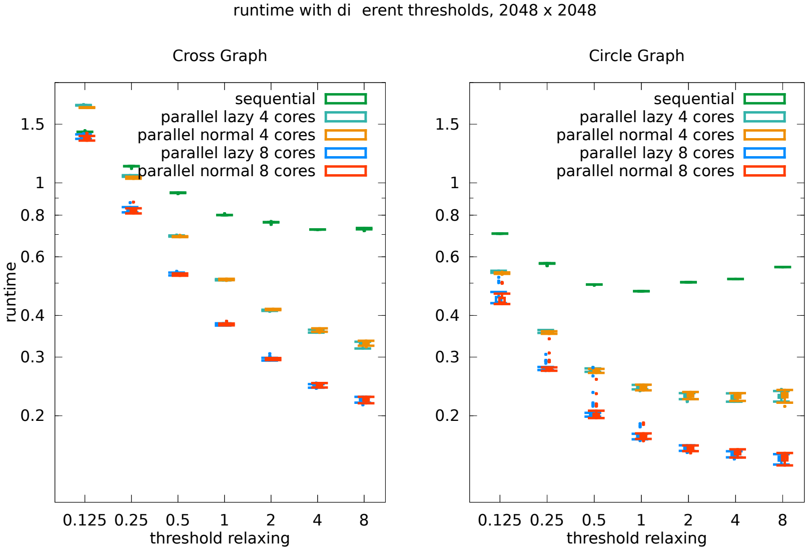


Benchmarking: Path length ↔ Threshold update

compared to A* from Boost Graph Library



Benchmarking: Run time ↔ 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 ↔ runtime trade-off can be tuned for desired result

The End