Parallel A* pathfinding algorithm

Giuseppe Accaputo
Pascal Iselin

December 16, 2013

Overview

- Goal
- Literature Review
- Approaches
- Results

Our Goals were

- Implement a correct parallel A* algorithm
- Make it faster than the serial version

Our Goals were

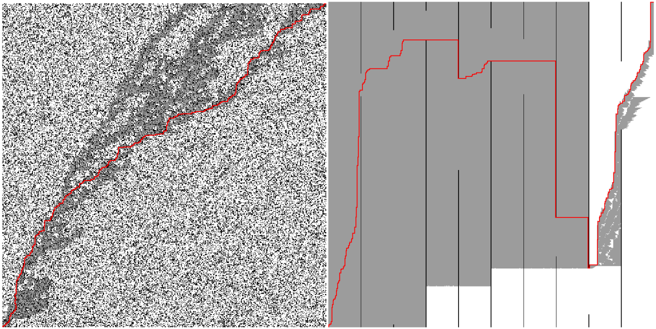
- Implement a correct parallel A* algorithm ✓
- Make it faster than the serial version

Our Goals were

- Implement a correct parallel A* algorithm ✓
- Make it faster than the serial version X

Best first heuristic search

$$f(n) = \underbrace{g(n)}_{\text{exact cost}} + \underbrace{h(n)}_{\text{estimated cost}}$$



Parallel A* in the literature

- **Shared Priority Queue**: Threads produce and consume simultaneously. Does not perform well!
  [Cohen et al., 2010]
- **Bidirectional Search**: Run two searches. Does not scale!
  [Rios, Luis Henrique Oliveira, and Luiz Chaimowicz. PNBA: A Parallel Bidirectional Heuristic Search Algorithm.]*
- **Sacrifice path quality for speed:** Converge towards other algorithms
  [Sandy Brand, and Rafael Bidarra. Multicore scalable and efficient pathfinding with Parallel Ripple Search. Comp. Anim. Virtual Worlds 23.2 (2012)]
- **Clustering**: Too complicated
  [Rafia, Inam. A Algorithm for Multicore Graphics Processors. (2010).]*

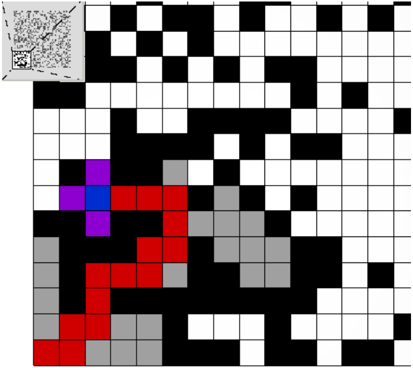
Ideas we implemented

- Concurrent Neighbor Expansion
- Shared Priority Queue
- Atomic ClosedFlags + Shared JobQueue

Same underlying datastructure for all the Implementations.
`SquareLattice` filled with Objects of Type `MapNode`

Concurrent Neighbor Expansion

Concurrent calculation of the neighbors in each step. This does not scale but it's easy!



Shared Priority Queue

- *concurrent_priority_queue* (CPQ) from Intel's Thread Building Blocks (TBB)
- CPQ does not allow rebalancing
- Recursive call of the search function with a counter
- One lock per node

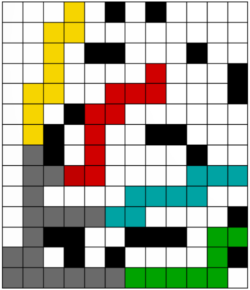
```
atomic <size_t> num_threads;
task_group t_group;

t_group.run([&]{ parallel_search(); });

void parallel_search() {
    //-----------
    // A* Magic
    //-----------
    size_t n_threads = ++num_threads;
    if(n_threads < max_threads) {
        t_group.run([&]{ parallel_search(); });
    } else {
        --num_threads;
    }
}
```

Atomic ClosedFlags + Shared JobQueue

- Run serial A* until we have enough open nodes
- Run a new A* in parallel on each of the open nodes
- Each thread has its own priority queue
- Threads communicate through a shared grid of closed flags
- Use atomic `CAS` to set the flags

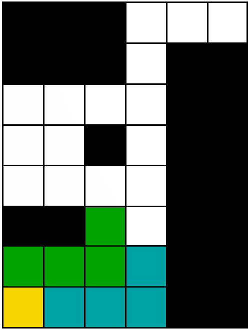


Atomic ClosedFlags + Shared JobQueue

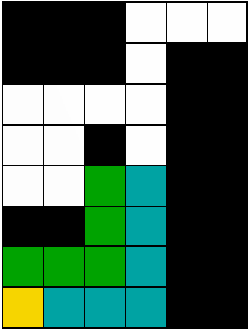
- Run serial A* until we have enough open nodes
- Run a new A* in parallel on each of the open nodes
- Each thread has it's own priority queue
- Threads communicate through a shared grid of closed flags
- Use atomic `CAS` to set the flags

- How to make sure threads don't just terminate? → Shared `JobQueue`
- How to guarantee the shortest path?

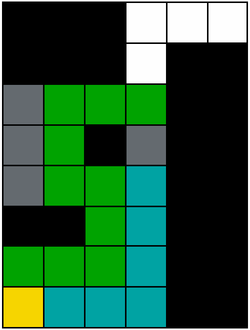
How to guarantee the shortest path?



How to guarantee the shortest path?



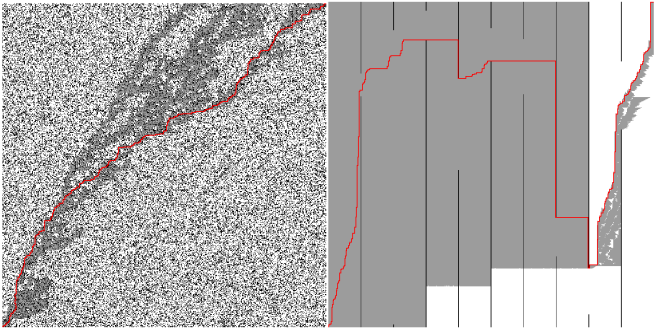
How to guarantee the shortest path?

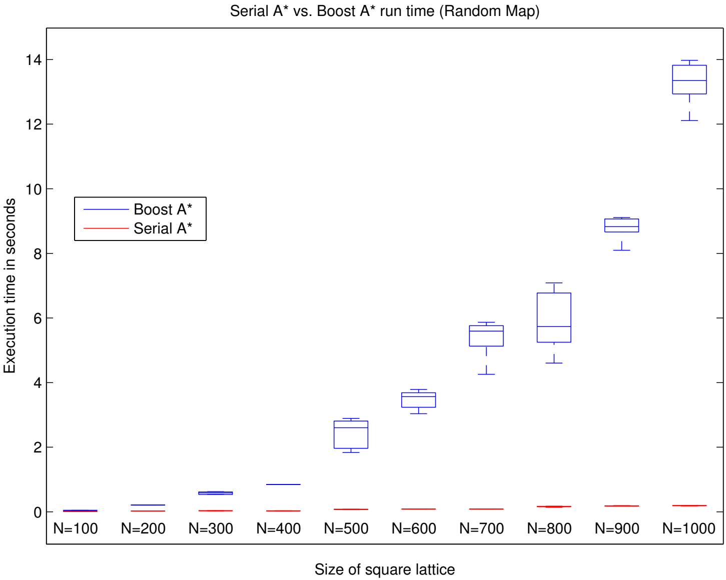


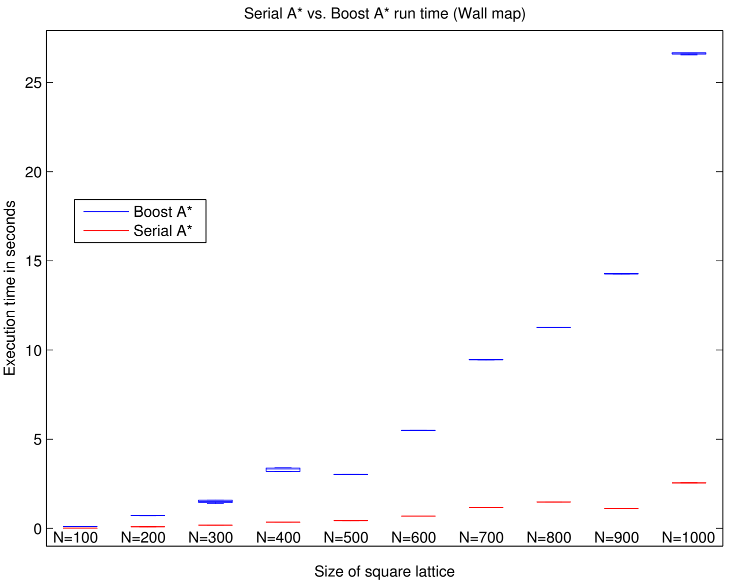
Just take the green path... NO! That would be fringe search!

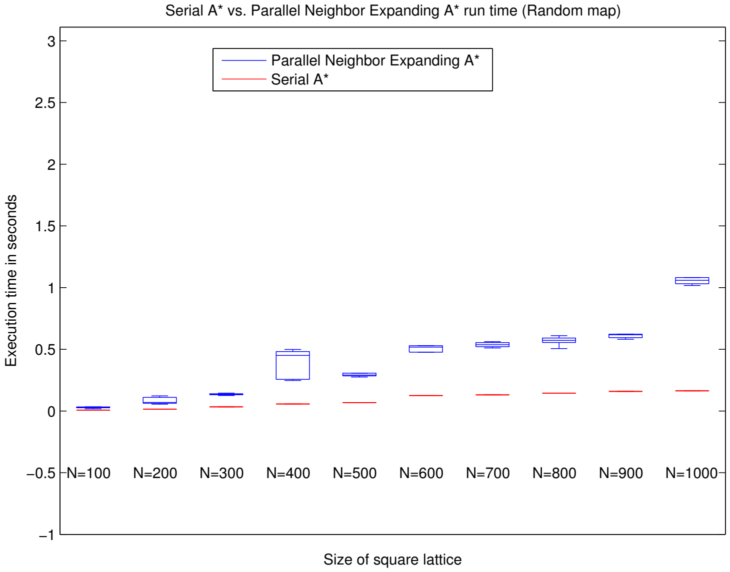
Test setup

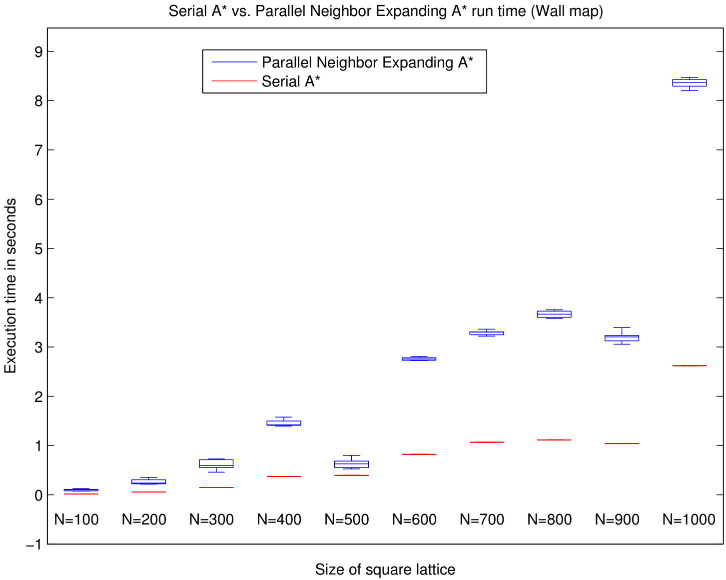
- Tested on Kanifushi
  (32 Cores Intel Xeon E7-4830 @ 2.13Ghz)
- Compiled with GCC 4.7 and TBB 3.0
- 10 runs per test case
- 90% of random map was walkable

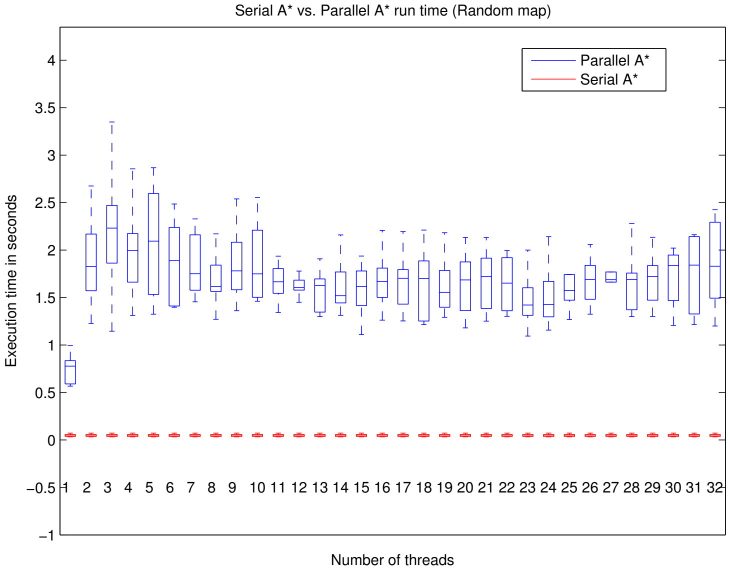


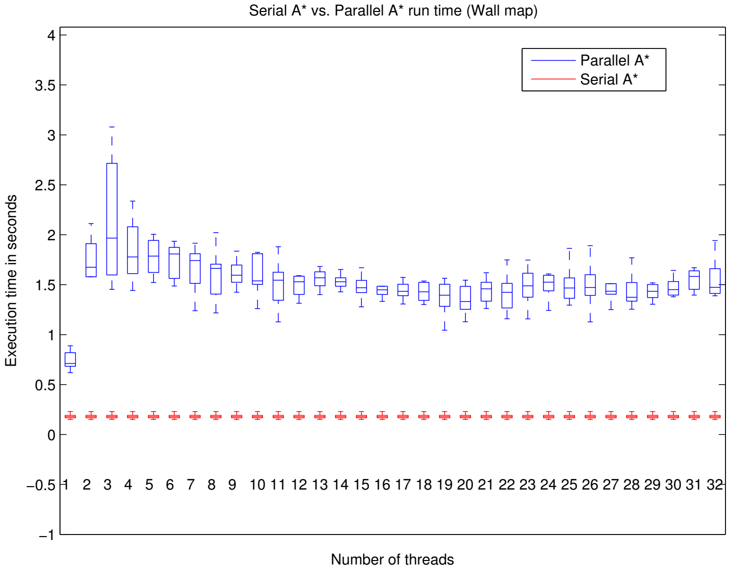

Serial A* vs. Boost A* run time (Random Map)
Execution time in seconds
0
2
4
6
8
10
12
14
Boost A*
Serial A*
N=100
N=200
N=300
N=400
N=500
N=600
N=700
N=800
N=900
N=1000
Size of square lattice

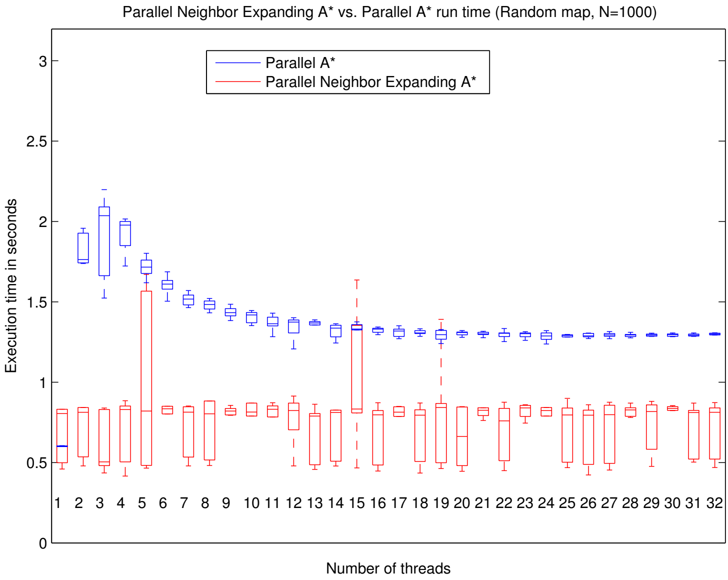

Serial A* vs. Boost A* run time (Wall map)
Boost A*
Serial A*
Execution time in seconds
0
5
10
15
20
25
N=100
N=200
N=300
N=400
N=500
N=600
N=700
N=800
N=900
N=1000
Size of square lattice

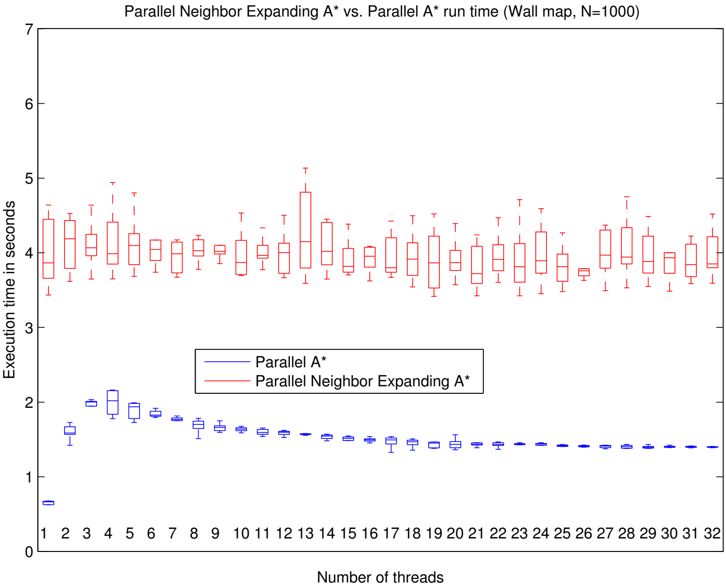

Serial A* vs. Parallel Neighbor Expanding A* run time (Random map)
Parallel Neighbor Expanding A*
Serial A*
3
2.5
2
1.5
1
0.5
0
−0.5
−1
Execution time in seconds
N=100
N=200
N=300
N=400
N=500
N=600
N=700
N=800
N=900
N=1000
Size of square lattice


Serial A* vs. Parallel Neighbor Expanding A* run time (Wall map)
Parallel Neighbor Expanding A*
Serial A*
Execution time in seconds
9
8
7
6
5
4
3
2
1
0
−1
N=100
N=200
N=300
N=400
N=500
N=600
N=700
N=800
N=900
N=1000
Size of square lattice


Serial A* vs. Parallel A* run time (Random map)
Parallel A*
Serial A*
Execution time in seconds
Number of threads
4
3.5
3
2.5
2
1.5
1
0.5
0
−0.5
−1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32


Serial A* vs. Parallel A* run time (Wall map)
Parallel A*
Serial A*
Execution time in seconds
4
3.5
3
2.5
2
1.5
1
0.5
0
−0.5
−1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Number of threads


Parallel Neighbor Expanding A* vs. Parallel A* run time (Random map, N=1000)
Parallel A*
Parallel Neighbor Expanding A*
Execution time in seconds
0
0.5
1
1.5
2
2.5
3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Number of threads


Parallel Neighbor Expanding A* vs. Parallel A* run time (Wall map, N=1000)
Execution time in seconds
0
1
2
3
4
5
6
7
Number of threads
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Parallel A*
Parallel Neighbor Expanding A*

Conclusions

- We comply with the literature!
- One must sacrifice path quality for speed
- There are much better alternatives out there:
  - Ripple Search [*Brand et al., 2012*]
  - Fringe Search