PARALLEL IMPLEMENTATION OF FRINGE SEARCH

Lukas Mosimann, Christian Zeman

ETH Zürich
Zürich, Switzerland

ABSTRACT

We've implemented and parallelized Fringe Search, a single-pair shortest path algorithm, for a shared memory environment. By already using two cores, the parallel version outperformed the sequential one on our benchmark graphs without showing any noticeable compromises in terms of path quality.

1. INTRODUCTION

This project was part of the course *Design of Parallel and High-Performance Computing* given by Torsten Hoefler and Markus Püschel in autumn 2013 at ETH Zürich.

Motivation. Pathfinding is an important problem occurring in many applications, especially in computer games and robotics. Finding the shortest path between two specific nodes is a problem in pathfinding handled by algorithms such as A* or Fringe Search. Unlike A*, Fringe Search doesn't guarantee to find the shortest path (rather a path "short" enough) but it generally outperforms A* as shown in [1].

In [2] Fringe Search has been implemented to run in parallel on a distributed memory environment. The goal of this paper was the implementation of a fast parallel version of Fringe Search for a shared memory environment.

Related work. This paper is mainly based on three papers. The Fringe Search algorithm has been introduced in [1]. S. Brand and R. Bidarra have implemented a parallel version in a distributed memory environment, published in [2] and in [3], whereas in this paper the implementation has been done for a shared memory environment.

2. BACKGROUND: SHORTEST PATH PROBLEM

In this section we formally define the single-pair shortest path problem and we consider and introduce two algorithms that are used to solve it.

Single-pair shortest path problem. A problem where the goal is to find the shortest path between a given start and an end node in a directed or undirected graph.

A*. A very popular algorithm that uses a best-first search approach for solving the single-pair shortest path problem. For the best-first search it uses a heuristic function, which estimates the distance to the end node. It always finds the shortest possible path as long as the heuristic function never overestimates the real distance. Therefore it's called optimal. A* uses a priority queue for selecting the best node and therefore each insertion into the queue has complexity $\mathcal{O}(\log n)$ with respect to the size of the queue.

Fringe Search. Another single-pair shortest path algorithm that is similar to A*, but instead of a priority queue that always has the most promising reachable node first, it stores the reachable nodes in a doubly linked list where an insertion has only complexity $\mathcal{O}(1)$ and visits a node if it's "promising enough" which is determined by a global threshold value that will be continuously increased. By doing this the path will not be optimal but it is generally faster than A*, as shown in [1]. See figure 1 for an easy example.

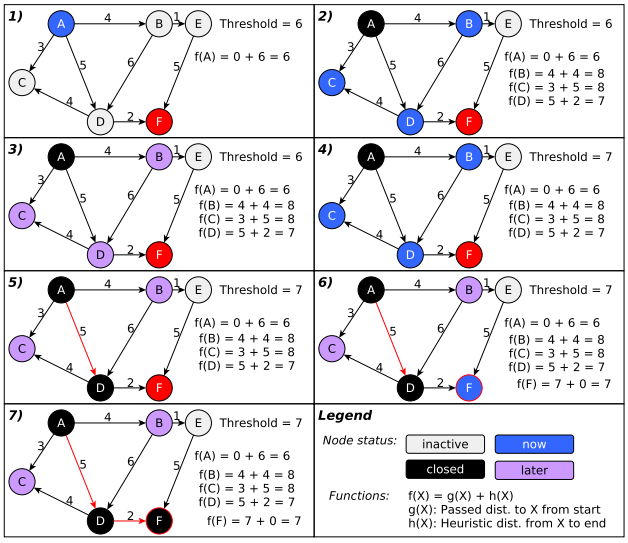

Fig. 1: Fringe Search example

3. CONCEPT AND IMPLEMENTATION

In this section we give an overview of the implementation, emphasize important aspects and illustrate the used concepts for locking.

3.1. Language and data structures

All implementations have been done with C++11, OpenMP 3 and some inline assembly for the locks.

Graph structure. We use a directed graph implemented as adjacency list. This means that each node stores a list of pointers to its neighbours as well as the distance to these nodes. Basically, this distance can be everything as long as there exists a good heuristic cost function which does not overestimate the real distance.
In our experiments we interpreted the graph as nodes in a 2D plane, mainly for visualization purposes. Every node has its position stored, so we can estimate the distance between two nodes with the Euclidean distance (on a rectangular grid one could also use the Manhattan distance).

Status of a node.

- *Inactive:* The node has not yet been visited. In the beginning each node except the start node is inactive.
- *Closed:* The node has been visited and its cost (sum of real distance form start to this node and the estimated distance from this node to the end) is below the current threshold. A closed node won't be updated anymore.
- *Now:* The node has a closed neighbor, what means that we know the real distance from the start to this node. We don't know yet if the cost of this node is below or above threshold.
- *Later:* Also a node with a closed neighbor, but the cost of this node is known and above threshold. The node won't be visited until we increase threshold or we find a better path to it.

Linked lists. We use two doubly linked lists we will henceforth call the *nowlist* and the *laterlist*. The nowlist contains the nodes that have status *now* whereas the laterlist contains the nodes with status *later*. Whenever the nowlist becomes empty we increase the threshold and swap the two lists as we also swap the definition of the status (now will become later and vice versa).

3.2. From sequential to parallel

First we implemented a strictly sequential Fringe Search in order to have a basis for the parallel implementation and also as a reference for the benchmarks regarding the speedup of the parallel version. The pseudocode for Fringe Search can be found in algorithm 1.

Parallel Fringe Search. Besides the necessary locks for the insertion and removal from the lists the threshold relaxation part (see lines 35-37 in algorithm 1) is the bottleneck because this part has to be done sequentially by only one thread and not before all threads reached this point.

Swapping the lists. Whenever we relax the threshold, the nowlist and the laterlist and the states of the included nodes will get swapped (lines 36 and 37 in algorithm 1).

```
1  add node start to nowlist ;
2  laterlist ← ∅ ;
3  threshold ← heuristicDist(start, end) ;
4  while nowlist ≠ ∅ or laterlist ≠ ∅ do
5      while nowlist ≠ ∅ do      // ∃ nodes ≤ threshold
6          x ← Node from nowlist ;
7          if x.distanceEstimation ≤ threshold then
8              if x = end then
9                  return reconstructPath() // done
10             x.status ← closed ;
11             foreach neighbour nb do
12                 if nb.status = now then
13                     calculate new distance estimation dist ;
14                     if dist < nb.distanceEstimation then
15                         nb.distanceEstimation ← dist ;
16                         nb.parent ← x;
17                         move nb right behind x in nowlist ;
18                 else if nb.status = later then
19                     calculate new distance estimation dist ;
20                     if dist < nb.distanceEstimation then
21                         nb.distanceEstimation ← dist ;
22                         nb.parent ← x;
23                         remove nb from laterlist ;
24                         insert nb right behind x in nowlist ;
25                         nb.status ← now ;
26                 else if nb.status = inactive then
27                     calculate nb.distanceEstimation ;
28                     nb.parent ← x;
29                     insert nb right behind x in nowlist ;
30                     nb.status ← now ;
31             remove x from nowlist ;
32         else
33             x.status ← later;
34             move x into laterlist ;
35     increase threshold ;
36     swap nowlist and laterlist ;
37     swap the definition of now and later ;
38 return -1 ;                  // no existing path to end
```

Algorithm 1: parallelizable Fringe Search with two lists

This means that the nowlist will become the laterlist and vice versa, as the nodes with status later are potentially below threshold after increasing it.
After swapping the lists we don't want to have all the threads starting from the first node in the nowlist but rather have them distributed over the whole nowlist. We achieve this by remembering the last node in the laterlist a thread has

worked with (e.g. we save the last node moved into the laterlist at line 34 in algorithm 1). By doing this each thread can start from this node after swapping the lists and we achieve the wanted distribution. Another advantage of this behavior is that we might profit from the cache effect because this node might still be in the cache.
But of course we have to check if this node truly is in the nowlist which can't be guaranteed (a node remembered at line 34 could be moved back at line 24). If it's no longer there we just start from the first node in the new nowlist.

3.3. Locking

Traversing the list. Whenever a thread traverses the nowlist it tries to lock every node it encounters. It does not force a lock! It just tries and if it's successful it can work with this node (visit the adjacent nodes, etc.). If the thread fails trying to lock the node it means that another thread is working with exactly this node.
In this case we don't go just to the next node in the list because the thread currently holding the lock will most likely try to lock the next node soon and like this the two threads would block each other quite often. Instead we skip a few nodes in order to achieve a nice distribution of the threads over the nowlist.
Skipping 150 nodes has proven to provide a relatively good distribution which results in a faster runtime for large graphs (more than 10^6 nodes).

Lock type. Next to the locking mechanism provided by OpenMP we've implemented the following locks

- TAS: test-and-set lock
- TAS EXP: TAS with exponential back-off

by using inline assembly.

Avoiding deadlocks. In order to avoid deadlocks we lock nodes always in the same order. This means we lock from right to left in both lists and we first lock the nodes in the nowlist and then the nodes in the laterlist (see figure 2). This requires that we know the relative location of the nodes we would like to lock (i.e. node X is left of node Y), which is always the case except for one situation:
Let's say we've locked node A in the nowlist and via this node we find a shorter path to an adjacent node B, that is already in the nowlist and below threshold. In this case we would like to update B's cost, its parent node (A will become parent node of B) and move it right behind A. For all this we require a lock on B, but we can't tell if B is left or right of A because we got to this node via a pointer in the adjacency list of A. If we now forced a lock on B we could run into a deadlock because it's possible that another thread has locked B and, because its cost is already below threshold, is waiting for the lock on A, since it's an adjacent node of B.
The solution of this problem is to not force a lock in this situation but instead, just trying to lock the adjacent node. If the try is successful we can update the path and move it, and if the try was not successful we just leave the old path that was already below threshold. This does not change the characteristic of the algorithm since the adjacent node could have already been closed by another thread since it is below threshold.
If we lock a node outside the two lists, this can be done without any problems, because no thread will ever try to lock more than one node outside the lists.
Acting like this, we prevent any possible circular wait dependencies and so no deadlocks can occur [4].

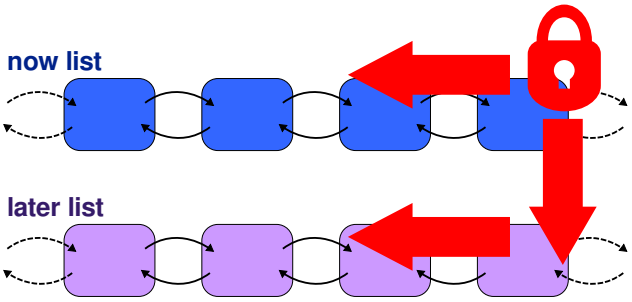

Fig. 2: Locking direction

Inserting nodes. The sequence for inserting nodes into a list is shown in figure 3.

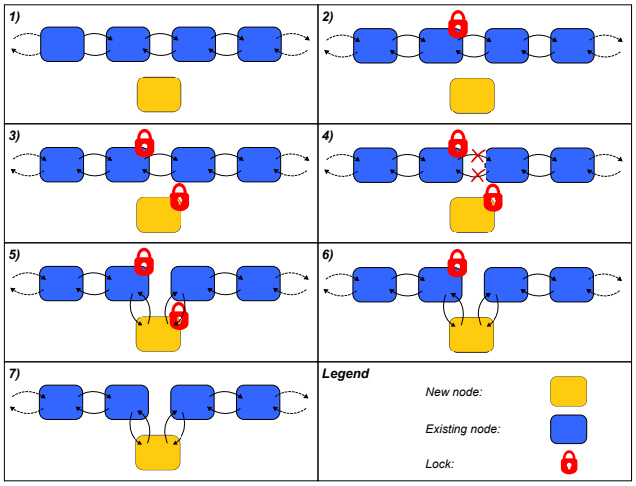

Fig. 3: Insert a node into the list

Removing nodes. The sequence for physically removing nodes from a list is shown in figure 4. We've implemented and tested the removal with two different approaches. The first approach is to physically remove the node immediately as shown in figure 4. The second one is to just mark it as removed, go on and let other threads physically remove it while they are traversing the list (lazy deletion).

Acquiring locks. All locks are acquired by "optimistic locking". This means, that if we try to get locks on nodes

with specific properties (e.g. the node must be predecessor of another node or the node must have a specific state), we spin/wait until we get the lock and after acquiring the lock we ensure the conditions are still true, and if not we release the lock immediately.

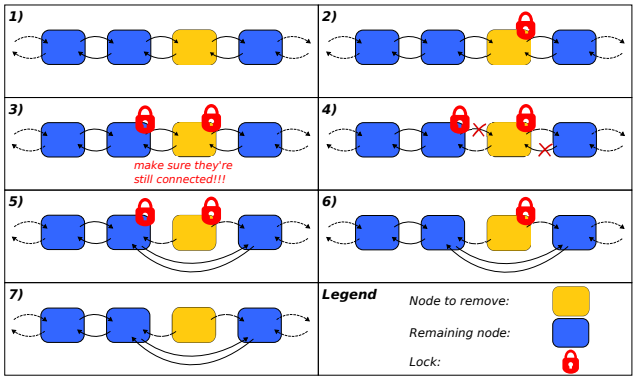

Fig. 4: Remove a node from the list. Note: The pointers of a removed node remain untouched so that if possible a traversing thread won't get lost.

4. EXPERIMENTAL RESULTS

In this section we evaluate our implementation in terms of the speed of the sequential algorithm, the performance of the different locks, strong scaling, weak scaling and several properties of the algorithm depending the threshold relaxation parameter.

4.1. Experimental setup

Hardware and compiler. The experiments have been done on 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

We've only used 1 node with 8 CPUs as the implementation has been done for a shared memory environment. To do this the application had to be executed with the *numactl* tool (e.g. *numactl --cpunodebind=0 --membind=0 fringe*). The code has been compiled with g++ v. 4.6.1 using O1 optimization. All of the following performance analysis plots are the result of 50 runs on 1 node.

Graphs used for benchmarking. Each experiment has been run on two graph types with different obstacles, the "Cross Graph" and the "Circle Graph" which are illustrated in figure 5. Both graphs have the following properties:

- based on a regular grid with distance 1
- 8 edges per node (except at the border)
- each node is displaced randomly according to a normal distribution with $\sigma = 0.3$
- start node is top left and end node is bottom right

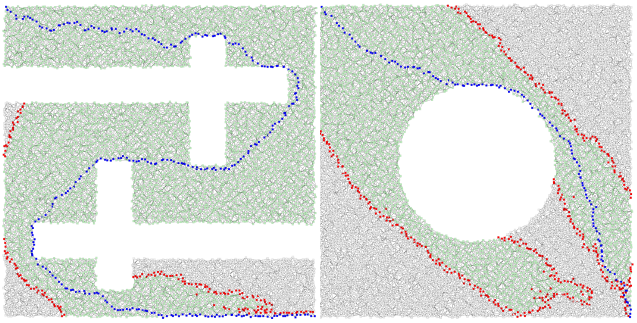

Fig. 5: The graphs used for benchmarking: "Cross Graph" (left) and "Circle Graph" (right). The blue nodes represent the found path, the green nodes are "closed" and the red nodes have status now or later when the algorithm finished.

4.2. Results

Sequential Fringe Search. In order to affirm good performance of the sequential version we tested the sequential Fringe Search against the A* search from the Boost Graph Library[1] as we couldn't find a reliable implementation of Fringe Search we could compare our code to.
Our implementation of Fringe Search proved to be much faster than A* from the Boost Graph Library (about 10 times as fast on a graph with 1024×1024 nodes with a threshold relaxation value of 1). But of course one has to take into account that Fringe Search doesn't guarantee an optimal path like A*.

Locks. The locks we've implemented with inline assembly (see section 3.3) were significantly faster than the locks provided by OpenMP (see figure 6). Therefore we used the test-and-set lock with exponential back-off for our implementation and also the following benchmarks are based on the implementation with these locks.

Strong scaling. One of the most important questions is how much faster the application gets by using more cores for a fixed problem size. The results are shown in figure 7. For both benchmark graphs the parallel version is faster than the sequential version if it uses 2 or more cores.

Weak scaling. Because of the limitations in terms of speedup according to Amdahl's law (also visible in figure 7), the next question is how the runtime behaves for an increasing problem size with keeping the problem size per core (number of nodes per core) constant.
As shown in figure 8 we still experience an increase in runtime with increasing problem size, but it's not that much

[1] http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/index.html

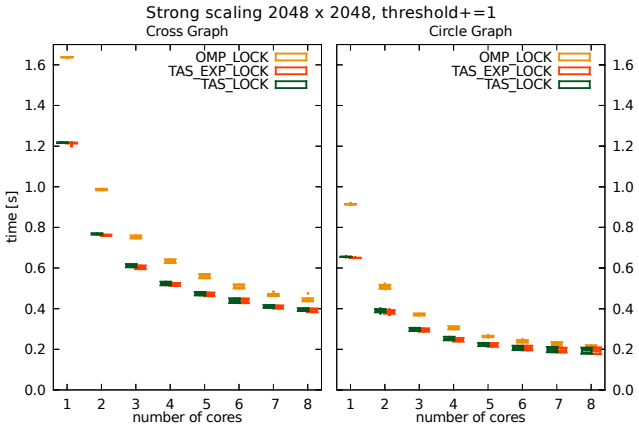

Fig. 6: Strong scaling using different locks.

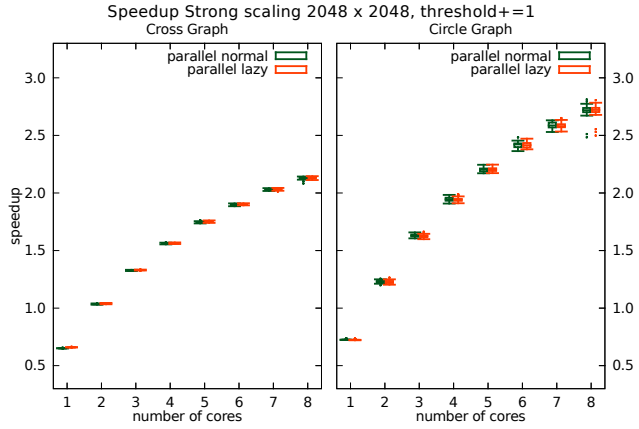

Fig. 7: Strong scaling (speedup against sequential version)

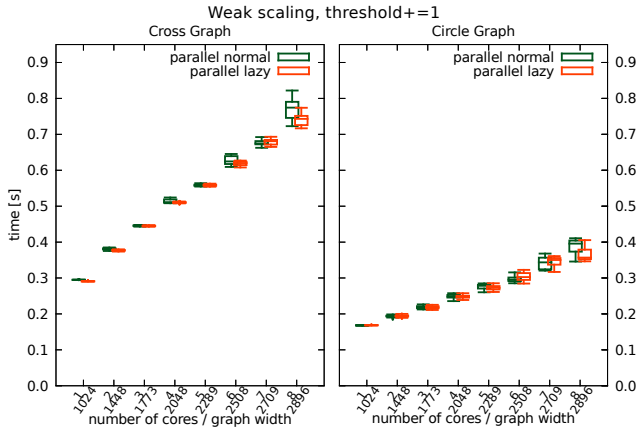

Fig. 8: Weak scaling

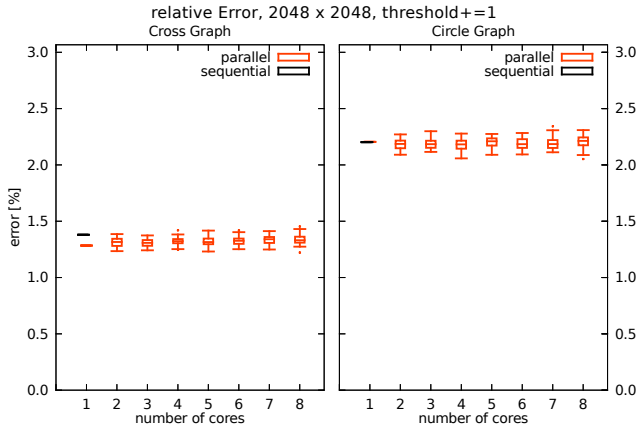

Fig. 9: Relative error in path length compared to A* depending on the number of cores

and the behavior actually meets our expectations, because with increasing problem size we also increase the time the program spends in the sequential part (longer path will lead to more threshold updates). Because the shortest path in the Circle Graph is shorter than in the Cross Graph, we need less threshold updates and so the time does not increase as much as for the Cross Graph.

Normal versus lazy removal. Looking at the figures 7 and 8 it's apparent that it doesn't matter which approach for removing the nodes we use (lazy vs. normal, see section 3.3). The difference in speed is negligible. Therefore we will only show the results for normal removal in the future plots.

Path length versus # cores. An interesting question is whether the path length gets worse the more threads we have. As mentioned in section 3.3 (avoiding deadlocks) we cannot always update a node with a better path if it's already below threshold and locked by another thread. And the more threads we have the higher is the possibility that this happens. As shown in figure 9 the number of cores/threads does not affect the quality of the path.

Path length versus threshold relaxation. The threshold relaxation value defines by how much we increase the threshold once the nowlist is empty. If we increase the threshold by a small value we will just consider few new nodes but they all are relatively promising nodes and result in a short path. If we increase the threshold by a bigger value we'll have more nodes to look at, but not all of them will be that good in terms of path length. As you can see in figure 10 the relative error gets quite high once we have a threshold relaxation value greater than 1 whereas the path quality is very good for small threshold relaxation values.

Runtime versus threshold relaxation. According to figure 10, a low threshold relaxation value leads to a short path. However, by doing this we always just have a few nodes in our nowlist, so the different threads will have more collisions and we have to increase the threshold more often, which means that we will spend more time in the sequential part of the algorithm. Therefore we expect our algorithm to be slower the smaller the threshold relaxation value is, which is also the case, as it can be seen in figure 11.

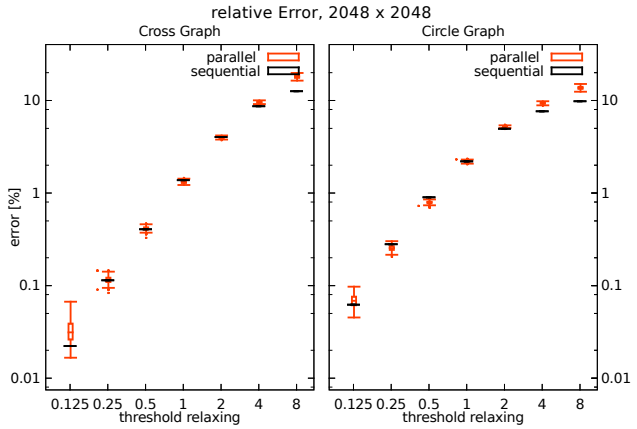

Fig. 10: Relative error in path length compared to shortest path (A*) depending on threshold relaxation value

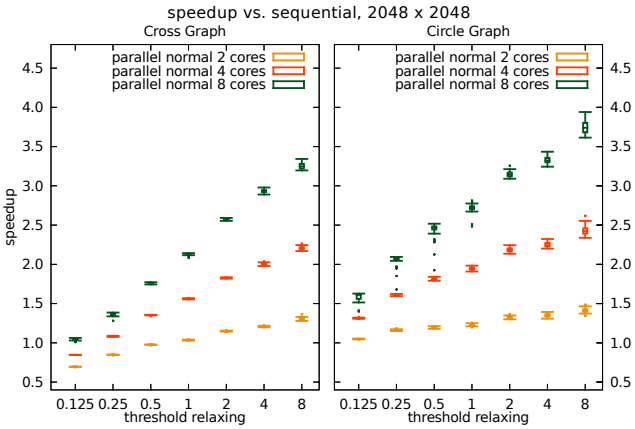

Fig. 12: Speedup of parallel version versus sequential version using different threshold relaxation values

Another interesting aspect of figure 11 is, that the sequential version gets slower for very high relaxation values. This is due to the fact, that high thresholds at the start will lead to a relatively big fan-out (compare with figure 5) that one core alone won't be able to handle in sufficient time. By running the application in parallel there are more cores that can handle these relatively big nowlist and, as shown in figure 12, the turnaround is later with more cores.

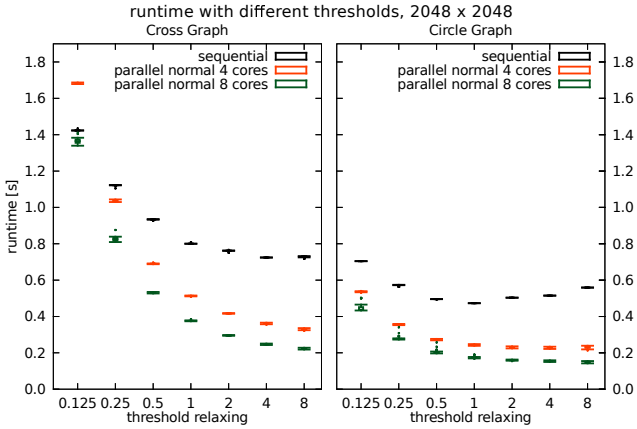

Fig. 11: Runtime depending on threshold relaxation value

5. CONCLUSIONS

We implemented a sequential version of Fringe Search that was significantly faster than the A* implementation from the Boost Graph Library. Then we parallelized it for a shared memory environment. The important results are:

- The parallel version running on only two cores was already faster than the sequential version.
- The parallel version didn't have any noticeable compromises in terms of path quality.
- Strong scaling has shown to be quite good, but of course limited by having a sequential part in the algorithm.
- Weak scaling is good but not perfect due to the increasing sequential part with increasing graph size.
- The implementation can be tuned with the threshold relaxation parameter in order to meet the individual requirements concerning path quality and runtime.

There is still a significant sequential part which is increasing with the problem size (threshold relaxation and list swapping) which can't be avoided easily, but nevertheless we believe that this is a relatively fast implementation on a shared memory environment.
Next steps could be enhancing the application to make it also work in a distributed/hybrid memory environment.

6. REFERENCES

[1] Yngvi Björnsson, Markus Enzenberger, Robert C. Holte, and Jonathan Schaeffer, "Fringe search: beating a* at pathfinding on game maps," in *In Proceedings of IEEE Symposium on Computational Intelligence and Games*, 2005, pp. 125–132.

[2] Sandy Brand, "Efficient obstacle avoidance using autonomously generated navigation meshes," M.S. thesis, Delft University of Technology, 2009.

[3] Sandy Brand and Rafael Bidarra, "Multi-core scalable and efficient pathfinding with parallel ripple search," *Computer Animation and Virtual Worlds*, vol. 23, no. 2, pp. 73–85, April 2012.

[4] A. Shoshani E. C. Coffman, Michael John Elphick, "System deadlocks," *Computing Surveys*, vol. 3, no. 2, pp. 67–78, June 1971.