PARALLEL IMPLEMENTATION OF FRINGE SEARCH

Lukas Mosimann, Christian Zeman

ETH Zürich
Zürich, Switzerland

ABSTRACT

Describe in concise words what you do, why you do it (not necessarily in this order), and the main result. The abstract has to be self-contained and readable for a person in the general area. You should write the abstract last.

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 occuring in many applications, especially in computer games and robotics. Fringe Search is a popular algorithm for single-pair shortest path problems. Unlike A* it doesn't guarantee to find the shortest path, but rather a path "short enough". The advantage of Fringe Search is that it generally outperforms A* as shown in [1].
In [2] a parallel implementation of Fringe Search has been done for a distributed memory environment. The goal of this paper was the implementation and benchmarking of a parallel version of Fringe Search for a shared memory environment.

Related work. This paper is mainly based on three papers. The Fringe Search algorithm was introduced in [1]. S. Brand and R. Bidarra implemented a parallel version in a distributed memory environment described in [3] and a bit more specific in the Master Thesis of S. Brand [2]. In this paper the implementation was 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. In our case we used a directed 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. A single-pair shortest path algorithm that is similar to A*, but instead of a priority queue that always has the most promising node first, it stores the nodes in a doubly linked list where an insertion has only complexity $\mathcal{O}(1)$ and visits a node already if it's "promising enough" which is determined by a global threshold value that will be continuously increased. The path will not be optimal but it is generally faster than A*, as shown in [1]. An example of the algorithm can be found in figure 1.

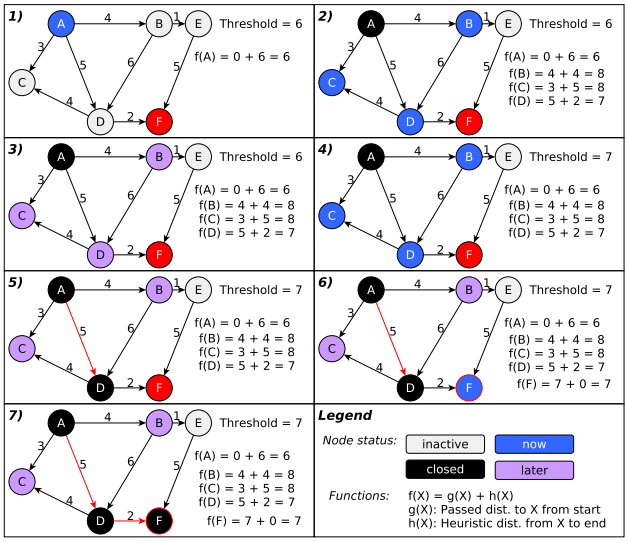

Fig. 1: Fringe Search example

3. CONCEPT AND IMPLEMENTATION

In this section we will show how the implementation has been done, emphasize important aspects and illustrate the used concepts for locking.

3.1. Language and data structures

Language. All implementations has been done with C++11 and OpenMP 3.

Graph structure. We used 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, because it is easy to visualize. Each node has its position stored, so we can estimate the distance between two nodes with the Euclidian distance or also the Manhattan Distance if we're having a quadratic grid.

Status. Each Node possesses a status. A node, which is *inactive*, has not yet been visited. In the beginning, each node except the start node is inactive. A *closed* Node has an estimated cost under the current threshold (i.e. the sum of the real cost from the start to this node and the estimated cost from this node to the end is lower than the current threshold). A node with status *now* is a node that possibly lies on the path between start and end. It has a closed neighbor, i.e. we know the real distance from the start to this node. Status *later* means, that the node also has a closed neighbor, but the cost estimation of this node currently is above threshold, so the node will not be visited again, until we increase the threshold.

Linked lists. We used two doubly linked lists we will henceforth call the *nowlist* and the *laterlist*. The nowlist generally contains the nodes that have status now whereas the laterlist contains the nodes that have 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

As a first step 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 36-38 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 37 and 38 in algorithm 1). This means the nowlist will become the laterlist and vice versa.
After increasing the threshold 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 in line 35 in algorithm 1). Like this, the probability is high that this node is in the nowlist after swapping and not

```
1  add node start to nowlist ;
2  laterlist ← ∅ ;
3  threshold ← heuristicDist(start, end) ;
4  while nowlist ≠ ∅ and 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                     calc nb.g, nb.f and set nb.parent ← x;
30                     insert nb right behind x in nowlist ;
31                     nb.status ← now ;
32             remove x from nowlist ;
33         else
34             x.status ← later;
35             move x into laterlist ;
36     increase threshold ;
37     swap nowlist and laterlist ;
38     swap the values of now and later ;
39 return -1 ;    // no existing path to end
```

Algorithm 1: parallelizable Fringe Search with two lists

all threads will start at the same node. Another advantage of this behaviour 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 is now truly in the nowlist which can't be guaranteed (a node remembered in line 35 could be moved back in line 24). If this is not the case 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 because this would slow the application down significantly (many threads would wait at the same node)! 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 as this might lead to some jam because the other thread will most likely try to lock this node soon and like this the two threads would get in each others way quite often.
So instead of just going to the next node if we encounter a locked node, 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 (larger than 10^6 nodes).

Lock type. We implemented and tried several locks. Next to the locks provided by OpenMP we implemented other locks using inline assembly (e.g. test-and-set lock).

Avoiding deadlocks. In order to avoid deadlocks we always lock in the same direction (see figure 2). If we don't know whether the second node we would like to lock is in the correct direction (e.g. left in the nowlist) we must not force a lock. Not knowing the direction can only occur if we want to move a node in the nowlist and this is only the case if we've found a shorter path to a node that already has a path that is below the previous threshold. In this case we can try locking it. 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 the old threshold. This does not change the behaviour of the algorithm since the other node was also in the nowlist and so it could have been closed by another thread before anyway.
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 (see [4]).

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

Remove nodes. The sequence for removing nodes from a list is shown in figure 4.

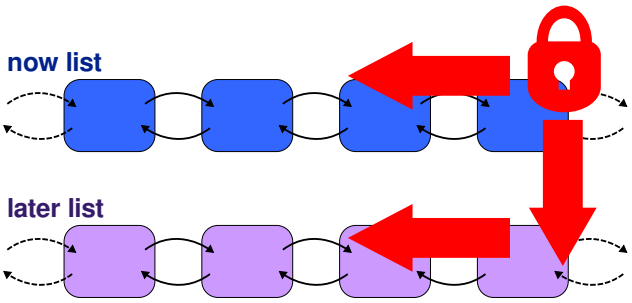

Fig. 2: Locking direction

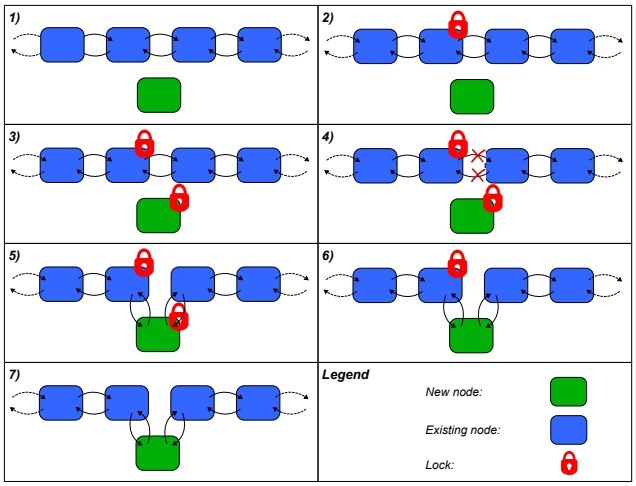

Fig. 3: Insert a node into the list

We implemented and tested two different strategies for removing the nodes. The first strategy is to physically remove the node immediately as shown in figure 4. The other strategy 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 met and if not we release the lock immediately.

4. EXPERIMENTAL RESULTS

In this section we evaluate our implementation by looking at 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.

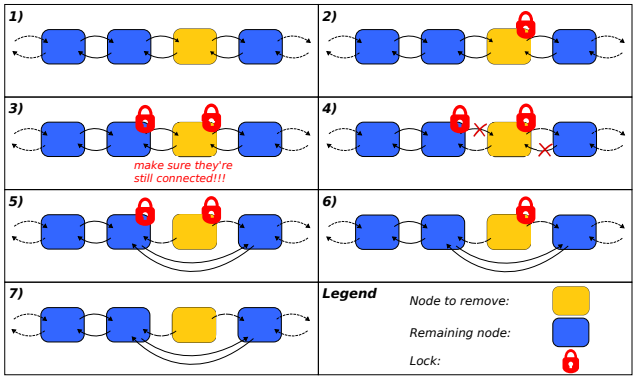

Fig. 4: Remove a node from the list. Note: The pointers of a removed node remain untouched so that a traversing thread won't get lost.

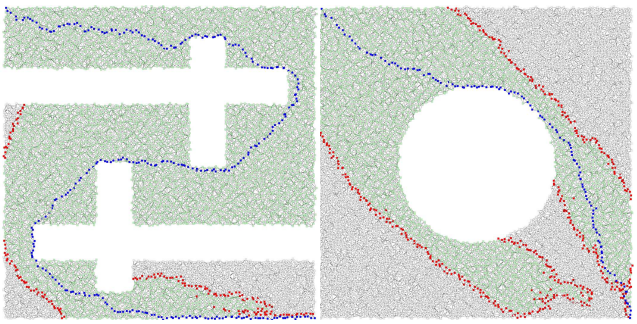

Fig. 5: The graphs used for benchmarking: "cross graph" (left) and "circle graph" (right). In this picture the blue nodes represent the found path, the green nodes are the closed nodes and the red nodes were in the nowlist or laterlist when the algorithm finished.

4.1. Experimental setup

Hardware and compiler. The experiments have been done on kanifushi.inf.ethz.ch, a computer with the following properties:

- 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

Because it's an implementation for a shared memory environment we've used only 1 node with 8 CPUs for the experiments. In order 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 different graph types with different obstacles, the "cross graph" and the "circle graph" which both are illustrated in figure 5. Both graphs have the following properties:

- based on a regular grid with distance 1 and 8 edges per node
- each node is moved randomly (normal distribution, $\sigma = 0.3$)
- start node is top left and end node is bottom right

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). But of course one has to take into account that Fringe Search doesn't guarantee an optimal path like A*.

Locks. As mentioned in section 3.3, next to the locks provided by OpenMP, we implemented some other locks using inline assembly. We implemented the following additional locks:

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

As shown in figure 6 the inline assembly locks were significantly faster than the OpenMP locks. 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.

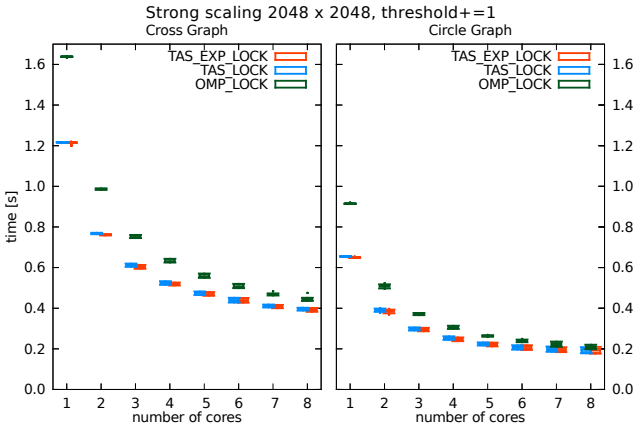

Fig. 6: Strong scaling using different locks.

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

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

7. For both benchmark graphs the parallel version is faster

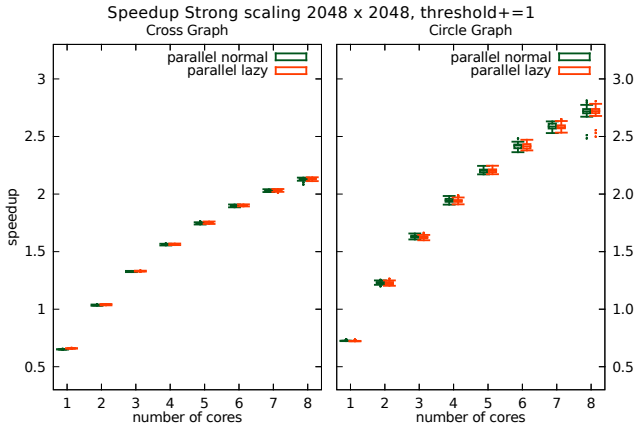

Fig. 7: Strong scaling (speedup against sequential version)

than the sequential version if it uses 2 or more cores. Using the parallel version with one core is of course slower than the sequential version because of the additional overhead and the locking.
One can also see that the more cores we use, the lower the actual benefit in terms of speedup gets. This is consistent with Amdahl's law as the sequential part of the program remains the same.
Also, it actually doesn't matter which strategy for removing the nodes we use (lazy vs. normal - see section 3.3). The difference in speed is negligible.

Weak scaling. Because of the limitations in terms of speedup according to Amdahl's law which are also visible in figure 7 the next important question is how the runtime behaves with an increasing problem size for an increasing number of cores. This means we keep 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 and certainly not as dramatic as if it would be if we didn't increase the number of cores. The behaviour 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 that much.
Also in this case the difference between normal and lazy removal is negligible. Therefore, for the future plots we will only show the results for normal removal.

Path length versus # cores. One interesting question was 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

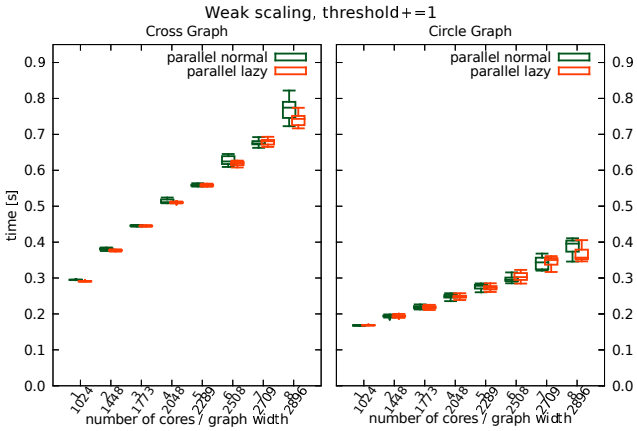

Fig. 8: Weak scaling

this happens. This behaviour is not ideal but it's still consistent with the definition of the algorithm. As shown in figure

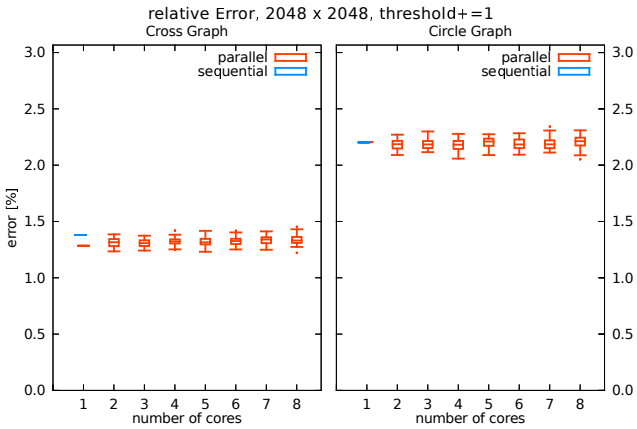

Fig. 9: Relative error in path length compared to A* depending on the number of cores

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 we don't have any more nodes below threshold left. If we increase the threshold only 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. As shown in figure 10 we can have a very good path with a low threshold relaxation value. However, by doing this we always just have a few nodes in our nowlist and we have to increase

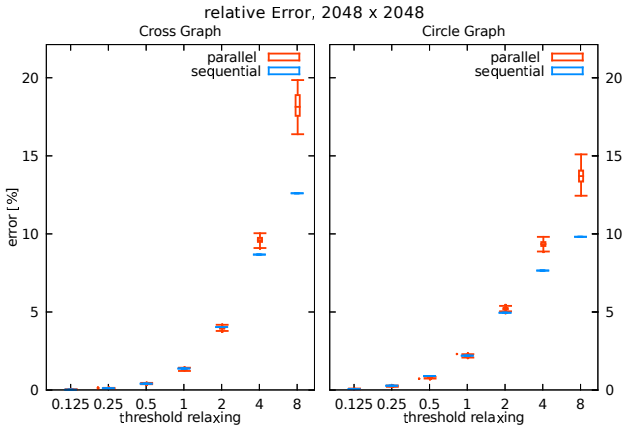

Fig. 10: Relative error in path length compared to shortest path (A*) depending on threshold relaxation value

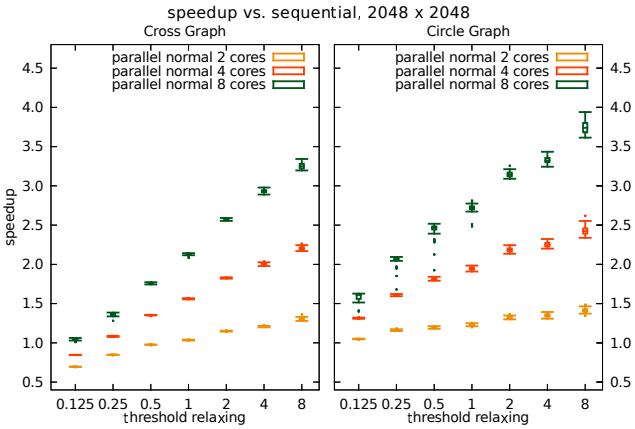

Fig. 12: Speedup of parallel version versus sequential version using different threshold relaxation values

the threshold more often, which is not good because it's 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 can be seen in figure 11. One can also see that the runtime for the sequential ver-

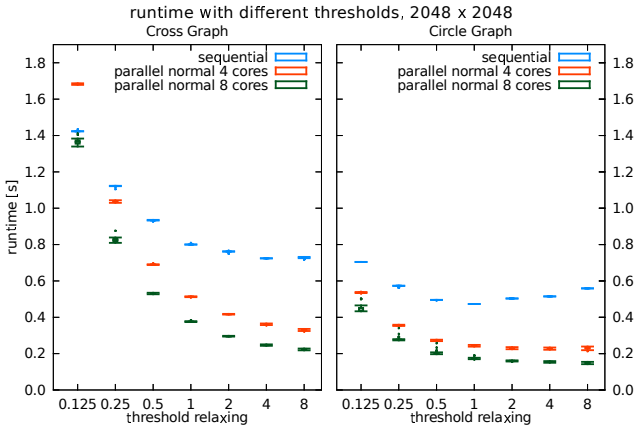

Fig. 11: Runtime depending on threshold relaxation value

sion gets slower if the relaxation value is too high. With a high value the algorithm will visit many nodes, even the not so promising ones. So we will have quite a 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 higher the threshold relaxation value is the more speedup we get with the parallel version and of course, more cores will result in a higher speedup.

5. CONCLUSIONS

We implemented a sequential version of Fringe Search that was significantly faster than the A* implementation from the Boost Graph Library. We implemented it as a parallel version 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 easily be tuned with the threshold relaxation parameter in order to meet the individual requirements concerning path quality and runtime.

We haven't found a solution to get rid of the sequential part (threshold relaxation and list swapping) and we are not sure if there exists one, but nevertheless we believe that this is a relatively fast implementation on a shared memory environment.
Possible next steps could be enhancing the application to make it also work in a distributed memory environment or in a mix of both.

6. REFERENCES

[1] Yngvi Björnsson, Markus Enzenberger, Robert C. Holte, and Jonathan Schaeffer, "Fringe search: beat-

ing 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.