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 part in many applications, especially in computer games and robotics. Fringe search is a popular algorithm for single-pair shortest path problems. Unlike A* it won'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 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 (e.g. Euclidean distance or Manhattan distance to end node). It always finds the shortest possible path and therefore it's called optimal. It uses a priority queue for the nodes and therefore each insert into the queue has complexity $O(\log n)$.

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 linked list where an insert has the complexity $O(1)$ and visits a node if it's "promising enough" which is determined by a global threshold value that will be continuously increased. It's not optimal (it doesn't guarantee to find the shortest path) but it is generally faster than A*, as shown in [1]. An example of the algorithm can be found in figure 1.

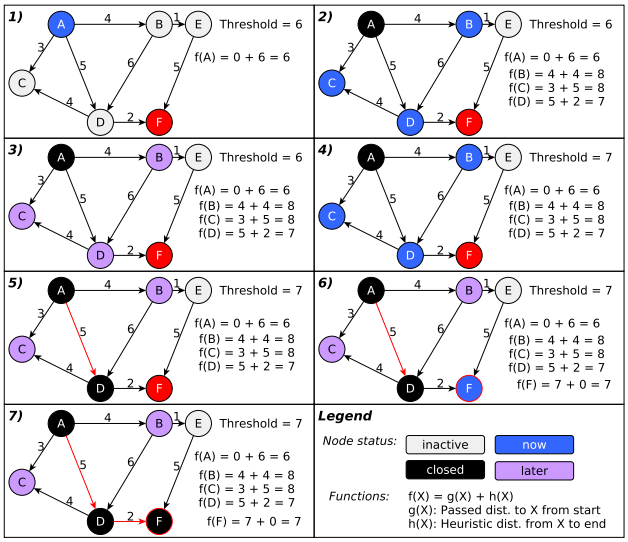

Fig. 1. Fringe Search example

3. CONCEPT AND IMPLEMENTATION

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

Language. The implementation has been done with C++11 and OpenMP 3.0.

Graph structure. We used a directed graph implemented as adjacency list. This means that each node has a list of pointers to the edges that connect it to other nodes. An edge contains a pointer to the node it's pointing to and the length of the edge.
Each node has a position in a 2D plane which is also needed for the heuristic cost function (e.g. Euclidean distance to the end node).

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 is 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).

Sequential Fringe Search. 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.

Parallel Fringe Search. Besides the necessary locks for the insertion and removal from the lists the threshold relaxation part (see line 32 in algorithm 1) is the bottleneck because this part has to be done sequentially by only one thread. The pseudocode for the parallel Fringe Search can be found in algorithm 1.

Swapping the lists. Whenever we relax the threshold the nowlist and the laterlist will get swapped (line 33 in algorithm 1). This means the nowlist will become the laterlist and vice versa. Also the node states will get swapped (now will become later and vice versa).
This part has to be done sequentially. After this we can run in parallel again but we don't want to have all the threads starting from the $zero$ node again but rather have them distributed over the new nowlist. We somehow achieve this by remembering the last node in the laterlist a thread has worked with (see line 30 in algorithm 1). Like this, the probability is high that this node is now in the nowlist and not 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 really in the nowlist which can't be guaranteed. If this is not the case we just start from the $zero$ node which is the only node we can be sure that it's still in the nowlist (see lines 34 and 35 in algorithm 1).

Locks: 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

```
1  add nodes zero and start to nowlist ;
2  add node zero and laterlist ;
3  threshold ← heuristicDist(start, end) ;
   // start parallel part
4  x ← start ;
   // barrier:  wait for all threads
5  while nowlist ≠ ∅ do      // any nodes left
6    while nowlist ≠ ∅ do   // nodes ≤ thresh
7      if x.f ≤ threshold then
8        if x == end then
9          return reconstructPath() ;    // done
10       x.status ← closed ;
11       for each neighbour nb do
12         if nb.status == now then
13           calculate new distance ;
14           if dist < nb.g then
15             update nb.f, nb.g and nb.parent ;
16             move nb right behind x in nowlist ;
17         else if nb.status == later then
18           calculate new distance ;
19           if dist < nb.g then
20             update nb.f, nb.g and nb.parent ;
21             remove nb from laterlist ;
22             insert nb right behind x in nowlist ;
23             nb.status ← now ;
24         else if nb.status == inactive then
25           calc nb.g, nb.f and set nb.parent ← x;
26           insert nb right behind x in nowlist ;
27           nb.status ← now ;
28       remove x from nowlist ;
29     else
30       x.status ← later and lastlater ← x ;
31       x ← x.next move lastlater from nowlist to
           laterlist ;
     // sequential part
32   increase threshold ;
33   swap nowlist ↔ laterlist and now ↔ later ;
     // run again in parallel
34   if lastlater still in nowlist then x ← lastlater ;
35   else x ← zero ; // zero from nowlist
36 return -1 ;    // no existing path to end
```

Algorithm 1: Parallel Fringe Search. Note: The locking mechanism is not part of this pseudocode, but it's thoroughly described in several paragraphs in section 3 and in figures 2, 3 and 4).

working with exactly this node.
In this case we don't go just to the next node in the list because this might lead to some jam because the other thread will probably 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.

Locks: Implementation. 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).

Locks: 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. This 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 threshold. In this case we can try locking it. If the try is successful we can update the path and move it and if not we just leave the old path that was already below threshold. This does not change the behaviour of the algorithm since the node was below threshold anyway (actually this node easily could have been closed by another thread before we'd found the better path), but acting like this we ensure that no deadlocks can happen.

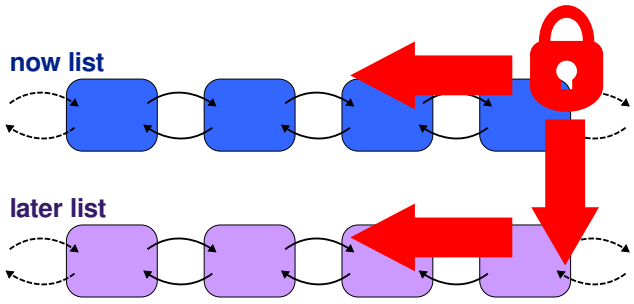

Fig. 2. Locking direction

Locks: Insert nodes. The sequence for inserting nodes is shown in figure 3.

Locks: Remove nodes. The sequence for removing nodes is shown in figure 4.
We implemented and tested two different strategies for removing the nodes. The first strategy is to remove the node immediately as shown in figure 4 when it has to be removed. The other strategy is to just mark it as removed, go on and let other threads actually remove it while they are traversing the list.

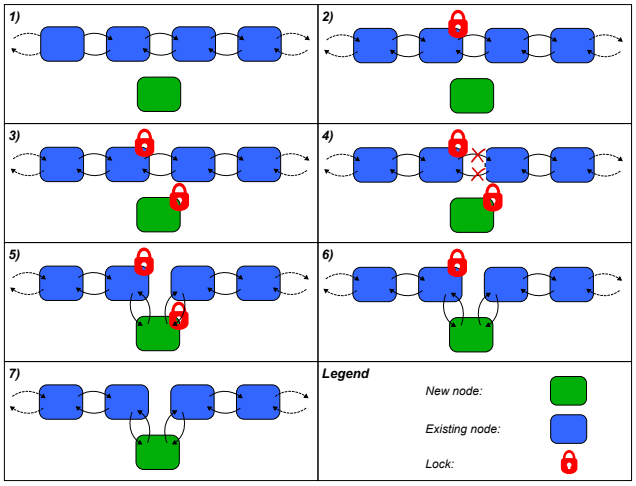

Fig. 3. Insert a node into the list

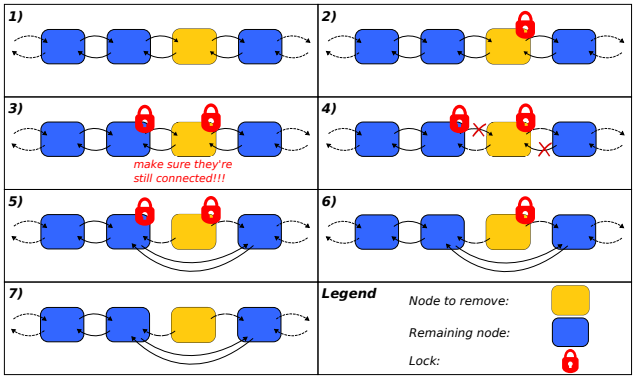

Fig. 4. Remove a node from the list. Note: The pointers of the removed node remain so that a traversing thread won't get lost (traversing is without locks).

4. EXPERIMENTAL RESULTS

Here you evaluate your work using experiments. You start again with a very short summary of the section. The typical structure follows.

Experimental setup. Specify the platform (processor, frequency, maybe OS, maybe cache sizes) as well as the compiler, version, and flags used. If your work is about performance, I strongly recommend that you play with optimization flags and consider also icc for additional potential speedup.

Then explain what kind of benchmarks you ran. The idea is to give enough information so the experiments are reproducible by somebody else on his or her code. For sorting you would talk about the input sizes. For a tool that performs NUMA optimization, you would specify the programs you ran.

Results. Next divide the experiments into classes, one paragraph for each. In each class of experiments you typically pursue one questions that then is answered by a suitable plot or plots. For example, first you may want to investigate the performance behaviour with changing input size, then how your code compares to external benchmarks.

For some tips on benchmarking including how to create a decent viewgraph see pages 22–27 in [4].

Comments:

- Create very readable, attractive plots (do 1 column, not 2 column plots for this report) with readable font size. However, the font size should also not be too large; typically it is smaller than the text font size. An example is in Fig. **??** (of course you can have a different style).
- Every plot answers a question. You state this question and extract the answer from the plot in its discussion.
- Every plot should be referenced and discussed.

5. CONCLUSIONS

Here you need to summarize what you did and why this is important. *Do not take the abstract* and put it in the past tense. Remember, now the reader has (hopefully) read the report, so it is a very different situation from the abstract. Try to highlight important results and say the things you really want to get across such as high-level statements (e.g., we believe that .... is the right approach to .... Even though we only considered x, the .... technique should be applicable ....) You can also formulate next steps if you want. Be brief. After the conclusions there are only the references.

6. FURTHER COMMENTS

Here we provide some further tips.

Further general guidelines.

- For short papers, to save space, I use paragraph titles instead of subsections, as shown in the introduction.
- It is generally a good idea to break sections into such smaller units for readability and since it helps you to (visually) structure the story.
- The above section titles should be adapted to more precisely reflect what you do.
- Each section should be started with a very short summary of what the reader can expect in this section. Nothing more awkward as when the story starts and one does not know what the direction is or the goal.
- Make sure you define every acronym you use, no matter how convinced you are the reader knows it.
- Always spell-check before you submit (to us in this case).
- Be picky. When writing a paper you should always strive for very high quality. Many people may read it and the quality makes a big difference. In this class, the quality is part of the grade.
- Books helping you to write better: [5] and [6].
- Conversion to pdf (latex users only):

  dvips -o conference.ps -t letter -Ppdf -G0 conference.dvi

  and then

  ps2pdf conference.ps

Graphics. For plots that are not images *never* generate the bitmap formats jpeg, gif, bmp, tif. Use eps, which means encapsulate postscript. It is scalable since it is a vector graphic description of your graph. E.g., from Matlab, you can export to eps.

The format pdf is also fine for plots (you need pdflatex then), but only if the plot was never before in the format jpeg, gif, bmp, tif.

7. 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] M. Püschel, "Benchmarking comments," online: http://people.inf.ethz.ch/markusp/teaching/263-2300-ETH-spring11/slides/class05.pdf.

[5] N.J. Higham, *Handbook of Writing for Mathematical Sciences*, SIAM, 1998.

[6] W. Strunk Jr. and E.B. White, *Elements of Style*, Longman, 4th edition, 2000.