Parallel A* Pathfinding

Module Code: CMP202 2023/24 Term 2
Mini-Project: Mini-Project1, CPU
Student Name: Finn Else-McCormick
Student ID: 2200864

Table of Contents

Introduction

The core of this project is an implementation of the A* search algorithm for arbitrary graphs, attempting to use concurrency to speed up its execution. To support this I also implemented a GUI with ImGui for visualising this pathfinding on 2D graphs and some basic k-nearest neighbour graph generation.

Implementation

The A* algorithm involves finding two scores for a node in a graph - the 'cost' from the origin node (the sum of the weights along all the edges taken to traverse to the node) and an estimate of the cost to the end node, represented by a heuristic function.

The sequential implementation of A* used in this project uses an open set represented by a priority queue, with a node's 'total cost' (cost from origin + heuristic) being used to determine its priority.
To begin with, the open set contains only the origin. Looping until the goal is found, the top node is popped from the open set and we attempt to traverse each of its neighbours, adding the edge weight to the current node's cost to find that neighbour's cost along the current path. If it's better than the currently stored cost for that neighbour (which starts as the maximum value of the weight type), we update the neighbour's cost, set its 'parent index' to the current node, and push it to the open set.
Once the goal is found, we can unroll the path by following the chain of parent indices back from the goal to the origin.

For the parallel implementation, I decided to implement Hash-Distributed A* *(Weinstock & Holladay, 2016)*. In HDA*, each thread has its own open set and nodes are mapped to threads using a hash function; as this mapping is static, no complicated task management needs to take up processing time - when a node is traversed, it is simply pushed to the open set of its corresponding thread.
The cost values and parent indices are still represented with simple vectors, but for thread safety they are now vectors of 'protected' objects, using a simple templated wrapper class which stores a value and offers getter and setter methods which internally lock a mutex.
The open sets are now represented by a mutex wrapper around a hash set.
Hash sets are used here instead of priority queues as sets in C++ share the strict weak ordering of priority queues, but allow access to elements other that the top, which is necessary as the ordering is dependent on the node's cost, so if the same node is traversed multiple times the old version needs to be erased before its cost can be updated and it can be re-added. While this results in many mutexes being used, it minimises the amount of computation that any one mutex needs to lock for, preventing race conditions.

When a thread's open set becomes empty, it hits a barrier. The barrier function loops over all the threads' open sets to check if any are not empty. If they all are, it sets a flag to say that all work is complete, and the thread functions can escape their internal loop and end processing.
While this results in some degree of waiting, it is necessary as there is no way for a thread in isolation to know whether processing is finished, as it doesn't know whether new nodes are going to come in from other threads.

Profiling

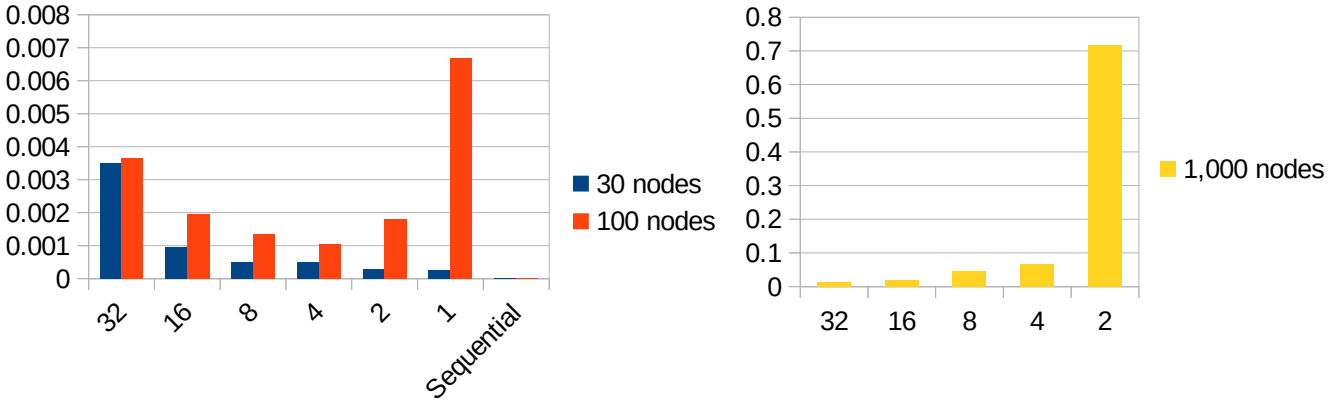
The algorithms are profiled by running and timing them repeatedly. The most accurate way to do this is the blocking version, in which the profiling is done directly on the main thread, but this freezes the GUI so I also provide a non-blocking version.

The non-blocking version is launched from a thread function, with the current state (in-progress or finished) being communicated to the main thread using a binary semaphore. The GUI also reads back information about how many operations have been completed, but this doesn't need to be protected with a mutex as the GUI doesn't modify it and it doesn't matter if the displayed value is slightly out of date.

Performance Evaluation

CPU: AMD Ryzen 7 2700 Eight-Core Processor

Number of Threads	30 nodes	100 nodes	1,000 nodes
32	3.51000E-03	3.66000E-03	1.24000E-02
16	9.43000E-04	1.96000E-03	1.80348E-02
8	4.84000E-04	1.36000E-03	4.47857E-02
4	4.84000E-04	1.05000E-03	6.58084E-02
2	3.00000E-04	1.80000E-03	7.16721E-01
1	2.58000E-04	6.69000E-03	5.35021E+00
Sequential	3.10000E-06	6.20000E-06	5.17000E-05



Looking just at the HDA* timings, we can see that it performs best with a middling number of threads. Very low numbers of threads always greatly slow it down, as it is taking all the overhead of the parallelisation without any of the gains. High numbers of threads slow it down at small graph sizes, as it leads to more time spent waiting for threads to hit the barrier, while at large graph sizes this goes away as threads are kept busier.

Overall, however, my HDA* implementation is evidently a failure. While it doesn't perform actively badly, even its best performance is an order of magnitude worse than the basic sequential implementation. In retrospect, this is because it's a poor choice of problem to parallelise; all its constituent tasks heavily depend on the outcome of the tasks before them, limiting the possible performance increase from concurrency, and the large amounts of synchronisation necessary to

protect the data involved and allow the algorithm to know it should terminate lead to any potential performance increase being buried in the synchronisation overhead.
In practice in the real world, it would almost always be a better idea to keep the pathfinding sequential and simply use a task farm to perform the pathfinding operations concurrently with each other, since they would have no data dependencies between each other.

Citations

Weinstock, A. and Holladay, R. (2016) _Parallel A* Graph Search_. dissertation. Available at: http://people.csail.mit.edu/rholladay/docs/parallel_search_report.pdf (Accessed: 2024).