Tree Search Parallelization with Parallel Task

A* for the Travelling Salesman Problem

1 Introduction

One way to solve combinatorial problems such as the travelling salesman problem (TSP) is to represent the solution space as a very large tree. Searching this tree using A*, a best first search algorithm, can be used to find the optimal solution. However, the travelling salesman problem is an NP-hard problem, so as the number of cities increases the time taken to find the solution increases exponentially. Although parallelization cannot reduce this time complexity, it allows us to solve larger problems more quickly, which has various applications such as in logistics, planning, microchip design, and even in DNA sequencing. There have been attempts to parallelize the best first search algorithm, with mixed performance and results. This research project attempts to parallelize the A* search algorithm using Parallel Task, in order to solve the travelling salesman problem faster.

2 Project Brief

The first stage of the project involves coming up with an A* best first search algorithm to find the optimal solution to the TSP by searching through the solution tree. To do this, it is necessary to come up with a consistent heuristic that underestimates the best possible length of the final solution path. The second stage is to parallelize this algorithm to get a significant speedup. The algorithm should be adjusted in various ways to get better performance. Finally, the performance is evaluated across different sized problems and with various numbers of processors.

3 Literature Review

In this project, research was mainly carried out on two areas; how the A* algorithm can be applied to the TSP, and how the A* algorithm can be parallelized. The information that was gathered was combined together to get a comprehensive understanding of the task. A project description from Arizona State University [1] explains how the TSP can be transformed into a search through a tree of paths, using a minimum spanning tree heuristic. This is important as it highlights the differences between using A* traditionally for path finding compared to using it to search through the tree of paths for combinatorial problems. A study on parallel heuristic search by Kumar, Ramesh, and Rao [2] details how this can be parallelised using both centralised and distributed approaches. Their research shows how the distributed approach can reduce contention on a shared open list, and details different communication strategies that could be used so that threads do not work on bad parts of the search space. The general approach is that multiple processors work on expanding different states at the same time. The reference guide for ParaTask by the University of Auckland details how this can be done using a tasking model, through the ParaTask Java library. Introduction to Parallel Computing by Grama et al. [3] explains how a different termination condition is needed for the parallel best first search algorithm since the sequential termination fails in certain cases for the parallel implementation. The research paper Scalable, Parallel Best-First Search for Optimal Sequential Planning explains another distributed algorithm which uses a message passing architecture to distribute the workload using a simple hash function[4]. Lastly, preliminary research has also been explored on minimizing contention on open lists by searching through an abstract graph search space as suggested in Parallel Best-First Search: Optimal and Suboptimal Solutions[5].

4 Library

This project uses problems provided by TSPLIB, which contains a collection of traveling salesman and related optimization problems stored in text files [6]. Furthermore, it provides answers which can be used to check the integrity of our solutions. This project also uses TSPLIB4J which is a free Java library licensed under the MIT license which enables problems to be extracted from text files in TSPLIB for use in Java applications.

5 Sequential Implementation

A priority queue is used to store the open list of states. A state contains the current path of cities along with the current cost value. The cost value (represented as *f(n)*) is a combination of the current cost of the path and the heuristic value for that state. The heuristic value is calculated using a modified minimum spanning tree (MST) heuristic as described in [1]. Although Kruskal's algorithm was suggested to be used to create the MST, instead it was found that Prim's algorithm is more suitable for a complete dense graph which is created for the travelling salesman problem. This is because Kruskal's algorithm requires sorting of all edges so it has a time complexity of $\mathcal{O}(E \log V)$ [7]. In order to get a more accurate estimate, the minimum distance between both the start city and state's end city with the MST is added to the heuristic value, and this will still always underestimate the TSP solution. It is also a consistent heuristic. At each iteration, the priority queue is polled to get the state with the least estimated cost. This is then expanded by finding all the children states, which are all the states that can be reached by the addition of a single edge that haven't yet been visited. A global closed list is not needed since there is only ever one path to get to a particular state in a tree. However each state contains a path that contains all the cities previously visited. This process continues until a state is reached that contains all the cities in the graph. At this stage, then the origin city is added to the end of the path, and this is now a Hamiltonian cycle [8], which is also the optimal solution.

6 Parallel Implementations

Both task and data parallelism were considered for the parallel implementation of A*. Task decomposition of A* showed that A* is an inherently sequential process where previous states have to be expanded before the next states. Additionally, brief research into parallelisation of heuristics found that the speedup is minimal and not feasible for efficient parallelisation of A* for TSP. As a result, task parallelism was not considered for this project because the sequential nature of A* limits the parallelisation opportunities. Instead, data parallelism was used. Data parallelism distributes the search state space between the threads. With data parallelism, each thread has a partition of the entire search space and is able to search independently. The distribution of the workload between the threads increases expansion throughput. In an ideal situation, each thread will be searching in the search space which contains the solution, however, this must be coordinated by communication between the threads. This introduces search, communication and synchronization overhead and there is a trade-off between exploring extra nodes that wouldn't be explored in the sequential version, and the communication cost of sharing between threads. In this project, 3 parallel A* and 1 parallel IDA* algorithms have been implemented and is discussed below.

6.1 Centralised Parallel Implementation

This is the simplest approach for parallelization of A*, and involves the use of a shared frontier that all threads can access. A Java PriorityBlockingQueue was used for the open list, which is a thread-safe priority queue. The threads each carry out the same algorithm, and will share the entire frontier. This was implemented using the TASK (*) annotation in ParaTask. Each iteration, each thread polls for the lowest cost state in the queue. The thread will carry out the same expansion process as the sequential algorithm, and then add the each of the child states into the shared frontier. The advantage of this approach is that it is easy to implement and there is little communication cost as threads do not interact with each other. The downside of this solution is that the threads are always competing for the shared frontier, which means there is considerable synchronisation overhead. Research suggests that contention for the open list limits speedup, since the parallel runtime will be at least the access time of the open list multiplied by the total number of iterations [3]. If an increased number of processors were used, many threads would continually be competing for the shared frontier, and this could lead to poor performance and scalability. Hence it was decided to tweak this algorithm into distributed versions to lower the locking and synchronisation overhead of the shared frontier.

6.2 Blackboard Distributed Parallel Algorithm

The next approach considered involves the use of a shared blackboard rather than an entire shared frontier. Threads contain their own local frontier, which are normal priority queues. The blackboard is a PriorityBlockingQueue. Once again, the TASK (*) annotation was used for data parallelization. At each iteration the thread operates only on its own local queue. Communication with the blackboard is done once per iteration, where each thread compares its best state to the best state of the blackboard. If the local best state is a certain threshold better than the blackboard best, then the thread will copy a number of the best states from the local frontier to the blackboard. If it's a certain threshold worse than the blackboard best, then the current thread takes some of the best states from the blackboard. Each thread carries out this process. As a result, communication with the blackboard is done only once per iteration so the access time to the shared frontier is reduced. However, all threads still access this shared blackboard once per iteration, so this would not scale as well as a fully distributed approach.

6.3 HDA*

HDA* is a distributed parallel best first search implementation developed by Akihiro Kishimoto, Alex Fukunaga and Adi Botea. It uses hash-based work distribution and asynchronous communication to search in parallel [4]. The algorithm proposed in the paper has been modified to fit the TSP problem description. Instead of using a true message passing architecture, this project mimics message passing in a shared memory system by passing messages through a shared memory buffer. To achieve data parallelism, this project uses the TASK(*) notation provided by ParaTask. Each thread maintains a private open list and a public channel which receives incoming messages sent by other threads. The private open list is a PriorityQueue which maintains discovered but not yet visited states and the channel is a thread-safe ConcurrentLinkedQueue. All channels are stored in the shared channels object which is accessible by all threads. At each iteration, a thread adds any received messages from its channel into its private open list. If the channel is empty, it checks its private open list and expands any states. The generated states are then sent to the target threads public channel using a hash function. This project uses the java hash function to determine the appropriate target thread and distribute the workload uniformly. If the channel queue and private

open list is empty, the thread goes into sleep and is awakened by other threads when messages have arrived. This approach is similar to the producer consumer model. The benefits of HDA* include using asynchronous communication which reduces synchronization overhead over the threads private open list. Furthermore, the use of a simple hash function to distribute the workload means that there is no complex load balancing mechanism and makes the algorithm simple.

7 IDA* and IDA* Parallel

The memory usage of the sequential and parallel A* algorithm grows exponentially as the traveling salesman problem size increases. To improve the scalability of the A* algorithm, Iterative Deepening A* algorithm (IDA*) was examined to improve the memory usage of A* algorithm. IDA* is a modified version of the Iterative Deepening Depth First Search which in this version, uses the heuristic cost of the A* algorithm to increase the depth limit of the search. IDA* algorithm takes much longer to find the optimal solution than A* algorithm, however, the memory complexity of IDA* is linear whereas A* algorithm memory complexity is exponential. This a significant improvement with respect to memory usage but at a cost of time performance. A sequential version of IDA* was implemented to gather results and compare with the sequential A* algorithm. The sequential implement of IDA* involves the following steps:

1. Calculate the initial bound using the heuristic cost of the initial state
2. Perform depth first search on the initial state
3. When the current state of the search is greater than the current bound, stop and return to the parent state with the current state heuristic cost as the next possible bound value
4. After the search expand all the states within the current bound, choose the minimum heuristic cost that was return as the next bound value.
5. Repeat step 2-4 until a solution is found.

A parallel version of IDA* was also implemented to investigate the potential speedup compared to the sequential A* and IDA* algorithm. The parallel implementation is based on a research article [9]. In the article, it describes the overall algorithm as a two stage process. Initially the algorithm generates a queue of states using depth limited breadth first search. The depth size can vary but in this implementation, the depth size of 1 was chosen to minimise the time it takes to generate the queue and also limit the number of states in the queue to process in the next stage. The next step is to perform IDA* on each of the states in the queue, and this is where the parallelization of the algorithm occurs. Multiple IDA* algorithms will run at the same time however they will be using a shared bound to ensure that the search is guarantee to find the optimal solution when a solution is found. The shared bound is updated at the end of each iteration of each IDA* search, where there is a synchronization barrier to find the minimum heuristic cost from all the IDA* searches as the next bound value. This was done using reduction at the synchronisation barrier.

8 Termination Conditions

The results of the literature review also showed that the termination condition for the sequential algorithm fails for the parallel versions, since at any moment p states are being explored, and any of these may lead to search spaces with better goal states. The initial goal state found in parallel

implementations is not guaranteed to be optimal therefore the other processors must prove there is no goal state better than the initial goal state. For the centralised approach, the termination condition needs to be that the cost of each state a processor is currently expanding needs to be at least the cost of the current goal state. This was implemented using a local flag and a shared countdownlatch. However, more factors need to be considered for the distributed parallel algorithms. For the blackboard approach, it is also necessary to ensure that that the blackboard states won't lead to any search spaces that contain better goal states. The way that this was implemented was that as soon as a thread finds a goal, the threads share the rest of the blackboard states and do not add any further states back to the blackboard. After this, the same termination condition as the centralised approach applies. Finally, the hash distributed termination approach checks every private open list and incoming messages to guarantee an optimal solution. This is implemented through a global flag, which is set true when the initial goal state is discovered. Each thread checks if any states in its private open list and any incoming messages is better than the discovered solution. There is a synchronization barrier, implemented using CurrentTask.barrier() which waits until all threads have completed this investigation and then terminates collectively. When the flag is set true, instead of sending messages into the target thread's public channel, expanded states are sent into its own private open list.

9 GUI

As the increase in cities increases the search space exponentially and time taken to find a solution, the GUI was used to visualise the search progress across each thread and confirm the program is still searching. During the project, two GUI versions were implemented. The current GUI shows the test problems found in TSPLIB and the main panel shows the number of worker threads working on the problem. For the parallel implementations it shows the number of logical processors found in the computer. To visualise the current explored path, the GUI uses the visualisation from the TSPLIB4J library.

10 Results and Discussion

All the experiment was conducted an Intel core i7 4770 processor @ 3.40GHz, 32GB ram machine. All of the algorithms was tested on problem size between 10 and 35 cities so that all of the algorithms was able to solve the problems within the hardware limitation and within reasonable time.

Looking at the speedup results from Figure 1, it was found that all of the parallel implementations perform worse than the sequential version with problem sizes of 15 cities or below, as these tend to take less than 100ms for the sequential version. For the larger problems, the parallel version is significantly better. All of the algorithms, the maximum efficiency achieved was for the 30 city problem. The algorithms appear to get a small decrease in speedup at 35 cities. It is suggested that a reason for this is that the hardware doesn't scale with the increased problem size thus resulting in more overhead due to factors such as the Java garbage collection.The centralised algorithm gave the best efficiency at 30 cities, which had a speedup of 4.61 and hence a parallelization efficiency of 0.58. However, further analysis of results across different numbers of processors should also be considered since it is thought that the speedup may decline as the number of processors increases for the centralised approach. Also, the extra termination conditions for the Hash and Blackboard approaches adds to the overhead, which is the main reason why they get less speedup compared to the centralised approach. These algorithms can achieve better speedup than the centralised version if the termination conditions are not included, but the tradeoff is that non-optimal solutions may

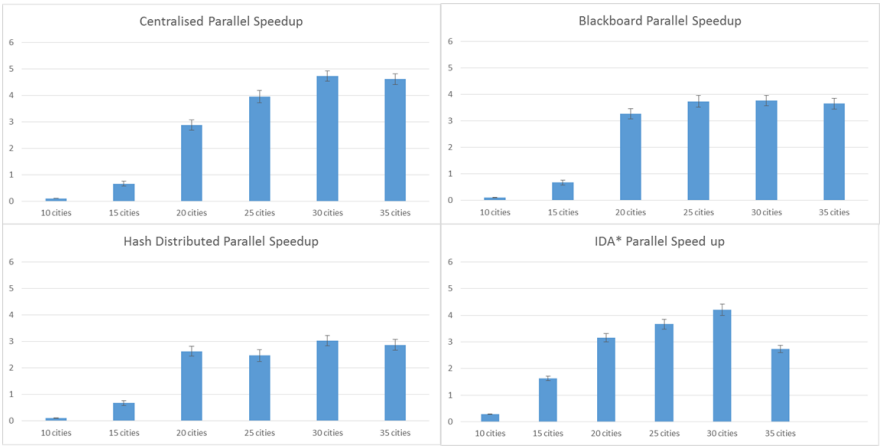

Figure 1: Speedup for Parallel A* implementations

be found, as described earlier. The IDA* algorithm is much slower than the three A* algorithms, therefore the speedup was calculated in comparison to the sequential IDA*. As a result, similar speedup values are achieved while keeping memory, synchronisation cost, and communication cost to a minimum. This means that it can potentially be used to solve larger problems if time was not limited. Using the number of threads parameter in Parallel Task, it was also possible to compare changes in speedup across different numbers of processors, as each thread runs on a single processor. The results from Figure 2 show the speedup on a 27 city problem.

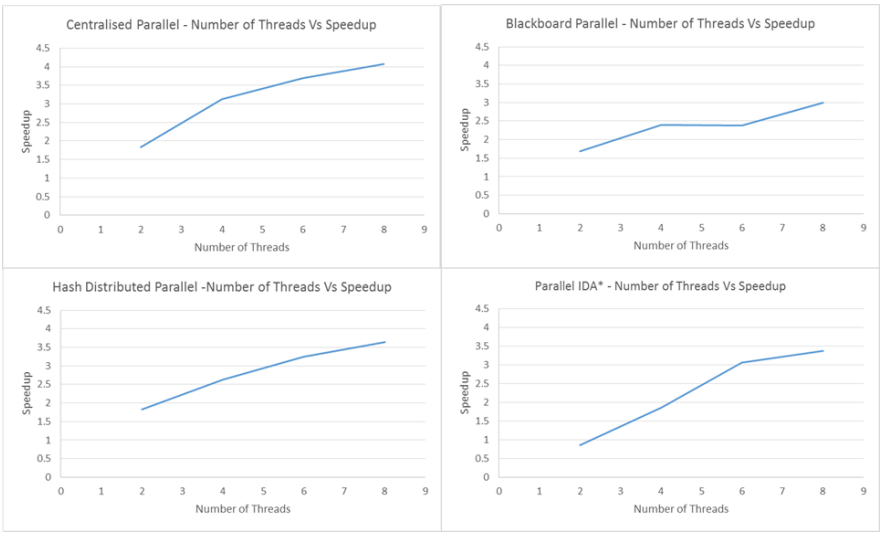

Figure 2: Speedup by number of threads for Parallel A*

As expected, the trend is that the speedup increases as the number of threads increases. It was expected that competition for the shared frontier in the centralised version would limit speedup with more processors, and this is noted by the slope appearing to decrease after 4 processors. However, this is not as obvious as expected. This could be because the number of processors was limited by the computer used, and if a computer with more processors was used then this decrease in speedup may become more noticeable. As previously noted, certain parameters such as the threshold and the number of states to copy have an effect on the performance of the blackboard algorithm. Experiments

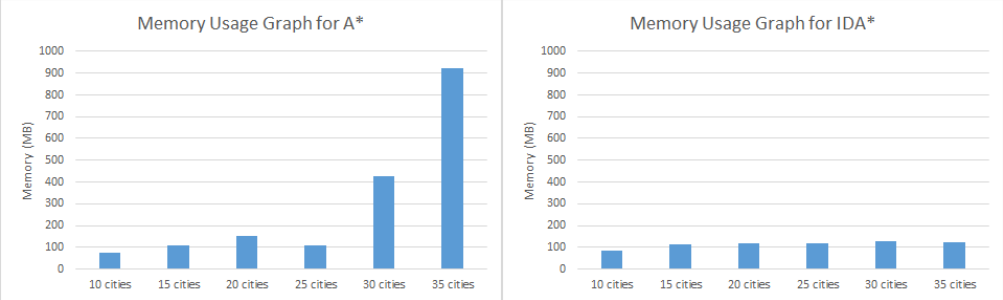

Figure 3: Memory usage for A* vs IDA*

were carried out to compare the differences of these parameters, and the results of these experiments on a 27 city problem show that the threshold for the blackboard algorithm seems to give the best speedup for the 27 city problem when it is around two processors. This means that states are transferred if the difference between the best local and blackboard state is more than two, which is fairly often. However, it is expected that if the same experiment was carried out with more than eight processors, there may be a difference in the results due to competition for the shared blackboard. This could mean higher thresholds would reduce competition and give better speedup. However this could not be tested due to hardware limitations. For memory comparisons, the sequential A* algorithm and the sequential IDA* algorithm were executed over several problem sizes. Figure 3 show a trend for the A* algorithm memory usage increasing exponentially while the memory usage of the IDA* algorithm doesn't grow significantly. This matches with the memory complexity theory of the A* and IDA* algorithms.

11 Limitations

One of the main limitations of this project was the amount of memory available for the A* algorithm to use. Since the memory usage exponentially increases, the A* algorithm was unable to be tested on larger problem sizes. Another limitation was the requirement of using ParaTask to parallelize the A* algorithm, as some of the parallelization methods such as HDA* use message passing technique to parallelize the A* algorithm in a distributed approach. As a result, message passing was built on top of a shared memory architecture to implement HDA*. Since HDA* implementation does not use a true message passing architecture, the results gathered from the HDA* algorithm may not reflect the true performance of the algorithm. The number of processors was also a limitation as there were no computers available with more than eight logical processors. For future work, the algorithms should be tested on larger problem sizes and a greater number of processors to reinforce the validity of the results gathered in this project. Lastly, at this stage, the parallel implementation of HDA* is not always guaranteed to be optimal because of the incorrect implementation of the distributed termination condition. The optimal solution is found in most cases but occasionally the non-optimal solution is found. If more time was available then this could be fixed to ensure the optimal solution is always found.

12 Conclusion

In this report, different ways of parallelizing the A* algorithm to solve the travelling salesman problem were investigated. It was found that the A* algorithm for the travelling salesman problem

is different than the traditional A* algorithm for path finding problems. The parallelisation can achieved using a centralised or distributed approach and this project presented three implementations which covers both approaches. It was discovered that the overhead of parallelizing the A* algorithm includes the communication cost, synchronisation cost and exploring unnecessary nodes in the search. On the other hand, the IDA* algorithm can be used when memory usage becomes a problem when using the A* algorithm, which can also be parallelized but in a different way. Finally, the results show that the speedup of the parallelized A* algorithms scale differently over the number of cities and processors.

13 Table of Contributions

Contributor	Work
Wesley	Integrate the TSPLIB4J into the A* algorithm
Wesley	Centralised Parallel Implementation + Termination Condition
Wesley	Blackboard Implementation + Termination
John	Basic A* algorithm implementation
John	IDA* algorithm implementation + Parallel IDA* implementation
Chang Kon	HDA* implementation + Termination Condition
Chang Kon	GUI
Everyone	Literature Review + Presentation preparation

References

[1] "Problem solving agent for traveling salesman problem," http://www.public.asu.edu/~huanliu/AI04S/project1.htm, 2004.

[2] V. Kumar, K. Ramesh, and V. N. Rao, "Parallel best-first search of state-space graphs: A summary of results." in *AAAI*, vol. 88, 1988, pp. 122–127.

[3] V. Kumar, A. Grama, A. Gupta, and G. Karypis, "Introduction to parallel computing. 2nd," 2002. [Online]. Available: http://parallelcomp.uw.hu/ch11lev1sec5.html

[4] A. Kishimoto, A. S. Fukunaga, and A. Botea, "Scalable, parallel best-first search for optimal sequential planning." in *ICAPS*, 2009.

[5] E. Burns, S. Lemons, W. Ruml, and R. Zhou, "Best-first heuristic search for multicore machines," *Journal of Artificial Intelligence Research*, pp. 689–743, 2010.

[6] G. Reinelt, "Tsplib," http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/, 2013.

[7] A. Kershenbaum and R. V. Slyke, "Computing minimum spanning trees efficiently," in *Proceedings of the ACM annual conference-Volume 1.* ACM, 1972, pp. 518–527.

[8] E. W. Weisstein, "Hamiltonian cycle," http://mathworld.wolfram.com/HamiltonianCycle.html, 2016.

[9] B. A. Mahafzah, "Parallel multithreaded ida* heuristic search: algorithm design and performance evaluation," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 26, no. 1, pp. 61–82, 2011.