UCSB CMPSC 290B: Java-centric Cluster & Concurrent Computing

Project Report

Distributed TSP Analysis

by

Erik G. Jansen, Varun Sivapalan, Steffen Birkeland & Martin S. Tveter

Abstract

We set out to improve an implementation of the distributed branch and bound algorithm for the travelling salesman problem (from now on referred to as TSP). The improvement involved adding additional functionality such that the general algorithm be improved in the face of latency in the distributed system, making the algorithm able to adapt to variation in the said latency. We also extended the API to include functionality and a user interface to provide instrumentation to the user application programmer, enabling analysis of the performance of the algorithm with a given input, and viewing the actual progress during execution time of the algorithm.

Table of Contents

1. Introduction

Branch and bound is an algorithm design used to solve combinatorial optimization problem in order to reduce the number of feasible solutions one has to compute. In a combinatorial optimization problem, the number of feasible solutions typically grows exponentially with the input size of the problem. With the branch and bound technique, the initial problem is recursively divided into subproblems until we get to the bottom case where all calculation is done locally at each processor. Thus, reducing the number of calculations to reduce the computation time for the system is crucial. The travelling salesman problem (TSP) is a NP-Complete problem in which we have a set of cities, and we are to find the shortest route that visits all cities exactly once and returns to the origin city [1]. The original TSP problem contains a set of cities in *n* dimensions, but in our case, we will use 2 dimensions, making it a problem in the .Euclidean space. This means that we can view the problem as finding the shortest tour between the cities located in a first quadrant of a coordination system, where each city x and y coordinates.

For this course project, we will modify the branch and bound API from the previous homework so that it will be robust in a distributed system. A distributed system may consist of computers located anywhere in the world, and the larger the physical distance between the computers are, the higher the communication latency will be between them. The average transatlantic network latency is 73 ms, and from Singapore to the US it is 175 ms [2]. However, for a fiber optic cable, there is on average a network latency of 4.9 µs per kilometer [3], which for a round trip along the equator (ca 40,000 km) would equate to ca 400 ms. Network latency may be also influenced by the time of day. Statistically, the total network traffic peaks during leisure and working hours, and is significantly lower during night hours [4]. We want to make the system more robust while having fluctuating network latencies. The system should be able to adapt to the experienced latency. With little latency, the recursion limit of where to stop decomposing tasks should be lower than if the system experiences higher latencies. We will do experiments where we will simulate different latencies and based on the result, we will optimize the system configuration.

Project goals:

1. Make the system application-independent.
2. Improve the pruning functionality of the branch and bound algorithm.
3. Make the system able to adapt to network latencies during execution.
4. Create a GUI that enables visualization of performance and results during execution, and gives the application programmer/user the possibility to analyze the input set and make changes to the application accordingly.

2. Functional requirements

ID:	**Description:**	**System test:**
FR1	A client can only send a job to a compute space when the compute space is running.	Run *ClientEuclideanTsp* without running *SpaceImpl*.
FR2	If the compute space has received one job, it can not received another until the first one is completed.	Run *SpaceConsoleGUI*, press ‘start space’, run ComputerImpl’, run *ClientEuclideanTsp* once, then twice.
FR3	A computer space can be started both with and without the ability to run tasks locally in space.	Run *SpaceConsoleGUI*, choose option ‘hasSpaceRunnableTasks’, press ‘start space’.’
FR4	User interface reports information dynamically to the user graphically during execution time.	Run *SpaceConsoleGUI*, start space, start *Computer(s)* and then *ClientEuclideanTsp*, observe GUI.
FR4.1	The user can view overall progress of job execution.	Run SpaceConsoleGUI, start space, start *Computer(s)* and then *ClientEuclideanTsp*, observe task progress bar.
FR4.2	The user can view information about the latency in the system.	Run *SpaceConsoleGUI*, start space, start *Computer(s)* and then *ClientEuclideanTsp*, observe latency graph.
FR4.3	The user can view information about the number of tasks in.	Run *SpaceConsoleGUI,* start space, start *Computer(s)* and then *ClientEuclideanTsp*, observe task information.
FR4.4	The user can view information about pruning depth.	Run SpaceConsoleGUI, start space, start *Computer(s)* and then *ClientEuclideanTsp*, observe pruning information.

3. Key technical issues

1. The API should be application-independent and generic so it can be reused in future projects and not be bounded to a specific problem.
2. Make the system adapt to fluctuating network latency
    a. Based on the latency experienced (simulated), we will adjust the prefetching/scheduling, the bottom case (recursion limit) and pruning.
        i. If the system experiences low network latency, then it will have a lower buffer size and recursion limit and pruning will be quicker.
        ii. With a high network latency, the system will set a higher buffer size and recursion limit, to make an effort to make up for the loss of computation time.
3. Instrumentation in form a GUI that will provide the user with a visualization of the performance and results during execution. It should give the user the possibility to analyze the input set and make changes to the application accordingly.
4. Improve pruning algorithm from previous homework.
    a. Changing the algorithm that calculates the lower bound of the branch and algorithm to be more efficient in terms of performance. By improving the algorithm that uses the 2 least cost arc in combination with a minimal cost spanning tree(MST) for the partial problem to only update the necessary parts of the MST the time complexity can be drastically reduced. The computation needed for updating the lower bound must be reduced to a minimum or it will otherwise have severe impact on the overall performance of the job execution.
    b. LIFO-queue for tasks to do DFS of the search tree.

4. Architecture

The changes done from the previous work with TSP is added functionality to simulate and analyze latency in the distributed network, and to adjust the settings/preferences of a *Computer* during execution depending on the dynamic latency. E.g. if there is high latency when sending method invocations calls between a *Computer* and a *Space*, it is beneficial to have fewer calls. By dynamically changing the limit of task depth where the *Computer* performs remaining calculations instead splitting the task into further subtasks this can be achieved.
The additional classes provided to implement said functionality is the class *ComputerStatus* and *ComputerPreferences*. These classes hold the settings and preferences of a given *Computer*.

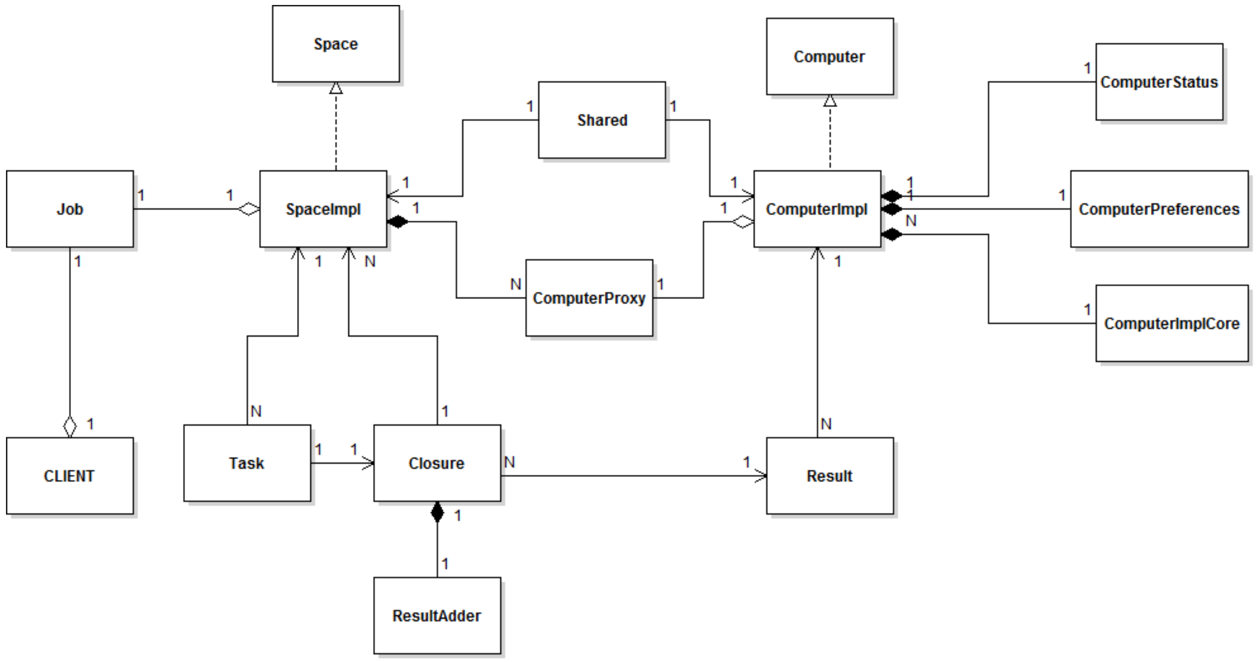

Figure 1: Class diagram of the API, client and system classes

A graphical user interface has also been implemented to allow for easier and more descriptive analysis of the execution of tasks. As one can see from ‘Figure 2’, it is based on a model-view-controller pattern, where there is controller for the *Space* that controls the *SpaceConsoleGUI* and updates it dynamically while the *Space* computes tasks and results. The controller accesses the models, *TimeLeftEstimation*, *TasksProgressModel*, and *LatencyData*, and uses this information to update the view so that the user can get informative results and statistics about the execution of *Job* while the *Space* is running, such as estimated time left to complete *Job*. Also, the application programmer can use the GUI to decide on which code changes and enhancements are needed.

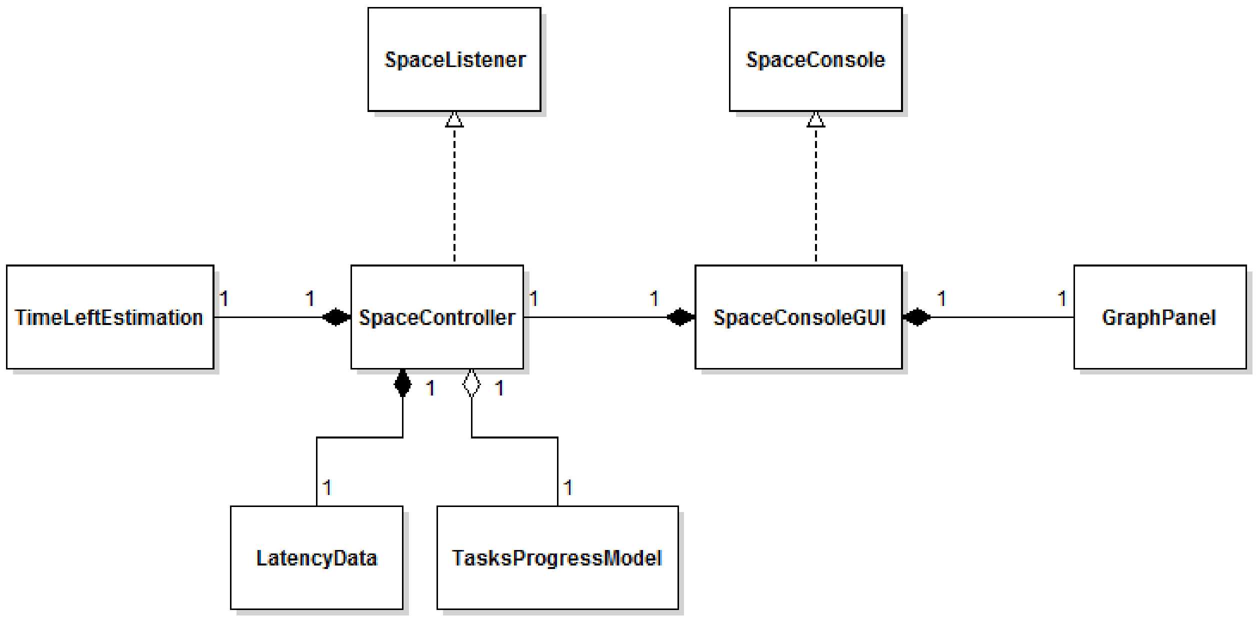

Figure 2: Class diagram of GUI package classes

5. Experiments

For our system to work optimally we needed to map the how much time our system would spend in each part of the task

Cities	RecLim	Time	TaskSplitTime		PermutationTime		Latency
		sec	Avg ms	Tot ms	Avg ms	Tot ms	
14	11	261	5	5	7208	64875	0
14	12	333	4	4	84148	84148	0
13	11	21	4	4	6636	6636	0
13	10	19	4	4	572	5154	0
13	9	21	4	4	56	4645	0
13	8	25	3	9	5.8	4106	0
13	10*2						0

We got a couple of pointer from this.

First we saw that the real bottleneck here is the space. We see from this data that approximately 75% of the total time is spent in the space. This is a very big overhead, and shows us that for our system to be efficient we need the tasks to be relatively big.

From this we also get that the best is to set the initial recursion to 9.
What we found through testing is that it's very individual to the size of the problem and the delay settings what preferences that are optimal. Fast rec 8 slow rec 10

Cities	RecLim	Time	TaskspliTime		PermutationTime		Latency	Dynamic
		sec	Avg ms	Tot ms	Avg ms	Tot ms		
13	9	237	236	4018	316	40766	50 +-10 ms	No
13	8,9,10	44	198	595	759	10631		Yes
14	9	450	2.6	16	121	130176	no	no
14	9	>40 min						
14	8,9,10							yes
14								

What we see from this is that the space now takes even more of the time total time spent. Also the permutations take longer time.

We realised after some testing was that we sent a lot of messages over rmi other than the task and result. What we think that we saved in signalling by adjusting prefetching and recursion limit, we were losing in transmission delay of the extra objects we were sending.

Also we noticed that when testing the real latency of our system it was quite bigger than we had expected. We could ping the working computer with 5-6 ms. But the actual transmission time was 30-85 ms. We think this is because of the use of TCP for RMI.

6. Conclusions and future work

We succeeded in implementing the extra properties we wanted for our system. We god dynamic adjustment of recursion limit and buffering, GUI that shows latency and where the time is spent in our system, as well as progress. We improved our queue, and pruning algorithm.

We also learned some lessons about distributed systems for problem solving. The optimal settings for adjusting to latency can be hard to get correct. It's very individual for the problem and the size of it. Also every communication between a computer and space will be affected by network latency, therefore a focus that is more application independent would be to focus on less communication between server and worker. Focus more on piggybacking on the objects that are already being sent.

If you want to use a distributed system that might suffer from latency make sure that the tasks are big enough to make it worth the extra computational power. Also the space can be a huge bottleneck, and running a GUI on the same machine can increase that bottleneck.

7. References

[1] Travelling Salesman Problem. In *Wikipedia*. Retrieved June 8, 2015 from http://en.wikipedia.org/wiki/Travelling_salesman_problem

[2] IP Latency Statistics - Verizon Enterprise Solutions. Retrieved June 8, 2015 from http://www.verizonenterprise.com/about/network/latency/

[3] Latency(engineering). In *Wikipedia*. Retrieved June 8, 2015 from http://en.wikipedia.org/wiki/Latency_%28engineering%29#Fiber_optics

[4] Hour-by-Hour Examination: Smartphone, Tablet and Desktop Usage Rates - Chitika Online Advertising Network. Retrieved June 8, 2015 from https://www.chitika.com/browsing-activity-by-hour