Parallel Heuristics for TSP on MapReduce

Siddhartha Jain Matthew Mallozzi

Brown University – CSCI 2950-u – Fall 2010

Abstract

We analyze the possibility of parallelizing the Traveling Salesman Problem over the MapReduce architecture. We present the serial and parallel versions of two algorithms - Tabu Search and Large Neighborhood Search. We compare the best tour length achieved by the Serial version versus the best achieved by the MapReduce version. We show that Tabu Search and Large Neighborhood Search are not well suited for parallelization over MapReduce.

1 Introduction

The Traveling Salesman Problem (TSP) is one of the most intensively studied problems in optimization. Loosely speaking, given a set of cities on a map, the problem consists of finding a tour that goes through each city exactly once and ends in the same city it started with. There has been much research done on finding efficient heuristics to get provably optimal and close to optimal solutions to TSP problems [1]. Recently a polynomial time approximation scheme (PTAS) was discovered for the Euclidean TSP problem [2]. However, with Moore's law nearing the end of its lifetime and parallel and cloud computing gaining prominence in the past few years, finding efficient parallelizations of existing algorithms will gain importance. Furthermore, given the push towards cloud computing, it will become increasingly necessary to adopt algorithms to existing cluster computing frameworks like MapReduce [6].

There's already been much work done in parallelizing TSP heuristics [12][4][3][15]. However there doesn't seem to be any literature on using advantage of existing cluster computing architectures for solving TSP. In this paper we explore the problem of adopting existing algorithms for TSP on the MapReduce cluster computing framework. We focus on the Euclidean TSP problem. We analyze the difficulties in adopting an algorithm for TSP to MapReduce and demonstrate the validity of our analysis via experimental evaluation on a MapReduce cluster. Finally, we propose ways to remedy the problems we encountered.

The paper is divided in the following manner. In Section 2 we formulate the TSP problem and present some common heuristics used to find close to optimal solutions for the problem. In Section 3 we present the MapReduce framework on which we are going to run the parallel versions of our TSP algorithms. In Section 4, we describe how we ported existing serial algorithms for TSP onto the MapReduce Platform. In Section 5, we present our experimental results and finally in Section 6 we draw some conclusions and present possible avenues for exploration.

2 The Traveling Salesman Problem

2.1 What is TSP?

In its most general form, the Traveling Salesman problem is defined as finding a min-cost Hamiltonian cycle in a graph. When the edge weights on the graph form a Euclidean metric, we are reduced to the familiar Euclidean Traveling Salesman Problem.

Definition Let $G = \langle V, E \rangle$ be a directed or undirected graph where V is the set of vertices and E the set of edges.. Let w_e be the weight of edge $e \in E$. Then the problem is to find a set set of edges $S \subseteq E$ such that there is a path from every $s \in V$ to every $t \in V, s \neq t$ which uses edges only in S, $\forall v \in V$, there exist exactly two edges in S having v as an endpoint and $\sum_{e \in s} w_e$ is minimized.

There are different variants of the TSP problem - all special cases of the general definition given above. The Euclidean TSP variant which is the canonical and most studied TSP variant has already been mentioned. In addition, we have Asymmetric TSP where the edges are directed and the edge $s \rightarrow t$ has a different weight than the edge $t \rightarrow s$, the Non-Metric TSP variant where the edge weights needn't form a metric space, etc. More complex problems based on the Traveling Salesman problem include the vehilce routing problem where several disjoint tours have to be found, the idea being that each tour is traversed by a different vehicle, pickup and delivery problems, etc. In this paper we study the canonical version of the TSP problem - the Euclidean TSP problem.

2.2 Solution Goals

Generally, for a TSP solver, one either tries to obtain a provably optimal solution or one tries to get a solution as close to the optimum as possible without actually proving that the solution is close to the optimum. While the former goal has an advantage in that it gives a guarantee of the quality of the solution, it is generally very slow and infeasible to apply to instances of a large size. Thus we opt for the second goal. We use a paradigm termed local search [11] to get high quality solutions to the TSP problem. The basic idea of local search is that one starts with a suboptimal solution. One then makes small local changes to the solution to move towards a more optimum solution. Below we present the common local search algorithms used to solve TSP.

2.3 Common Algorithms

2.3.1 Tabu Search

Tabu Search was first introduced by Glover in 1986 [9]. It has since been used in countless optimization heuristics including heuristics for TSP [13] [8] [14] in both serial and parallel settings.

The idea is that one defines a set of *moves* one can make to change the tour - for instance, one could swap the locations of two cities in the tour or if two edges of the tour are crossing each other, one could exchange them with non-swapping edges. The set of tours reachable from the current tour by making a single move is called the *neighborhood* of the current tour. We then keep making moves which decrease the total length of the tour until we reach a local minimum. If we stop here, then in general, we'll get a highly sub-optimal solution. Thus we need to get out of the local minimum somehow. Let the change in the tour length caused by making a move be denoted by Δ. One way to get out of the local minimum is, rather than only taking a move that has a negative Δ, one takes a move that has the minimum Δ out of all possible moves (In particular,thus a move with a positive Δ is permitted. However this can lead to an infinite loop.

Let the tour at the local minimum be T. Now if we make any move to the tour T' to get out of the local minimum, since T is a neighbor of T' with a smaller tour length (as T is the local minimum), the algorithm will immediately make a move towards T again and will keep cycling. To prevent this, the Tabu Search heuristic forbids the algorithm from repeating any recent configurations. The basic idea is to maintain a tabu list of finite length which keeps track of some of the recent configurations. This means that now if we move from T to T', T will be in the tabu list and we'll be able to get out of the local minimum.

The Tabu heuristic applied to the Euclidean TSP yields the following algorithm :-

1. Let the set of cities be C. For all cities $c \in C$, initialize a tabu queue t_c of maximum capacity k called the tabu length. The tabu queue t_c will store the past *successors* of the city c.
2. A random tour is generated by the following algorithm :-
   (a) Let the initial set of cities be $S = C$. Then
   (b) Remove c_1, c_2 from S and make an initial partial tour by setting them to the the successors of one another.
   (c) While $S \neq \emptyset$
      i. Remove c from S.
      ii. Insert c in the tour in the place that will increase the tour length by the least.
3. Loop until n iterations:
   (a) A random city c is chosen.
   (b) For all cities in the set $N = \{c' \in C, c \neq c'$ and $succ(c') \notin t_c$ and $succ(c) \notin t_{c'}\}$ where $succ(c)$ is the successor city of c, choose $c' \in N$ such that the Δ associated with swapping c and c' is minimum.
   (c) Add $succ(c)$ to the end of queue t_c and $succ(c')$ to the end of $t_{c'}$ and then swap c, c'.
   (d) If the new tour is better than any previous tour (in terms of route length), store it.
4. Output the best tour reached in the course of the search.

2.3.2 Large Neighborhood Search

The second major heuristic which we experimented with is called Large Neighborhood Search. The basic idea is to start with a random solution. We repeatedly remove a small segment of the tour from the tour and then insert the cities into the tour. We accept the new solution only if it will decrease the total tour length. The algorithm in full is :-

1. Let the set of cities be C. Let $k < |C|$ be the length of the segment to remove at each iteration.
2. A random tour is generated just as how it was for the Tabu Search algorithm.
3. Loop until n iterations:
   (a) Choose a segment of length k to remove from the tour. Let the cities in the segment be the set S. Then
   (b) While $S \neq \emptyset$
      i. Remove c from S.
      ii. Insert c in the tour in the place that will increase the tour length by the least.
   (c) Make the new tour the current tour only if it has a smaller tour length than the current tour.
4. Output the current tour.

2.3.3 Other Algorithms

Two other algorithms that our popular for TSP are Ant-colony [7] and Genetic algorithms [10]. At a high level, in ant colony algorithms, several solutions are generated and then edges which have the picked the most number of times have a higher probability of being part of future solutions. Thus in effect, multiple solutions are repeatedly generated to tune the heuristics to generate a good tour for the problem. Genetic algorithms also generate a large number of solutions but instead of optimizing the heuristic for generating the tours, they have a sophisticated merging procedure to incorporate the good characteristics of different tours into one tour.

3 MapReduce

MapReduce [6] is a distributed computation framework developed at Google in order to process very large sets of data that have been split over many computers. Its programming abstraction allows the user to forget about many of the difficulties associated with distributed computing: splitting up the input to various machines, scheduling and executing computation tasks on the available nodes, coordinating the necessary communication between tasks, and dealing with any machine or network failures that will almost certainly arise.

MapReduce deals with its input in terms of key-value pairs, which are generated from an input file by user-configurable rules. MapReduce uses a very simple programming model, which in its most abstract form requires its user to write only two functions - unsurprisingly, *map* and *reduce*.

3.1 Framework

In functional programming, *map* is a higher-order function that applies a one-argument function to every item in a list of items, and returns a new list. In MapReduce, however, since the input is in terms of key-value pairs rather than just arbitrary items in a list, *map* is a construct provided by MapReduce that applies a mapping function to a list of key-value pairs, where the mapping function consumes a key-value pair and outputs a list of key-value pairs. This list may be empty, it may have only one element (common), or it may have multiple elements.

Although *map* is close to its common interpretation in functional programming, *reduce* is quite different. It follows the same general principle: take a bunch of items in a list and *fold* or *reduce* them into one item. However, this is where the similarity ends. In functional programming, *reduce* or *fold* is a higher-order function that applies a two-argument function successively to each item in a list, where the result of each application is fed into the next *reduce* as one of its arguments. In MapReduce, a *reduce* is performed over all outputs of map with the same key. A task's *reduce* function will take in a key and a list of values, and output a key-value pair. In this way it is more flexible than the common view of *reduce*, both in what can actually be done with the computation and in the types that can be returned - in functional *reduce*, the type of the result must be the type of whatever is stored in the list, because the return type of the binary operation passed into *reduce* must be the type of its two arguments, so the result can be the argument to another instance of the same binary operation.

3.2 Typical Problems

The original MapReduce paper provides some examples of what can be implemented using this framework.

The primary example is WordCount, which simply counts the number of occurrences of each word in a document. The input to the function to be mapped is a key-value pair, where the key is the line number in the input file, and the value is the text from that line - this function outputs a list of key-value pairs, where the keys are words, and the values are the value 1. The *reduce* function is applied once for each key (word), receiving the key and a list of all the values (1) output by *map* corresponding to the word. It then simply sums these values to obtain a single number, which is the number of occurrences of the word in the input document. The output from reduce is then the word as the key, with the number of occurrences as the value. Thus the output from the whole MapReduce task is a list of key-value pairs representing each word and its frequency.

Another example is Distributed Grep, which searches for lines in a file that matches a particular pattern. The *map* function takes in a line number and line as its key-value pair, and outputs the line only if the line matches the pattern. The output key in this case is arbitrary - it may be the same arbitrary value for everything (like 1), or for more information, it may just be the input key (the line number). The reduce function is trivial - it is simply the identity function. Therefore, the output of the whole MapReduce task for Distributed Grep is a list of key-value pairs, mapping the line number (or 1) to matching lines from the file.

Both of these examples have some common themes, which differ from our own use of MapReduce:

1. They take advantage of the framework and programming abstraction to define very simple, elegant code.

2. They perform very simple operations over extremely large sets of data.

3.3 Hadoop

Hadoop [16] is an open-source implementation of the MapReduce framework. All of our code was written for and evaluated with Hadoop version 0.20.1. It provides a great deal of customization, including how to transform an input file into a list of key-value pairs to apply *map* and *reduce* to, as well as more systems-level concerns like splitting the input file at the byte level to determine how many *map* processes to be run and how many *map* tasks to give to each process.

4 Parallelizing TSP on MapReduce

As mentioned above, the approach we have chosen to take towards solving TSP is local search - we want a very good solution in a short amount of time, but we don't care about proving how close our solution is to the optimal solution (whether it *is* the optimal solution, or whether it is within a certain factor of the optimal). Even for the same serial algorithm, there can be multiple ways to parallelize it, even on such a simple framework as MapReduce.

4.1 Parallelizing Tabu Search

Since Tabu search at a very high level involves evaluating a bunch of possible moves, picking a good one, making it, and then repeating, there seems to be an easy translation into MapReduce:

1. Map: Have each mapper evaluate a different move (which is not in the tabu list), returning a description of the move and a value indicating by how much it would change the objective value.

2. Reduce: Compare all the moves returned by the various mappers, and apply the move that decreases the objective value the most.

3. Repeat.

This approach has a large flaw, however: the startup time of MapReduce is non-negligible, so repeatedly starting up MapReduce enough times to make enough moves for Tabu search to be effective would eat a lot of time. A Tabu search algorithm will commonly make tens of thousands or even up to millions of small moves - the overhead of starting many processes, setting up communication with them, and interacting with data in a distributed file system would be unacceptable for this many iterations.

With this in mind, we want to minimize the amount of MapReduce iterations we need to do. Since a Tabu search relies on its small, incremental moves exploring a lot of the best parts of the solution space, it would be worth having each mapper explore a lot of the solution space on its own, and then the reducer could determine the final solution based on the results of all the mappers:

1. Map: Have each mapper pick a random starting solution, and run its own instance of Tabu search from there. After a while, return the best solution found during this process.
2. Reduce: Compare the results returned by all the Mappers, and output the solution with the lowest objective.

This is the approach to parallelizing Tabu search that we focused our evaluation on. It is much better than the approach that relied on multiple iterations of MapReduce itself - we only suffer through the MapReduce overhead once, but we still manage to take advantage of as many machines as we have to look at as much of the solution space as possible.

4.2 Parallelizing Large Neighborhood Search

Large Neighborhood Search, at a high level, involves splitting up a tour into multiple pieces, optimizing each one separately, and then gluing these pieces back together to form a better solution than what we had before. This is practically begging for MapReduce already, with no modification needed:

1. Generate a random tour.
2. Partition the tour into pieces that overlap only at their endpoints.
3. Map: Have each mapper optimize one of the pieces from the partition of the initial tour, preserving the beginning and ending cities for the segment. Return that re-optimized segment.
4. Reduce: Use the knowledge that each segment overlaps with two others, just at its beginning and ending points, to glue the tour back together.
5. Return this tour.

This is a pretty satisfying approach - it doesn't incur too much overhead from MapReduce startup, it is keeping each mapper busy with computation that is guaranteed not to be redundant, and the reducer is doing a bit more than just finding the maximum element in a list. However, it doesn't explore as much of the solution space as we might like: one clear example of this is that when a random tour is partitioned to be optimized in pieces, the two endpoints in a segment cannot possibly end up adjacent in the final tour. Even if we were to modify our algorithm a bit by making the segments truly disjoint, not forcing the endpoints of segments to be stationary, and performing a more intelligent rearrangement of the segments in the reducer, we are still enforcing a meta-structure on the solution that constrains our possible solution space.

With this in mind, we have an alternative algorithm, which may seem very familiar:

1. Generate a random tour.

2. Partition the tour into pieces that overlap only at their endpoints.
3. Map: Have each mapper optimize one of the pieces from the partition of the initial tour, preserving the beginning and ending cities for the segment. Return that re-optimized segment.
4. Reduce: Use the knowledge that each segment overlaps with two others, just at its beginning and ending points, to glue the tour back together.
5. Repeat steps 2-4 with the resulting tour some number of times.
6. Return the resulting tour.

For each iteration of MapReduce, the resulting tour from the previous step (with a random tour to start) is partitioned differently and re-optimized. But wait - didn't we already throw out the idea of running multiple iterations of MapReduce when we were talking about Tabu search? Sort of: for Tabu search, we would have required tens of thousands of iterations, and the number of iterations would have grown with the TSP instance size, but with this approach to LNS, we would only iterate a small constant number of times to make sure we are truly exploring a large enough portion of the solution space. For a larger instance, we would just perform more computation in each individual mapper, instead of firing up MapReduce more times. This is the approach to parallelizing Large Neighborhood Search that we focused our evaluation on.

4.3 Differences From Normal Use of MapReduce

As we mentioned before, we are using MapReduce very differently from the ways it is normally used. MapReduce is built to process extremely large amounts of data, but typically this processing is very simple: arithmetic, string parsing, or something similar. We are turning these typical uses upside down: our input is always very small (represented in a few kilobytes), but our processing of this input is actually an attempt to solve an NP-hard problem.

In order to take advantage of the parallelism that the MapReduce framework can offer us, we needed to convince Hadoop that even though our input is only a few kilobytes in size, it was actually our intent to fire up expensive JVM processes on multiple machines rather than just having one process take care of everything. To do this, it was very important to understand how Hadoop translates a flat input file into a distributed computation.

When we talk about splitting up an input file, there are actually two different splits we need to worry about: `Records` and `InputSplits`. The split that is commonly considered is the `Record` - this is the key-value pair that is passed into *map*, such as a line number and a line of text. These `Records` serve a purpose that is entirely semantic. Since MapReduce is a data-oriented framework, it doesn't care about what your data means semantically; MapReduce cares about the chunks of data as raw bytes, living in various replicated points in a distributed file system.

This is where the `InputSplits` come in. These are the splits that are made according to raw byte-level concerns, such as where the bytes are physically located in the underlying distributed file system. Every example MapReduce use that we came across uses a subtype of the `FileInputFormat` to split the data at the byte level. This splits the input according to HDFS blocks, which are on the order of 64MB. Even `TextInputFormat`, which claims to split the input line by line, is a type of `FileInputFormat` that creates `InputSplits` of size 64MB, and creates `Records` that each represent one line of the input. When it comes time to distribute the computation, each process receives one `InputSplit`, and that process breaks its `InputSplit` into one or more `Records`, each of which is processed by the mapping function.

Not knowing this, we were very confused when even after putting copies of the TSP instance on multiple lines in the input file and selecting the `TextInputFormat` to distribute our data, we were seeing no benefits from parallelism, whether we were running 4 or 400 mappers! After discovering how Hadoop

Instance	Serial	NM=4	NM=6	NM=8	NM=10	NM=12	NM=14	NM=16
pr439	1.59	1.65	1.64	1.65	1.65	1.65	1.67	1.64
rl1304	2.4	2.58	2.67	2.57	2.63	2.56	2.58	2.56
pcb3038	3.19	3.48	3.42	3.38	3.43	3.4	3.41	3.33

Table 1: The table shows the ratio of the objective achieved to the optimal objective value for the *Tabu Search* algorithm as the parallelization steadily increases. NM stands for number of mappers.

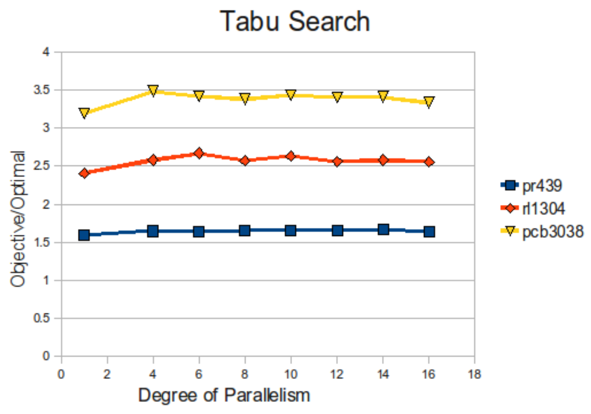
Instance	Serial	NM=4	NM=6	NM=8	NM=10	NM=12	NM=14	NM=16
pr439	1.07	1.06	1.08	1.1	1.13	1.11	1.12	1.1
rl1304	1.16	1.11	1.11	1.11	1.12	1.12	1.14	1.14
pcb3038	1.15	1.12	1.13	1.11	1.11	1.11	1.11	1.1

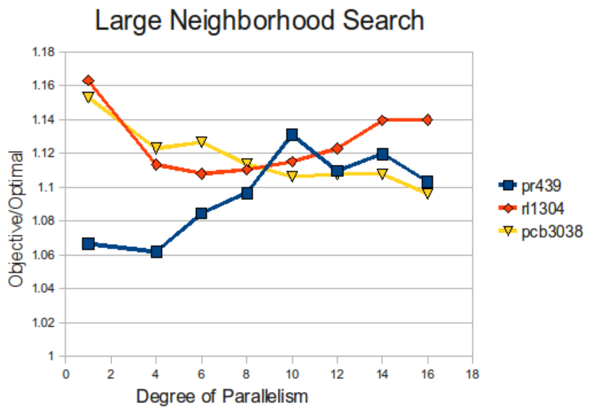
Table 2: The table shows the ratio of the objective achieved to the optimal objective value for the *LNS* algorithm as the parallelization steadily increases. NM stands for number of mappers.

splits the data differently at the semantic level and the system level, we decided we needed to write our own `InputFormat`. The semantics of the `TextInputFormat` were correct: each `Record` should correspond to a line in the input file, on which we would store a copy of the TSP instance. However, we wanted to have more than one `InputSplit` for our data, so that our computations could run simultaneously on more than one machine. Instead of creating `InputSplits` of size 64MB, our `LineInputFormat` class creates `InputSplits` that match the `Readers`: one per line. Thus to achieve parallelism with n mappers, we simply had to create a file with n copies of the TSP instance, and our `LineInputFormat` would handle the rest.

Even this, though, was not very satisfying. Why was it necessary to make so many copies of the input? With the power we just discovered by defining our own `InputFormat` class, it would be easy to prevent this unnecessary duplication. Our new class, `TspInputFormat`, operates on a file of a single line (the TSP instance), generates an `InputSplit` that contains this entire line, and makes n copies of this one `InputSplit` to give to n mappers, which each create one line-oriented `Record` out of the split. This only saves us a few kilobytes in terms of storage and communication, but it saves us from copying data.

5 Evaluation





We ran our experiments with the Serial Version of the algorithm on machines running Intel Core 2 Quad Q6600 2.4 Ghz processors with 3 GB of RAM. We ran our MapReduce experiments on the Brown CS department's 16-node Hadoop cluster which uses the same type of machines. For both the Tabu and LNS heuristics, instead of looping for a fixed number of iterations as specified in Section 2, we looped until we hit a time limit. We ran experiments for three common instances from TSPLIB [18] - pr439 with 439 cities, rl1304 with 1304 cities and pcb3038 with 3038 cities.

Tabu Search For the Tabu search experiments, we gave each Map instance a time limit of 300s. However, since we observed that each Map iteration incurred a runtime of 24s, we gave the Serial version a time limit of 324s to account for that. The results for both the serial and the parallel version are given above. Except for the rl1304 TSP instance, parallelization has almost no effect on the final objective. The reason the serial version performs somewhat better is because it has an extra 24s in which to run the tabu search compared to the MapReduce version. Investigating a bit further, we found that the reason the MapReduce version didn't fare better was because there was very little variance in the final objective over several runs of the Tabu Search algorithm - the standard deviation was approximately 1.0-2.5% for the various TSP instances. This means that running multiple instances of Tabu Search in parallel would have yielded almost the same final objectives for each of them giving no advantage to the parallel version compared to the serial version.

Large Neighborhood Search For the Large Neighborhood Search experiments, we gave each Map instance a time limit of 300s and ran 5 iterations of MapReduce for each run. After accounting for the overhead, we gave a time limit of 1630s to the serial version. The results are given above. The standard deviation was 1.0-2.0% for the various TSP instances. As can be seen above, the MapReduce version performed very slightly better than the serial version. What's happening is that after a few minutes, LNS gets much less effective and the tour length hardly decreases. This means that doing multiple iterations whether it's in parallel or in serial doesn't help at all. This is why the results for LNS over MapReduce are about the same as the results for the serial version. What we think might be helpful is running the MapReduce version of LNS over a much bigger TSP instance with a lot more Map nodes. However, since we had access to only 16 Map nodes, such an evaluation was not possible.

6 Conclusions

We presented various algorithms that are used to solve TSP and parallelized two of them over MapReduce. We evaluated the effect of parallelization for the algorithms. While the results for Tabu Search were in accordance with our intuition, we were surprised by the results for LNS as we expected the MapReduce version to perform better. We think that parallelization of Ant Colony and Genetic Algorithms over MapReduce, possibly over MapReduce Online [5] to reduce overhead would be fruitful directions to explore.

References

[1] David L. Applegate, Robert E. Bixby, Vasek Chvta, William J. Coo. The Traveling Salesman Problem: A Computational Study. *Princeton University Press*, 2006.

[2] Sanjeev Arora. Polynomial-time Approximation Schemes for Euclidean TSP and other Geometric Problems. *Journal of the ACM 45(5)*, 753–782, 1998.

[3] Ranieri Baraglia, Jose I. Hidalgo, Raffaele Perego. A Parallel Hybrid Heuristic for the TSP. *Proceedings of the 1st European Workshop on Evolutionary Computation in Combinatorial Optimization*, 193–202, 2001.

[4] Giovanni Cesari. Divide and conquer strategies for parallel TSP heuristics. *Computers and Operations Research*, Vol. 23, Issue 7, Pages 681–694, 1996.

[5] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, Russell Sears. MapReduce Online. *UCB/EECS-2009-136*, 2009.

[6] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Sixth Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[7] Marco Dorigo, Luca M. Gambardella. Ant Colonies for the Traveling Salesman Problem. 1997.

[8] Claude N. Fiechter. A parallel tabu search algorithm for large traveling salesman problems. *Discrete Applied Mathematics*, Vol. 51, Issue 3, Pages 243–267, 1994.

[9] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, Vol. 13, Issue 5, Pages 533–549, 1986.

[10] Randy L. Haupt, Sue E. Haupt Practical Genetic Algorithms. *Wiley-Interscience*, 1997.

[11] Holger H. Hoos, Thomas Stutzle. Stochastic Local Search: Foundations and Applications. *Morgan Kaufmann* 2005.

[12] Gerard A. P. Kindervater, Jan Karel Lenstra, David B. Shmoys. The parallel complexity of TSP heuristics. *Journal of Algorithms*, Vol. 10, Issue 2, Pages 249–270, 2004.

[13] John Knox. Tabu search performance on the symmetric traveling salesman problem. *Computers and Operations Research*, Vol. 21, Issue 8, Pages 867–876, 1994.

[14] Miroslaw Malek, Mohan Guruswamy, Mihir Pandya and Howard Owens. Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem. *Annals of Operations Research*, Vol. 21, Issue 1, Pages 59–84, 1989.

[15] Shigeyoshi Tsutsui, Noriyuki Fujimoto. Parallel Ant Colony Optimization Algorithm on a Multi-core Processor. *Lecture Notes in Computer Science* Vol. 6234/2010, 488–495, 2010.

[16] Apache Hadoop. http://hadoop.apache.org/.

[17] Apache Hadoop Documentation v0.20.1. http://hadoop.apache.org/common/docs/r0.20.1/.

[18] TSPLIB: A Library of Sample Instances for the TSP. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.