PARALLELIZING ANT COLONY OPTIMIZATION FOR TSP OVER HADOOP MAP-REDUCE

1.1 The Travelling Salesman Problem

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

The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept *city* represents, for example, customers, soldering points, or DNA fragments, and the concept *distance* represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder.

In the theory of computational complexity, the decision version of the TSP (where, given a length L, the task is to decide whether any tour is shorter than L) belongs to the class of NP-complete problems. Thus, it is likely that the worst case running time for any algorithm for the TSP increases exponentially with the number of cities.

1.2 Solution Goals

Generally, for a TSP solver, one either tries to obtain a provably optimal solution of 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 an algorithm termed the

Ant Colony Optimization algorithm that simulates the way ants find the shortest route to a food source. There already exist sequential versions of this algorithm. We aim to parallelize this algorithm over Hadoop MapReduce and thereby improve its performance.

1.4 MapReduce

MapReduce 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.

1.4.1 Framework

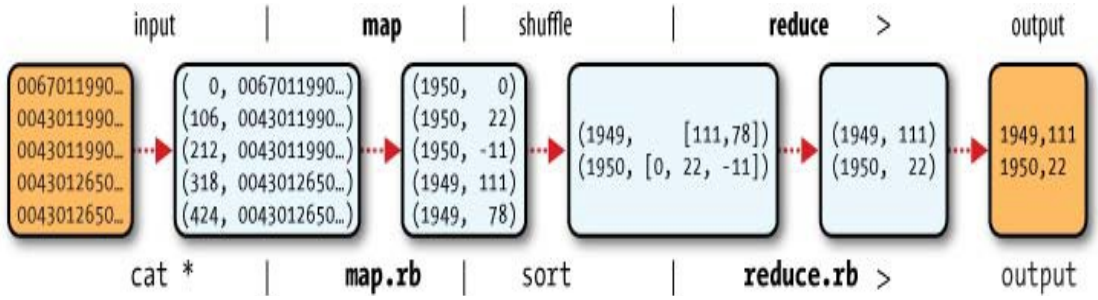

Figure 1.5 – Example data flow in MapReduce Program

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.

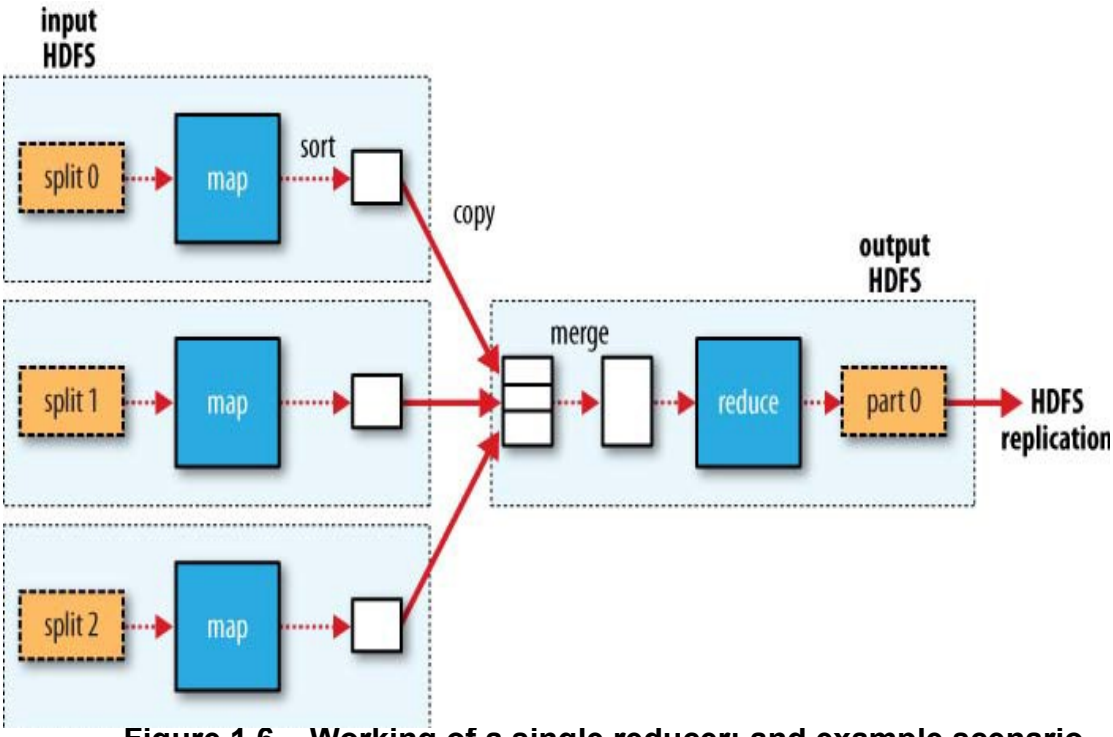

Figure 1.6 – Working of a single reducer; and example scenario

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.

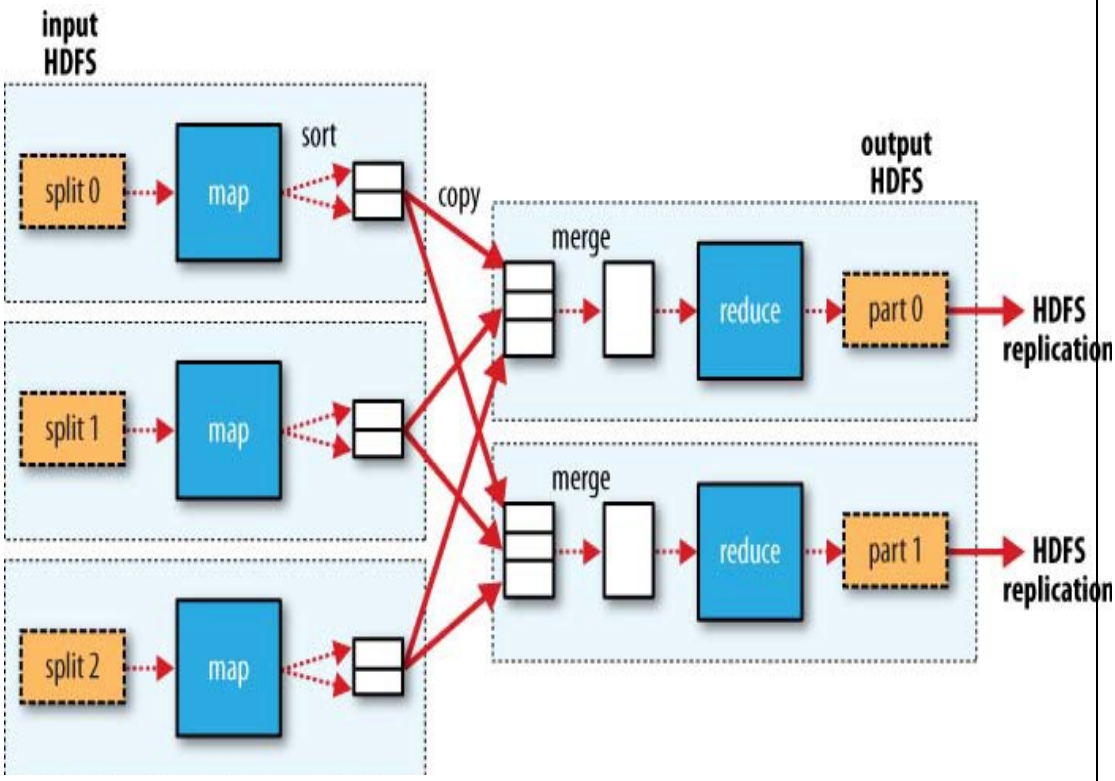

Figure 1.7 – Multiple reducers in action

1.4.2 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.

1.5 Hadoop

Hadoop is an open source framework for writing and running distributed applications that process large amounts of data. Distributed computing is a wide and varied field, but the key distinctions of Hadoop are that it is:

1) *Accessible*—Hadoop runs on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).

2) Robust—Because it is intended to run on commodity hardware, Hadoop is architected with the assumption of frequent hardware malfunctions. It can gracefully handle most such failures.

3) Scalable—Hadoop scales linearly to handle larger data by adding more nodes to the cluster.

4) Simple—Hadoop allows users to quickly write efficient parallel code.

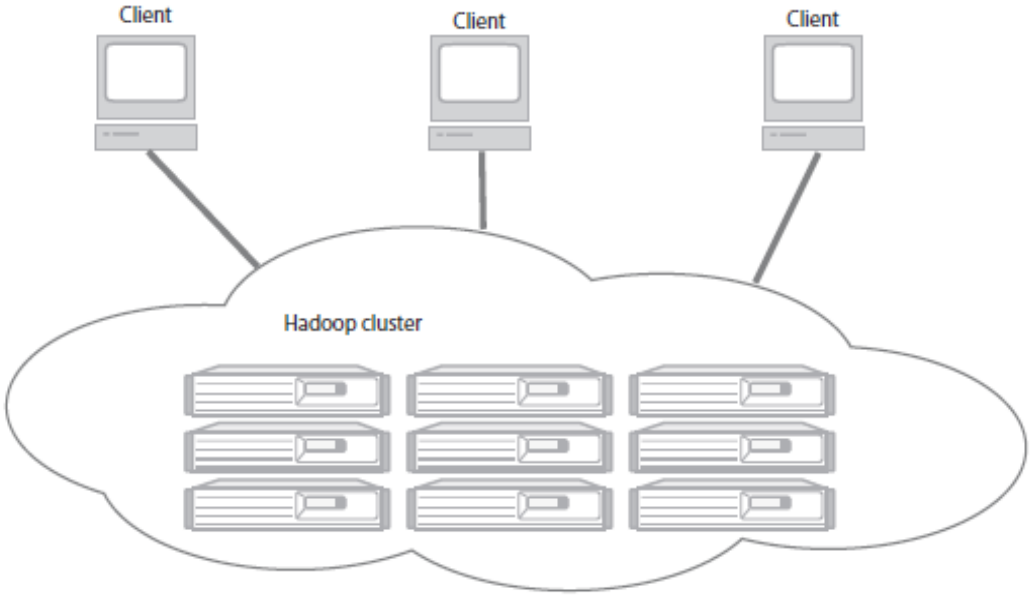

Figure 1.1 A Hadoop cluster has many parallel machines that store and process large data sets. Client computers send jobs into this computer cloud and obtain results.

Figure 1.8 – A Hadoop Cluster

Hadoop's accessibility and simplicity give it an edge over writing and running large distributed programs. Even college students can quickly and cheaply create their own Hadoop cluster . On the other hand, its robustness and scalability make it suitable for even the most demanding jobs at Yahoo and Facebook. These features make Hadoop popular in both academia and industry.

Figure 1.8 illustrates how one interacts with a Hadoop cluster. As you can see, a Hadoop cluster is a set of commodity machines networked together in one location. Data storage and processing all occur within this "cloud" of machines . Different users can submit computing "jobs" to Hadoop from individual clients, which can be their own desktop machines in remote locations from the Hadoop cluster.

1.5.1 The building blocks of Hadoop

On a fully configured cluster, “running Hadoop” means running a set of daemons, or resident programs, on the different servers in your network. These daemons have specific roles; some exist only on one server, some exist across multiple servers. The daemons include:

1) NameNode
2) DataNode
3) Secondary NameNode
4) JobTracker
5) TaskTracker

We’ll discuss each one and its role within Hadoop.

1) Namenode

Let’s begin with arguably the most vital of the Hadoop daemons—the NameNode, Hadoop employs a master/slave architecture for both distributed storage and distributed computation. The distributed storage system is called the *Hadoop File System*, or HDFS. The NameNode is the master of HDFS that directs the slave DataNode daemons to perform the low-level I/O tasks. The NameNode is the bookkeeper of HDFS; it keeps track of how your files are broken down into file blocks, which nodes store those blocks, and the overall health of the distributed filesystem.

The function of the NameNode is memory and I/O intensive. As such, the server hosting the NameNode typically doesn’t store any user data or perform any computations for a MapReduce program to lower the workload on the machine. This means that the NameNode server doesn’t double as a DataNode or a TaskTracker.

There is unfortunately a negative aspect to the importance of the NameNode—it’s a single point of failure of your Hadoop cluster. For any of the other daemons, if their host nodes fail for software or hardware reasons, the Hadoop cluster will likely continue to function smoothly or you can quickly restart it. Not so for the NameNode.

2) Datanode

Each slave machine in your cluster will host a DataNode daemon to perform the grunt work of the distributed filesystem—reading and writing

HDFS blocks to actual files on the local filesystem. When you want to read or write a HDFS file, the file is broken into blocks and the NameNode will tell your client which DataNode each block resides in. Your client communicates directly with the DataNode daemons to process the local files corresponding to the blocks. Furthermore, a DataNode may communicate with other DataNodes to replicate its data blocks for redundancy.

DataNodes are constantly reporting to the NameNode. Upon initialization, each of the DataNodes informs the NameNode of the blocks it's currently storing. After this mapping is complete, the DataNodes continually poll the NameNode to provide information regarding local changes as well as receive instructions to create, move, or delete blocks from the local disk.

3) Secondary Namenode

The Secondary NameNode (SNN) is an assistant daemon for monitoring the state of the cluster HDFS. Like the NameNode, each cluster has one SNN, and it typically resides on its own machine as well. No other DataNode or TaskTracker daemons run on the same server. The SNN differs from the NameNode in that this process doesn't receive or record any real-time changes to HDFS. Instead, it communicates with the NameNode to take snapshots of the HDFS metadata at intervals defined by the cluster configuration.

As mentioned earlier, the NameNode is a single point of failure for a Hadoop cluster, and the SNN snapshots help minimize the downtime and loss of data. Nevertheless, a NameNode failure requires human intervention to reconfigure the cluster to use the SNN as the primary NameNode.

4) Job Tracker

The JobTracker daemon is the liaison between your application and Hadoop. Once you submit your code to your cluster, the JobTracker determines the execution plan by determining which files to process, assigns nodes to different tasks, and monitors all tasks as they're running. Should a task fail, the JobTracker will automatically relaunch the task, possibly on a different node, up to a predefined limit of retries.

There is only one JobTracker daemon per Hadoop cluster. It's typically run on a server as a master node of the cluster.

5) Task Tracker

As with the storage daemons, the computing daemons also follow a master/slave architecture: the JobTracker is the master overseeing the overall execution of a MapReduce job and the TaskTrackers manage the execution of individual tasks on each slave node. Figure 1.2 illustrates this interaction.

Figure 1.9 - Job Tracker and Task Tracker interaction

Each TaskTracker is responsible for executing the individual tasks that the JobTracker assigns. Although there is a single TaskTracker per slave node, each TaskTracker can spawn multiple JVMs to handle many map or reduce tasks in parallel.

1.5.2 Anatomy of a MapReduce program

As we have mentioned before, a MapReduce program processes data by manipulating (key/value) pairs in the general form

map: (K1,V1) → list(K2,V2)

reduce: (K2,list(V2)) → list(K3,V3)

Figure 1.10 - The anatomy of a MapReduce Program

2. FEASIBILITY STUDY

2.1 PRELIMINARY STUDY

Travelling Salesman Problem (TSP) is a famous problem in algorithm, which asks for the shortest route that passes all cities once and only once, and finishes up at the starting city, where the length of the route and distance travelled is calculated as Euclidean distance between the coordinates of the cities.

TSP is a problem with exponential complexity, and proven to be NP-complete. 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 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. We want a very good solution in a short amount of time, but we don't care whether it is the optimal solution, or whether it is within a certain factor of the optimal. Following are the common local search algorithms used to solve TSP.

a) Tabu search

The idea is that one defines a set of moves one can make to change the tour. 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.

b) Ant Colony Optimization

In this 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.

c) Genetic algorithms

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.

Among the above approaches the ACO algorithms are the most successful and widely recognized algorithmic techniques based on ant behaviors. Their success is evidenced by the extensive array of different problems to which they have been applied, and moreover by the fact that ACO algorithms are for many problems among the currently top-performing algorithms.

They have an advantage over simulated annealing and genetic algorithm approaches of similar problems when the graph may change dynamically; the ant colony algorithm can be run continuously and adapt to changes in real time.

2.2 EXISTING SYSTEM

Brute force algorithms can be used to obtain the optimal solutions for the TSP problems comprising of a very few nodes but this becomes impractical as the number of the nodes increases just above 20 nodes.

The existing solution to the TSP problem is based on the sequential implementation of the local search algorithms. The local search algorithm used in the project is the Ant Colony Optimization algorithm. The basic idea is to use artificial ants (threads). By the use of probabilistic rules based on local information they can find the shortest path between two points in a graph.

This algorithm is a reproduction of the real-life behavior of ants. Ants are quite smart in finding their way between their colony and their food source. A lot of worker "drones" are walking through the near environment and if they find some food, they lay down a pheromone trail.

Some of the other ants are still searching for other ways, but the most of them are following the pheromone trail make this way more attractive. But over time the pheromone trail is starting to decay/evaporate, it is losing its attractiveness. Due to the time component, a long way has a very low density of the pheromone, because the pheromone trail along the long way is evaporating. Thus a very short way has a higher density of pheromones and is more attractive to the workers leading into a kind of optimal path for our graph problem.

In the TSP problem, the ants are dropped on random vertices in our graph. Each ant is going to evaluate the best way for his next move to another vertex, based on a probabilistic formula.

The whole thing works like this:

1. Initialize the best distance to infinity
2. Choose random vertices in the graph to plant the ants
    1. Let the ants work their best paths using the probabilistic formulas
    2. Let the ants update the pheromones on the graph
3. If the worker returned with a new best distance update the currently best distance
4. Start from step 2 until we find a best way or we have reached our maximum workers.
5. Return or output the best distance

2.3 PROBLEMS PERSISTING IN THE PRESENT SYSTEM

Current single-system implementation of Ant Colony Optimization yields results close to two minutes for a 52-node TSP instance. Cluster based and Multi-core based implementations of ACO yield better results. ACO for a 52-node TSP instance on a 4-thread Intel Core i5 processor yields the result in about 22 seconds.

Though these values seem feasible for a structure of 52 nodes, the implementation becomes useless when the number of nodes is increased to about 500 nodes. The time for execution of such an instance is in the order of >30 minutes.

Also the best results for the implementation are found only when the vertices randomly chosen are done on the basis of a normal distribution.

Figure 2.1

No. Of Cities	[X-axis]	52	76	101	127	150	198
Time (sec)	[Y-axis]	146	331	584	924	1287	2443

The line graph in figure 2.3.1 represents the exponential rise of the execution time of the Ant Colony Optimization algorithm for different number of cities given as the input.

2.4 PROPOSED SYSTEM

The proposed system is based on parallelizing the sequential version of Ant Colony optimization algorithm over a cluster. This is done based upon the map-reduce framework. MapReduce is a data processing model which works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the map function and the reduce function. Decomposing a data processing application into mappers and reducers is sometimes nontrivial.

The main advantage of using MapReduce framework is that once an application is written in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change.

In the proposed system the modified algorithm would be given as a job to cluster implemented using Hadoop Map/Reduce and hence the cluster with parallel computation capabilities would be able to provide a more efficient and quicker result to the TSP using the ACO algorithm thereby decreasing the amount of time required to obtain a solution compared to its sequential existing versions.

2.4.1 DIFFERENT METHODS OF PARALLELIZING

There are different possible methods to parallelize the algorithm. They are explained below.

1. Multiple sequential nodes

Here, we have multiple nodes, each of which executes the algorithm in a sequential manner. The results computed by each node is then combined and the best path is then chosen.

We tested this algorithm with various inputs and plotted the results in the graph shown in Figure 2.2.

The above figure depicts the running time of the algorithm when the number of systems in the cluster is one, two and five. From observing the graph it can be inferred that as the number of nodes in the cluster increases the running time decreases by a considerable amount.

2. Multiple agents

Here, we model each node to act as an agent, computing one path through the graph based on the input pheromone array. After the agents have finished finding the paths, the results of all the agents are sent to the reducer which computes the best path for that stage. The system can be implemented in multiple stages, where each stage uses the pheromones computed by the previous stage as their input.

3. Multiple sequential nodes, multi stages

Here, we combine the above two models. First, we model each node as a sequential implementation of the algorithm, accepting the input graph and the initial pheromone levels, and computing the optimum result in the way a sequential algorithm would.

Next, we combine the results of different nodes and reduce it to the result of the stage. Depending on the number of stages, the result would then be fed

to the next stage as input and the process would continue until the final stage, where the best result is calculated by the reducer and displayed.

3. DESIGN

The design of the entire project involves the designing of the cluster as well as designing the Map/Reduce job. Designing the cluster involves finding answers for questions like what the software and hardware requirements are for each system as well as the requirements of the networking architecture. The software and hardware requirements for each system as well as for the network are given below.

3.1 SYSTEM REQUIREMENTS:

3.1.1 SOFTWARE REQUIREMENTS

Required software for Linux and Windows include:

- The distro used for the project is Ubuntu 11.04.
- Java 1.6.x, preferably from Sun, must be installed.
- Ssh must be installed and sshd must be running to use the Hadoop scripts that manage remote Hadoop daemons.

Additional requirements for Windows include:

- Cygwin- required for shell support in addition to the required software above.

3.1.2 HARDWARE REQUIREMENTS

- 2 GHz Intel Pentium Dual Core processor
- 1 GB DDR2 RAM
- 160 GB Hard Disk

3.1.3 NETWORK REQUIREMENTS

For this particular project the cluster setup includes 4 individual systems that are networked together using high-speed LAN. One of the computers will be given the responsibility of the Namenode and all the

computers including the Namenode act as the Datanodes. One random system will also be assigned the responsibility of the Secondary Namenode. The above daemons work on the data that is spread across the distributed file system. Regarding the job assignment and the job implementation, there is one Job Tracker for the entire cluster which is present in the master node and one Task Tracker for each slave system. This is the design details for the network cluster.

An HDFS cluster has two types of node operating in a master-worker pattern: a *namenode* (the master) and a number of *datanodes* (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts. A *client* accesses the filesystem on behalf of the user by communicating with the namenode and datanodes. The client presents a POSIX-like filesystem interface, so the user code does not need to know about the namenode and datanode to function. Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

Without the namenode, the filesystem cannot be used. In fact, if the machines running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, it is important to make the namenode resilient to failure, and Hadoop provides two mechanisms for this. The first way is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFS mount. It is also possible to run a secondary namenode, which despite its name does not act as a namenode. Its main role is to periodically

merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode's metadata files that are on NFS to the secondary and run it as the new primary.

	3
5	
1	
1	**4**
	2

5	**3**
2	
4	**1**

Figure 3.1

The design of the Map/Reduce job basically involves how the Map and Reduce functions are implemented and how the Ant Colony Optimization algorithm has been modified so as to be able to run in a Map/Reduce framework. As far as the Map and Reduce are concerned, the entire job is divided into one initial Map/Reduce phase, several intermediate Map/Reduce phases and a final Map/Reduce phase. Each phase produces a result which is expected to be closer to the optimum value. The number of stages is user controlled.

Figure 3.2

The Ant Colony Optimization algorithm itself is divided into different modules by defining different classes for the different components of the algorithm namely there is the Class Ant that represents an individual ant that is used to find a route in the graph, the Class Config that is used to set the values for parameters ALPHA and BETA and to initialize the graph details, Class Graph that represents the graph on which the algorithm has to be applied, the Class Pheromone that is responsible for reading pheromone values and updating the value of the same after each stage, Class WalkedWay that represent the solution produced by one ant, Class TSPInput that represents the input to the Map/Reduce job and other than all these classes there are the Mapper and Reducer classes for the initial, intermediate and final stages each namely the Class TspMapper, Class TspReducer, Class TspInterMapper, Class TspInterReducer, Class TspFinalMapper and Class TspFinalReducer. Also there are the three classes for the three respective Map/Reduce stages which are basically used to define each of those job stages namely Class TspHadoop, TspInterHadoop and Class TspFinalHadoop.

3.2 DATA FLOW DIAGRAM

Figure 3.3 represents the Data Flow Diagram of the flow of execution. The connected graph is given as the input as a file to the first stages of the Map/Reduce where the multiple mapper states processes the given data. Here the mappers implement the working of the probabilistic algorithm, the Ant Colony Optimization in which the behavior of a group of ants in a colony is simulated. In this behavior ants try to find food by following random paths and following those paths with the highest amount of pheromone deposits in it with

the assumption that the previous ant that had used this path had found the correct route to the food and back. When more number of ants choose the particular root one by one the amount of pheromone deposit keeps on increasing and the hence the probability of the path being the correct one. Thus we say that this algorithm is a heuristic one because it depends on assumptions and probability. Each mapper state simulates multiple ants in a sequential manner and the routes found out by each ant is collected and stored for forwarding to the reducer. So as much number of routes is produced as there are number of ants. This number of ants can be determined by the programmer. Also the number of mappers to be executed per Map/Reduce stage can also be controlled by the programmer. The result of the processing is a sequence of values consisting of shortest routes from each mapper states. These values are passed to the reducer of the same stage where further processing resumes and as a result the best route among all from the current stage is chosen and also the pheromone deposits are also updated using the routes produced by all mappers. These two data are passed to the next stage of Map/Reduce where the next set of mappers takes the two data as inputs for processing and this intermediate Map/Reduce stage is repeated to a predetermined number of times and in the final stage the final reducer produces the best route (shortest route) as the result and this result is stored in the master node. Also during the completion of every reducer process the result then produced is displayed on the output device.

The connected graph is given as the input as a file to the first stages of the Map/Reduce where the multiple mapper states processes the given data. Here the mappers implement the working of the probabilistic algorithm, the Ant Colony Optimization in which the behavior of a group of ants in a colony is simulated. In this behavior ants try to find food by following random paths and following those paths with the highest amount of pheromone deposits in it with the assumption that the previous ant that had used this path had found the correct route to the food and back. When more number of ants choose the particular root one by one the amount of pheromone deposit keeps on increasing and the hence the probability of the path being the correct one. Thus we say that this algorithm is a heuristic one because it depends on

assumptions and probability. Each mapper state simulates multiple ants in a sequential manner and the routes found out by each ant is collected and stored for forwarding to the reducer. So as much number of routes is produced as there are number of ants. This number of ants can be determined by the programmer. Also the number of mappers to be executed per Map/Reduce stage can also be controlled by the programmer. The result of the processing is a sequence of values consisting of shortest routes from each mapper states. These values are passed to the reducer of the same stage where further processing resumes and as a result the best route among all from the current stage is chosen and also the pheromone deposits are also updated using the routes produced by all mappers. These two data are passed to the next stage of Map/Reduce where the next set of mappers takes the two data as inputs for processing and this intermediate Map/Reduce stage is repeated to a predetermined number of times and in the final stage the final reducer produces the best route (shortest route) as the result and this result is stored in the master node. Also during the completion of every reducer process the result then produced is displayed on the output device.

3.3 SCHEMATIC REPRESENTATION:

The following schematic representation illustrates the flow of computation in a Hadoop cluster.

4. IMPLEMENTATION

In the sequential implementation of the Ant Colony Optimization algorithm each artificial ant is placed in a randomly chosen node. The graph initially contains the initial pheromone levels. The ants initially traversing the graph select the path they travel on the basis of the attractiveness (reciprocal of the distance of the edge) of an edge. As they traverse through the graph they deposit the pheromones, thereby making different pheromone levels at different edges of the graph. Now the consequent ants choose their path by considering the amount of the pheromones deposited and the attractiveness of each path. Also the pheromone deposits in an edge evaporate or decrease when this edge is not used. In This way the pheromone deposit goes on increasing in the path through which most ants travel and becomes the path for the travelling salesman with a distance close to the optimal solution.

We modified the existing sequential version of the ACO algorithm to fit into the MapReduce framework of Hadoop. The basic idea is to give copies of the TSP instances to multiple mappers which the work in parallel to produce the results. The main difficulty in implementing the algorithm in Hadoop was regarding the pheromone array which needed to be shared between the working ants. The initial plan was to consider each mapper as a single ant that will use a global pheromone array to calculate the routes. But in Hadoop, all the mappers must work on independent data and so the sharing of pheromone updates was not possible between the mappers. So, in order to implement this algorithm in Hadoop, we decided to use the concept of chaining MapReduce jobs. Chaining means executing multiple MapReduce jobs one after the other.

In this method, there will be multiple stages of MapReduce. In the first stage, the mappers will work on the input data using the initial pheromone levels. There is no sharing of pheromone updates among the mappers. Once the mappers have finished their work, they'll pass on the calculated routes to the reducer which will then update the pheromone values. The results are

then passed on to the next stage. Therefore, the mappers of the intermediate stages will get to work on the updated pheromone values. The number of intermediate stages required can be chosen by the user.

Once all the stages are complete, the final stage will give the best result obtained among all stages. In order to further improve the performance of the parallel version, we decided to implement multiple ants inside a single mapper. These ants will work just like in the sequential version as they will have access to the pheromone updates produced by each other instantaneously.

The detailed working of each of the different stages along with the working of the driver functions, mappers and reducers are explained below.

4.1 STAGES OF IMPLEMENTATION

4.1.1 INITIAL STAGE

Before a MapReduce program can be run, it needs to be fully configured. This involves specifying the input and output directories, the mapper class, the reducer class and various other properties with regard to the MapReduce job.

The basic algorithm for the initial stage of our MapReduce implementation is as follows:

i) Create a MapReduce job object of type JobConf and initialize it with the job name and the driver class. Driver class is the class that configures the JobConf object and starts the MapReduce job. Quite often this is the same class in which the object is defined.
ii) Use the member functions of the JobConf class to set the input and output directories for the job.
iii) Also set the InputFormat and OutputFormat for the job. After that specify the mapper and reducer classes.
iv) Once all the parameters for the job have been specified, start the job by calling JobClient.runJob (conf).

v) Once the stage is completed, call the driver function for the next stage. Intermediate stages are run as many times as specified by the user.
vi) After all the stages are complete, display the final result and the total time taken.

In the first stage, the input to the MapReduce job is the TSPLIB input file that contains the Euclidian 2D coordinates of a TSP problem. Once the job has started, the JobClient calls the getSplits() method of the InputFormat that returns the input splits generated from the input file. Next it calls the getRecords() method of the RecordReader that will return the records from the input file. We use the NLineInputFormat that is available in Hadoop. This input format ensures that each map task gets exactly n lines of the input. Our input file contains m lines, where each line holds the single TSP instance i.e. the coordinates are replicated m times in the file. NLineInputFormat generates key-value pairs of type <LongWritable, Text> which is then passed on to each mapper. The functioning of the mapper is as follows.

i) Each map() obtains the key-value pair of type <LongWritable, Text> from the RecordReader. Here the value of type Text is one instance of the TSP problem.
ii) The mapper then parses the Text item and retrieves the coordinates. It then constructs the graph.
iii) Then the pheromone array is initialized and ‘p’ ants (value p is specified by the user) are generated.
iv) Each ant uses the classic ACO algorithm to find routes in the graph. Within a mapper, the updates to the pheromone array are available to all ants instantaneously.
v) After all the ants have finished their work, the mapper initializes the TspInput object with all the paths generated by the ants.
vi) Finally it produces a key-value pair of type <Text, TspInput>. All mappers use the same key, so that all key-value pairs go to the same reducer.

The reducer takes the key-value pairs generated by all the mappers and iterates through them. During this process, the reducer performs updates

to the global pheromone array which will then be passed on to the next stage of the MapReduce. Also, the reducer calculates the best route of each stage. Once the reducer finishes its work, it writes the key-value pairs into a binary file (using SequenceFileOuputFormat). This format stores the key-value pairs so that subsequent stages in chained MapReduce jobs can process them faster. Like the original file, the output file from the first stage also contains replicated data. Also, it is important to note that the output of the first stage contains the updated pheromone table along with the original TSP input.

The code for the map() and reduce() methods used in the first stage of our MapReduce program is given below. Please note that the map() and reduce() methods of the intermediate and final stages are almost exactly similar to these. The only difference is in the types of the key-value pairs.
To serve as the mapper , a class implements the Mapper interface and inherits the MapReduceBase class . The MapReduceBase class, not surprisingly, serves as the base class for both mappers and reducers.

In TspMapper class a fixed number of ants work sequentially. These ants work on the pheromone update obtained by the previous ants in the same map. Similarly different maps run in the same stage mutually exclusive to each other thereby obtaining different routes and solutions corresponding to each ant in the maps.

The TspReducer class obtains the results from the different mappers of the stage compares them and obtains the best solution among them and combines the pheromone levels in the graph obtained from each mapper.

4.1.2 INTERMEDIATE STAGE

The output obtained as SequenceFileOutputFormat from the reducer of the initial stage is fed to the intermediate stages through the TspInterHadoop

class. It uses a modified mapper and reducer .they are different from the initial map reduce functions as they work on input in the binary form ie; SequenceFileOutputFormat.

The output format of TspInterhadoop is also SequenceFileInputFormat as they give their output to the next stages of computation. The TspInterMapper is same implementation of the initial TspMapper class.

The TspInterReducer is same as that of the Tspreducer which also compares the different values obtained from the mappers and selects the best result.

4.1.3 FINAL STAGE

The TspFinalHadoop class implements the TspFinalMapper and TspFinalReducer as the map and reduce functions respectively for the final stage . The input obtained from the previous stages is in SequenceFileOutputFormat and the output is as TextOutputFormat as they have to be understood by the user.

The TspFinalReducer ,as the previous reducers compares the solutions of different maps in the stage and finds the best possible one. And writes these to a file for the user to access.

4.2 SYSTEM PARAMETERS

4.2.1 INPUT PARAMETERS

There are many parameters that the user has control over when using the BoB algorithm:

1. Number of mappers per stage

This parameter determines the number of mappers that will be executed in each stage of the MapReduce program. This value determines the amount of parallelism in each stage as the mappers are executed in parallel.

2. Number of ants per mapper

This value controls how many ants will function inside each mapper across all stages. Higher value means that each stage will have more pheromone updates to pass on to the next stage of the job. Higher values will also increase the amount of computation to be performed inside each mapper.

3. Number of stages

This value indicates the number of stages in the MapReduce chain of our program. Each subsequent stage after the initial stage will have updated pheromone values for their ants to work with. This should make the computations of the ants more accurate as the number of stages increases.

4.2.2 OUTPUT

The two main outputs from our system are:

1. Time taken for the algorithm to execute

2. The shortest path computed by the algorithm

Since we are using a heuristic algorithm (Ant Colony Optimization), we will not be getting exact optimal solutions. However the algorithm gives approximate solutions that are quite close to the optimal solution. Usually the errors are between 2-5%.

4.2.3 COMPARISON OF INPUT AND OUTPUT VALUES

The MapReduce implementation of the ACO algorithm was then executed on a Hadoop cluster to evaluate performance. The different parameters and the results that we obtained for them were then compared to find a correlation between the input parameters and the results, and therefore find the optimal input parameters to obtain a good result.

5. TESTING

There are two main parts in this project – the sequential version of the ACO algorithm and the MapReduce version written in Apache Hadoop. The details of the testing done on both are explained below.

5.1 UNIT TESTING

5.1.1 Sequential Version

One of the major part of unit testing was with regards to the parsing of inputs from the input files. Since we were using input instances from the TSPLIB library, the parsing needed to be in accordance with the structure of those input files. The method for reading the input graph from the TSPLIB file was extensively tested to ensure that the co-ordinates were being properly read and that the graph was properly constructed. Different input files were fed into the program to ensure correctness.

5.1.2 Hadoop MapReduce Version

The MapReduce version of the program was written in Hadoop. The most important modules are the mappers and the reducers. Each map() method got an instance of the tsp input and calculated a route around the graph. The reduce() method collects the results from all the mappers and returns the best result.

The map() method was tested to ensure that it was producing the correct <key, value> pairs for the reducer to process. The output of the mapper is a route around the graph. So the mapper was tested with a variety of input graphs to ensure that it was producing the expected results.

The reduce() method collects the routes calculated by the mappers and returns the shortest route among them. The working of the reducer was again tested with various number of mappers.

Also since the intermediate stages in the MapReduce chain uses binary input and output formats, the process was tested for different number of stages to make sure that the mappers and reducers in the intermediate stages were also working as expected.

5.2 SYSTEM TESTING

In system testing the main objective was to make sure that the results obtained from the ACO algorithm was as close to the optimum results as possible.

5.2.1 Sequential Version

The accuracy of the sequential version of the ACO algorithm mainly depended on the number of ants and the values of the various parameters used in the algorithm. Several tests were carried out by varying the total number of ants and trying out different values for the greediness factor ALPHA, and the heuristic parameter BETA. After several tests we were able to decide on optimum values for these parameters. Also, it was found that increasing the number of ants beyond a particular value had not much effect in the accuracy of the results obtained.

The algorithm was also tested in multiple core processors to find out the performance enhancement that could be obtained. The time taken by the algorithm for different input sizes (i.e. number of cities) was calculated both using single core and multiple cores. The results were then compared which gave us an idea on the level of performance enhancement that could be obtained by parallelizing the algorithm. The faster times achieved the by the multiple core processing indicated that parallelizing the algorithm should provide better overall performance.

5.2.2 MapReduce version

After running the MapReduce version of the algorithm for a few times, it was observed that the results obtained seemed random and were far less

accurate than the results produced by the sequential version of the algorithm. Tests were carried out for different inputs, different number of ants and different number of MapReduce stages. Contrary to our belief, increasing the number of stages did not have any effect on the results and they continued to be random.

This was investigated further and after multiple tests it was found that the result produced by one stage was not being used by the subsequent stage. The error was fixed and the initial tests were carried out again. The change led to an improvement in accuracy but still mappers in a stage were unable to get pheromone updates. Also, increasing the number of stages introduce greater overhead with respect to job setup and other Hadoop overhead.

In order to be able to increase the total number of ants without increasing the number of stages, we decided to try and run multiple ants within a mapper as opposed to each mapper acting as a single ant. This increased the computation within each mapper, but these ants could share pheromone updates, and each subsequent stage had more pheromone trails to work with.

After the modification, the new system was tested with different number of input sizes, different number of stages and different number of ants. This helped us decide on optimum values for these parameters that would help us obtain the maximum benefit from parallelization over Hadoop.

6. COMPARISON OF PROPOSED SYSTEM AND CURRENT SYSTEM

The final system, which uses the Best of Breed parallelized algorithm, was compared with the original sequential implementation to check the difference in results and thereby conclude the optimization achieved by using the new algorithm. The test is divided into two parts, namely Load Comparison and Quality Comparison.

6.1 LOAD COMPARISON

Load comparison involves testing the system against the original algorithm for the same amount of computational load. The computational load in this system is the number of ants that we are executing in the system. Various tests were conducted to evaluate the ability of the parallelized system to handle large loads when compared to the sequential algorithm.

Computational power plays a large role in this algorithm as the program needs a large amount of processing power to enable it to execute the large number of processor computations.

Memory requirements were more in the sequential system when compared to the parallelized system as the memory footprint of this algorithm was fairly small compared to the requirements of the sequential system as the number of ants running in each stage was lesser than that in the sequential system. Memory played a smaller role in the program, but the reduced memory requirements means that this system can be implemented on systems that are smaller, or cheaper than the powerful sequential systems that we would require otherwise.

Figure 6.1 shows the system running different number of ants against different number of stages. We can see that the number of ants does not significantly affect the time of execution of the algorithm as compared to the number of stages

.**Figure 6.1 – Comparison of jobs with 1,2 and 3 stages chained**

The figure 6.1 depicts the running time of the algorithm for different number of ants represented on the X-axis and three different lines representing 1, 2 and 3 map stages. It can be observed that the slope of the graph is less than 45 degrees once the number of ants gets to a significant value. On increasing the number of ants the running time is not affected to the extent that increasing the number of stages would cost.

The overhead on the system is large when the algorithm is parallelized as the network congestion greatly raises the time required for the computation to complete. Moreover, Hadoop is not designed to handle large number of inter-node communication, but rather for large amounts of single transactions. Hence, as this system involves a large number of transactions, Hadoop's overhead causes a significant addition of time in the system.

It must be noted however, that this system is efficient on a larger number of cities as the computational efficiency masks the overhead caused by the underlying Hadoop system.

We tested this algorithm with various inputs and plotted the results in the graph shown in Figure 6.2.

Figure 6.2 – Variation in execution time against different inputs for one, two and five system cluster

Figure 6.3 – Variation of execution time against different number of inputs for 500, 1000 and 4000 ants

Figure 6.4 – Variation of execution time against different number of inputs for 1, 2 and 3 Map/Reduce stages.

Graph in figure 6.4 depicts the behavior of the algorithm when the number of Map/Reduce stages is increased. From the graph it can be observed that as the number of stages increases the running time of the algorithm also increases.

After comparing the different results, we concluded that the third method of parallelizing the algorithm was the best of breed (BoB). Also it can be inferred from the previous results that for accurate result and optimum performance of the Ant Colony Optimization algorithm it is necessary to weigh the factors of time and error to carefully choose the number of stages of Map/Reduce, the number of ants(agents) per mapper and the number of nodes in the cluster which can produce the best result among all and with the most optimum performance.

However, as the system is parallelized, we are able to run a larger amount of ants per system which makes the system more efficient when handling much larger loads than the sequential system. The sequential system is not able to handle as many ants as the parallelized system as it is a single processor with limited computational power. When the parallelized system is implemented on a large cluster, the computational power is greatly magnified and the system is able to perform computation much better.

6.2 QUALITY COMPARISON

Quality is a measure that the parallelized system is able to best the sequential algorithm by a considerable margin. The tested system is shown in Figure 6.2. The algorithm consistently generated results that were close to the optimal result.

The advantage of this system over the sequential system is that we can control the quality to a larger extent than the sequential system. While in the sequential system, once the system begins execution, there is no means to control it until we get the final result.

However, in the new algorithm, we are able to extract the result in each stage of execution, which allows us to get results with increasing amount of optimization. This allows us to run the system on a large number of stages and extract the best result at any given point of time from the system, and the system will continue to generate better results as the system executes for a longer time.

Figure 6.2

The above graph depicts the fact that even though the sequential version of the Ant Colony Optimization algorithm produces a more accurate value with a lesser error the algorithm when run of a multi-node cluster produces close to sequential values and even if the accuracy is not as good as the sequential the result is produces with a better running time. And a less accurate result can be afforded if the result can be obtained more quickly than the time to produce a more accurate result.