Making Ant Colony Optimization Parallel

Doug Patti
pattid2@rpi.edu

Ziv Kennan
kennaz@rpi.edu

Patrick Phipps
phippp@rpi.edu

ABSTRACT

This paper provides a look into the thought processes, considerations, results and implementation details of our final project: parallelizing the ant colony optimization to solve the traveling salesman problem within a reasonable bound.

It then provides a discussion on the results of the modified algorithm, comparing it to a serial solution as well as studying the scaling properties on two different computer systems. The two systems used in this study will be an optimized high perfomance system called Kratos and a Blue Gene/L system here at RPI.

General Terms

Traveling Salesman Problem

Keywords

Parallel, MPI, Ant Colony, Optimization, TSP

1. INTRODUCTION

Computationally hard problems commonly come up when attempting to solve useful and practical problems. One such problem is the traveling salesman problem. This is a well known problem in the computer science community. Formulated in 1930 [3] this problem is heavily studied as a case of an optimization problem. The premise of the problem is thus: given a list of cities and the distances between them pick a shortest path that visits each of them exactly once and returns to the starting city.

We chose to use the ant colony optimization to solve this problem. The ant colony technique is a probalistic technique that relies on sending out many ants that all take random paths, the best path is chosen and is more emphasized for the next iteration of ants. Over time this will converge to the best solution found. This technique was first discussed by Marco Dorigo in [1] for his PhD thesis. His application was similar to the one we have chosen but the technique has been applied to other NP-Hard problems.

2. REVIEW OF RELATED WORK

The first paper we found relating to our project is *A Parallel Implementation of Ant Colony Optimization*[4]. This paper does a thorough job in overviewing the various methods of parallelizing ACO. They mention independent ant colonies, which essentially runs the problem from several start points and then picks the best answer out of all of them (the embarassingly parallel solution). The second scheme they discuss is the same as the first, except the best trails are synchronized across all of the different problem sets. Another scheme is a master/slave approach with the processor being the master and the ant being the slave. In this case the master assigns the slave to a processor in which to do its entire bit of processing. The master then is in charge of synching the overall graphs. This approach seems to lend itself to shared memory architectures but can be done with message passing. The fourth scheme does an analysis of available solution elements before selecting one (rather than just at random). The fifth combines the fourth and third solution. None of these approaches are quite like what we have done in our paper in that none of them partition the graph and send the ants across processors. This paper also only tests graph sizes up to 657, this pales in comparison to the 1k-12k graphs we ran our algorithm on. This paper also observed some strong scaling properties of their algorithms as the processor counts used increased. In summarizing this paper also details the best scenarios to use each of the five schemes it had been investigating.

In a similar study at the Université Libre de Bruxelles [2], a slightly different approach was taken. The entire graph was duplicated among each core as a different *colony*, at which point multiple simultaneous simulations were run and results were communicated using a variety of techniques. In one method, each colony communicates to each other by using a single master colony to collect and broadcast results so that every other colony may synchronize. In a similar method, instead of the master broadcasting the best result to everyone, it only sends it to the colony that performed the worst so that it may replace it. Two more strategies involve different topologies of communication: the ring and hypercube. In these, each colony only shares results with its neighboring colonies to try and create a best result. The final method is the most trivial, and that is to have each colony execute entirely by itself, and then finally communicate results at the end to see which was the most successful. The results among all parallel implementations were more effective than the serial version; however, it was noted that the independent runs were the most efficient in terms of time. It was suggested that a compromise be made between the best-so-far update and the low-communication independent runs so that synchronization is only done when the best colony's optimal tour exceeds another's by a certain threshold.

3. SERIAL ACO ALGORITHM

All ACO algorithms begin with a graph. To begin the algorithm, an ant is placed on each node of the graph. At each iteraion of the algorithmm every ant chooses a new node to go to. It does this based on a probabilistic schema. Each edge has two weights. The first weight is proportional to the distance between two cities. The second weight is fundamental to the ACO scheme, it is the pheromone weight. The ant picks an edge based on a random roll relative to these two weights. These ant choosing iterations are repeated until the ants have completed a full tour of the graph.

Upon all the ants completing a full tour of the graph the ant with the shortest tour is picked and a global decay is applied to all the edges' pheromone weights. The algorithm then takes the best ant's tour and adds to the relevant edge's pheromone weight accumulator. The purpose of this step is to emphasis this edge in future ant walks. The algorithm is repeated until the ants begin to converge on a result.

Serial ACO

```
 1: procedure ACO(g)                          ▷ Returns best path in g
 2:     ants                                  ▷ Array of ants
 3:     while not converged do
 4:         for all ants → ant do
 5:             ant.tour
 6:         end for
 7:         decay all pheromone trails
 8:         find best ant trail
 9:         increment all edges on best trail
10:     end while
11: end procedure
```

4. PARALLEL ACO ALGORITHM

There are several modifications that are necessary to transform serial ACO to parallel ACO. The first task necessary in parallelizing ACO is the partitioning of the graph. This can be done in a variety of ways but this step is considered pre-processing as it makes sense to process a potentially large graph once and load it from the disk when it's needed. Potential partioning schemes include: round robin, clusters distribution, random distribution among other methods.

For our inital implementation, we chose a round robin graph partitioning scheme since it was relatively easy to implement and worked well with our implicit graph representation, which uses a hash function to calculate and store distances between nodes. Clustering schemes are perhaps better suited to this application since they can reduce the amount of network traffic, and speed up the time needed to find a solution.

After partitioning the graph, the ACO algorith is run on each node. During each iteration an ant attempts to make a tour. However, it will inevitably find that it needs to visit a node that is absent from the processor responsible for it. At this point, the ant is sent over to the proper processor. Once it is received there, it joins the rest of the ants on that processor in the queue waiting its turn to process. After being sent around to every processor at least once, visiting all of the nodes in the graph, the ant has completed its tour. The processors then do a reduce amongst themselves to see which ant(s) had the best tour. A reduction is then done over all processors that sums the total number of ants that have shortest tours. Every processor then expects to receive that many ants minus the number of shortest tour ants that it currently has. This is the sync step, now all processors have a copy of the best tour ants. Each processor then updates the pheromone weights using the best ants' tours.

A somewhat trivial improvement to this algorithm that cannot be done on the Blue Gene is to have threads process the ants in the queue. This can be somewhat tricky to implement with all of the coordination that must go on in order to keep the processes from having unexpected behavior.

Parallel ACO

```
 1: procedure ACO(g)                          ▷ Returns best path in g
 2:     ants                                  ▷ Array of ants
 3:     while not converged do
 4:         while every ant has not visited my nodes do
 5:             for all ants → ant do
 6:                 ant.tour
 7:             end for
 8:             receive new ants
 9:             add new ants to ants queue
10:         end while
11:         perform reduction to determine smallest tour
12:         count = reduction sum of total smallest tour ants
13:         if I have a smallest tour ant then
14:             send ant to all
15:         end if
16:         receive ( count − my smallest tour count ) ants
17:         decay all pheromone trails
18:         find best ant trail
19:         increment all edges on best trail
20:     end while
21: end procedure
22: procedure Ant Tour(ant)                  ▷ Parallel Ant Tour
23:     while all nodes not visited do
24:         Determine edge
25:         if edge is not on processor then
26:             send ant to processor with edge
27:             break
28:         end if
29:         Take edge
30:     end while
31: end procedure
```

5. IMPLEMENTATION DETAILS

The first step in implementation was to create a throwaway prototype in Python in order to assist in getting an understanding of a serial ACO algorithm. This used a simple adjacency matrix and allowed us to tune parameters to find a suitable reference for our main implementation.

Our main implementation is written in C/MPI. The most essential components of the ACO algorithm in the program is the partial adjacency matrix holding the pheromone levels and the ants themselves, which must be transferred from core to core. Firstly, the adjacency matrix is not a full one, so each core only has a slice of the matrix equal to one full row for each vertex of the graph that belongs to it. Each cell of the matrix contains a double type store for the pheromone level. Because this is a full matrix being divided between cores, it means the memory scaling of the edge matrix is

$O(n^2)$.

The adjacency matrix does not actually store the weights; those are calculated using a deterministic algorithm that is meant to produce weights during computation. Any two nodes will always produce the same edge weight with this method. This was a compromise from reading in a file from disk, mainly because not only would the file occupy a large amount of the file system for a significant graph, as it would be close to 40 gigabytes for a full core's worth of data, but also because it was not a very important element of the problem.

We did create a few different algorithms to simulate different partitioning systems. First was the round robin method, which was designed to give very edge weights completely unrelated to the nodes that were supplied, i.e., it was almost a deterministic random edge distribution. On the other hand, the distance method returns smaller weights for nodes that are numerically closer to each other, and because cores are given sequential nodes of the graph, they have a higher tendency to stay within the core than to jump back and forth. The final method, clustering, was an experiment to accentuate the distance method and additionally penalize nodes that were specifically not part of the current core's node cluster.

The other essential component was the ant object. We allocate enough ants for each graph node on each core, but in addition to that, a buffer of ant objects are allocated so that cores may receive more ants at any given time. This helps keep communication blocking time low. This contiguous block of memory is populated with a few details about the ant's first and current position, but in addition to that it contains an array of integers equal to the number of nodes in the graph. This is important, as the ant is passed from core to core using MPI, and when it reaches its destination, it must know which edges it can take. This means that the size requirement for a single ant is $O(n)$. If the number of ants is set equal to the number of vertices, the total cost of all ants is $O(n^2)$.

The array of elements that is part of the ant's path is organized in a way to solve two computational problems in $O(1)$ time. Firstly, when an ant reaches a node and must select an edge to take, it must ignore each of the edges that it has taken already. As it computes the probability of taking each edge, it can look up the edge index in its array using random access to test. While this is simple, there is a caveat. The ant that is the most successful must retour its path at the end of the iteration, and so the explicit order of the nodes must be known. To solve this, when an ant picks an edge to cross, it stores the id of the node it is departing towards as the value of the array at the index of the node which it is leaving from. This way, it works similar to a linked list, where you can always find the next node given the previous in $O(1)$ time.

To organize the flow of ants, two data structures are used: queues and arrays. The three queues we use are meant to separate ants based on where they are currently located. The spare queue is where each ant starts out, the process queue contains ants who are looking for another edge to augment their path, and the finished queue is for your original ants that have completed their tour and are waiting to be scored. All unused ants are left in the spare queue, and this is where the communication module draws from so that it may receive ants from other cores into the pre-allocated space.

For communication between cores, an array was used. The primary benefit of this is the easy access to the MPI_Testsome() call, which takes an array of requests and returns the ones that have completed. The array starts out with one index for each ant allocated, and each request object is set to a null request, so that MPI will ignore those indices when checking for data. A few other arrays are created to hold data in a similar fashion for each request index: request type and ant object pointer. Because this single array is used for all communication, the request type signifies whether the ant is being sent or received. The ant object pointer simply holds on to a reference of the space in memory while the operation is being completed, after which the pointer will be returned to the spare queue. The ants are placed into the array for communication using a rotating index that increments each time a spot is used and wraps around to the first index when it reaches the end. Because transfers do not take a deterministic amount of time, the index will always scan for an empty index to populate in this same manner.

Communication is done on a by-call manner. Non-blocking receives are initiated every loop based on certain parameters. Currently, no more than a set number of requests are opened at any given time, which prevents scenarios where hundreds of ant requests are left unfulfilled and inevitably cancelled at the end of the iteration. Sends are always initiated by the ACO cycle when it decides that an ant will be transferred to a new core.

At the end of an iteration, each core tallies the scores for its ants, and that score is reduced as a sum and as a minimum between all cores. The sum allows us to find the average tour length of all the ants, while the minimum specifies the ant with the best tour of that iteration. Each core checks to see if it has any ants with that tour length, and this count is reduced so that each core knows how many ants found the best tour of that iteration. The ants are sent to each other's cores simultaneously, at which point the core can adjust the pheromone levels from its nodes on the edges that the ant took. This is the crucial synchronization step of the ACO algorithm.

At this point, all ants are back in the spare queue, the initial ants are reset for each core, and the simulation begins another iteration. When all iterations are complete, a summary of statistics is generated, and the allocated memory is freed before the program exits.

6. TESTING AND RESULTS

We ran a battery of testing configurations on both Kratos and the IBM BlueGene/L at the CCNI. To get a wide survey of our algorithm's performance we tested all three graph partitioning methods on Kratos. Taking the best performing method and using that to test on the Blue Gene in order to

alleviate the potential of running over the 10 minute execution time limit. On Kratos we ran a graph of 4,096 nodes on 5 different processor configurations: 1, 2, 4, 8, and 16. We ran these runs using each of our three partioning schemes: round robin, distance, and clustering. We then took the best perfoming method out of these tests and ran an 8,192 node graph using all of the above listed processor configurations. An important note to our results is that Kratos is time to complete 10 iterations of our algorithm. On the Blue Gene we timed only one iteration to ensure that we could run the largest processor counts as well as the smallest processor counts in the limit. The results of these runs are present in the graphs below.

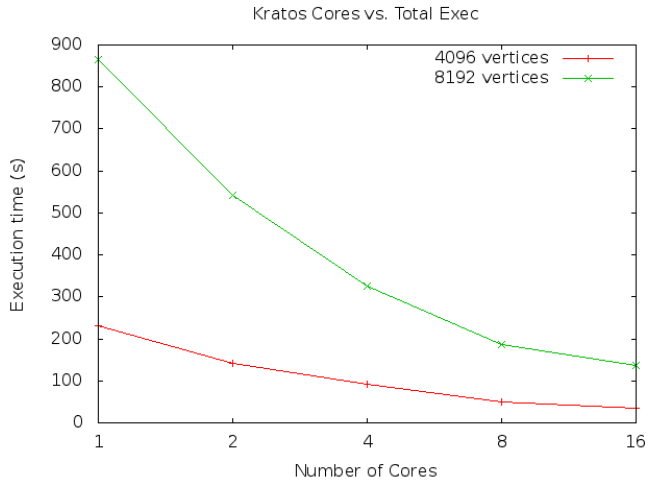

Figure 1: Total Execution Time vs. Core Count on Kratos using distance partioning

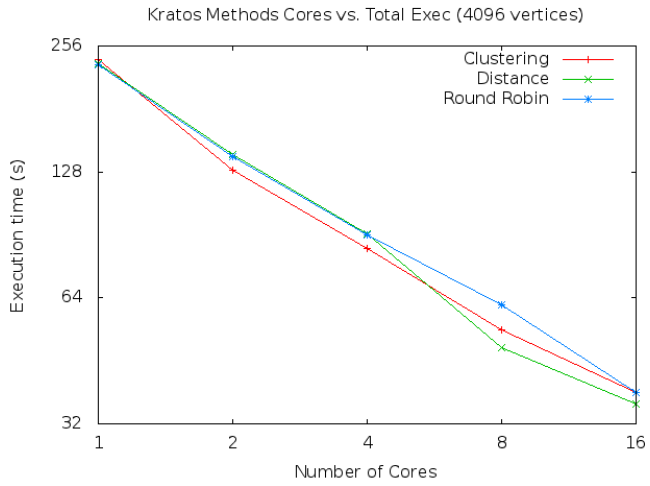

Figure 2: Total Execution Time vs. Method/Cores on Kratos on 4,096 nodes

Taking a closer look at the Kratos results we can see that our implementation exhibits strong scaling qualities. For case of 4,096 nodes vs. execution time (red line) it is shown that for a problem of this size the algorithm gets faster as more cores are added, but it is not a very dramatic improvement. As we scale up the problem size to 8,192 the scaling properties of our algorithm begin to show through. From the serial version of almost 900 seconds is cut to less than 200 seconds using 16 cores. The trend of this line exhibits the excellent time reductions given by additional processors. With the scaling propertiees of the algorithm being demonstrated in the first graph, the second shows demonstrates the difference made by using other partioning techniques. In general we see that the round robin partioning scheme yields the worst performance. The cause of this is the number of times ants have to cross cores. With the round robin scheme there is no sense of order to the nodes in each processor, thus each ant has a higher probability of needing to be transfered on its next move. The clustering scheme works well for relatively small core counts but is beat out in the end by the distance method. However, it is interesting to note that on the graph size of 4,096 all three methods give competitive times for the 16 core case. Further testing on larger graph sizes is required to really see how these partioning schemes affect performance.

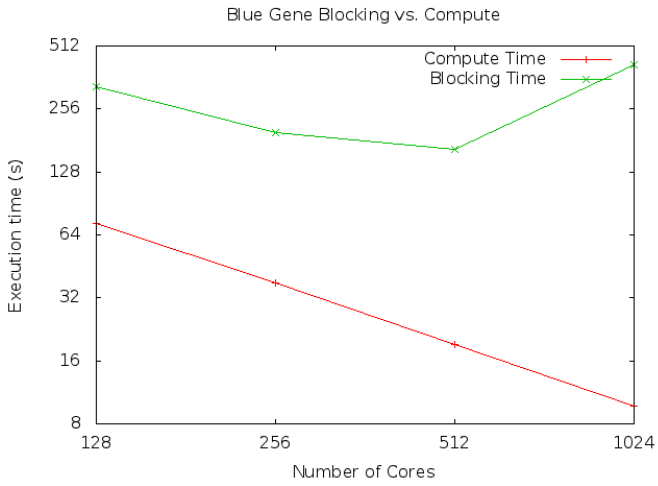

Figure 3: Blue Gene Blocking Time vs. Computation Time

We see in the computation time vs. transfer time (blocking time: message passing time) that transfer time is clearly the bottleneck in the performance of our algorithm. These Blue Gene results utilize the clustering partioning method. The rational behind using this method is that it performed best on Kratos, and that we want to choose the method with the overall lesser amount of ant transfers on average which the distance method delivers. As expected the time spent computing drops linearly with the amount of cores, as the amount of processing done relative to transfers as the core count increases becomes small. Looking at the total execution time vs. cores for various problem sizes on the Blue Gene it is important to note the graph below regarding VN mode vs. Normal Mode. This is important because it helps to explain the seemingly anomalous result we receive at the 1024 core execution (that can only take place in VN mode). We observe that there is an overhead associated with VN mode causing a performance drop and presumably that large spike at 1024 cores. Disregarding the 1024 point a decrease in execution time is associated with an increase

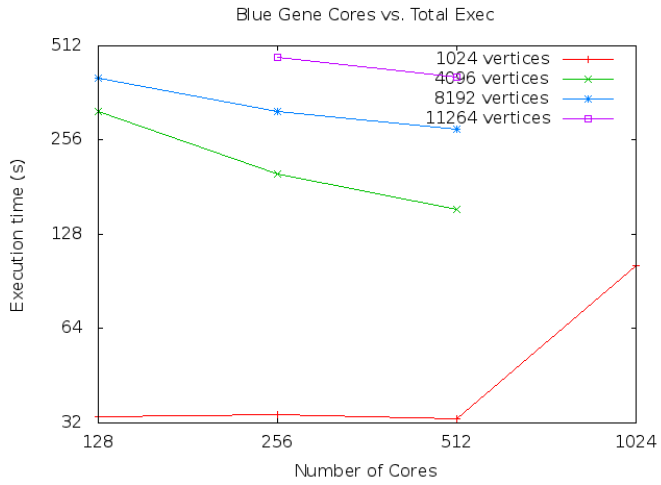

Figure 4: Blue Gene Total Execution time vs. Core Count - various graph sizes

in the number of cores used. This, along with the Kratos results, leads us to believe that our algorithm experiences a strong scaling effect with the number of cores

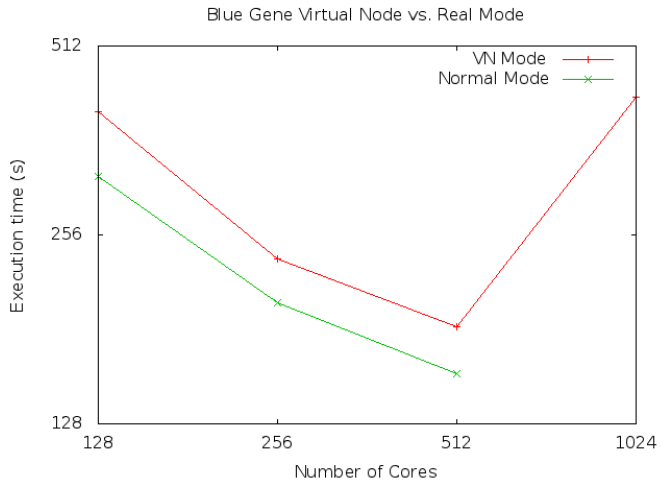

Figure 5: Blue Gene Normal Mode vs. VN Mode

An interesting observation of the tests that we ran is that the Blue Gene is out performed by Kratos for the problem sizes that we used. The assumption being that we did not achieve proper graph size to core count ratios to maximize the Blue Gene, implying that there is an optimization problem to maximize the algorithm in the problem size to core ratio. It is also important to realize that we are not saturating Kratos' bandwidth in sending ants across the network, meaning that processor speed is more important in the test battery that we used in testing than the speed of the network.

7. COMPARISON TO SERIAL ACO TSP

Our algorithm is written such that it takes advantage of n cores, from 1 to k, where k is the maximum number of cores available to the system. Large problem sizes will have difficulty fitting on a single core's worth of memory. Our algorithm is intended to require less memory per processor as the number of processors increases, so the more processors you have the bigger the problem size you can handle even if each processor has a relatively small amount of memory. From the graphs above, we can see that the parallelized version was successful at least as far as outperforming its serial competitor; the parallel version being roughly 7 times faster on the largest test case that was run on Kratos.

8. CONCLUSIONS

The Ant Colony Optimization problem sets out to solve an NP-hard problem, giving little hope to a well scaling serial solution. We set out to adapt this algorithm to the parallel programming domain by splitting up the graph problem into process sized chunks. By spliting up the graph we necessarily complicate the problem slightly by requiring a communication scheme that allows us to maintain an ant taking a complete tour across the graph by moving between the processes. It is also required that we can sync the best trail across all of the partial graphs.

We believe that we have met the requirements that we had set forth for ourselves on this project, and while we never got to attempt to see how our algorithm works on our theoretical Blue Gene memory limit of around 120,000 nodes the tests performed gave us a good idea of how well our implementation actually works. We have seen from the results sections that on the whole the algorithm had more impressive completion times on Kratos and that our scaling properties were very well pronounced on that system. The results we have obtained from the Blue Gene show a general trend of improvement as the core count increased we speculate that the problem sizes tested were not properly scaled to the processor counts we had accesss to.

9. FUTURE WORK

The first avenue of future work for this project is to more thoroughly test our implementation. Ideally, we would acquire more processing time on the Blue Gene and work out some discrepencies we had been observing in VN mode runs vs. non-VN mode runs with VN mode hindering algorithm performance. With the 700mHz processors in the Blue Gene we cannot get through one iteration of problem sizes $>$ 18k nodes in 10 minutes. These larger problem runs can give us better insight as to how the algorithm performs on the Blue Gene as a whole, considering that our smaller problem size results were better overall on Kratos we would like to scale out to problems that could not fit on Kratos.

While our implementation and exploration are relatively robust and thorough, there are many ways in which this project could be further improved and explored. One option that might provide an easy performance boost is to use threading (pthreads) in addition to MPI. While we didn't have time to explore this, it is a relatively straightforward extension, and on a hardware-threaded system such as BG/Q it might provide significant performance boosts.

Another interesting extension to our work is to use genetic algorithms in addition to ACO to further increase the quality of our solution while also allowing the algorithms to run

in fewer iterations. Genetic algorithms can be usaed to generate string mutations which iteratively create better routes for the ants to follow, instead of choosing random routes to start off with.

One of the limitations of our implementation is that we use a round-robin distribution on an implicitly defined graph; this means that not only can an arbitrary explicit graph not be used with our system, but any graph used is not partitioned in an optimal manner. As mentioned above, alternative partitioning schemes and support for explicit graphs are two important features that we would add to our system in the future.

One of the last improvements we would like to make to our implementatino is the ability to auto tune certain internal parameters of the algorithm based on the quality of the solution desired and the speed at which it is needed. By adjusting the step and decay functions for our pheremone levels (alpha, beta, etc) as well as adjusting the precision of the numbers we use we can greatly affect the runtime performance of our implementation. Since we ran into some issues using floats, we switched to doubles early on, but that decision could be made at runtime depending on the type of solution desired. By auto-tuning all of these parameters our implementation would have much more flexibility; however given the time-frame we had, we were unable to do this.

10. ACKNOWLEDGMENTS

We would like to thank professor Carothers for the feedback he provided to our group during the project, as well as the tips he gave us throughout the semester, they were very helpful in completing this project.

11. WORK DISTRIBUTIONN

The idea was chosen by all three group members collaboratively, and fleshed out during a group meeting. Most of the coding was done with all three members present, with Doug writing most of the code and other members providing input and aiding with debugging. The paper was written by all three group members collaboratively, with Ziv and Patrick doing most of the actual writing. Each step of the algorithm was written out in comments in the source files to be filled out later. All group members participated in this pseudocoding step of our process.

12. FURTHER NOTES

The source code to our project can be found in /*home*/*parallel*/2012/*PPCpattid*2/*project*/ on Kratos. Also present is a read me and the raw data for our graphs.

References

[1] Durigo Marco. Optimization, learning and natural algorithms. *Ph.D. Thesis*, 1992.

[2] Thomas Stützle Max Manfrin, Mauro Birattari and Marco Dorigo. Parallel ant colony optimization for the traveling salesman problem. 2006.

[3] E. Keith Lloyd Norman L. Biggs and Robin J. Wilson. *Graph Theory 1736-1936*. Clarendon Press, 11 December 1986.

[4] Marcus Randall and Andrew Lewis. A parallel implementation of ant colony optimization. *Journal of Parallel and Distributed Computing*, 62(9):1421 – 1432, 2002.