Meta-Heuristics in Multi-core Environments

Ragnar M. Ragnarsson

2010

MSc thesis in Decision Engineering

Author: Ragnar M. Ragnarsson
Id no:
Supervisors: Dr. Eyjólfur Ingi Ásgeirson and Dr. Hlynur Stefánsson

Tækni- og verkfræðideild
School of Science and Engineering.

Upplýstar Leitaraðferðir í Fjölkjarna Umhverfum

Útdráttur

Mörg raunveruleg vandamál eru stór NP-hard vandamál og fyrir öll stærri tilvik er nánast ógerlegt að finna bestu lausnina á þeim á ásættanlegum tíma. Upplýstar leitaraðferðir eru oft notaðar til að fá góða lausn á vandamálum á ásættanlegum tíma en þau gefa ekki endilega bestu lausn. Í þessari grein verður farið yfir það hvernig má auka afköst upplýstra leitaraðferða með fjölkjarna vélum og finna út hvað ber að varast. Til að gera það voru valin þrjú reiknirit og þrjú vandamál. Reikniritin eru Aimulated Annealing, Ant Colony og Tabu Search en vandamálin eru Ackley fallið, Traveling Salesman vandamálið og Vehicle Routing vandamálið.

Með því að keyra Simulated Annealing reikniritið á hverjum kjarna sem eyju fengum við út að hraðinn jókst um 1.6 þátt við hvern kjarna sem var bætt við.

Einnig kom í ljós, að með því að skipta verkefninu upp í hluta sem tekur langan tíma að reikna og ekki þarf að nota mikið af sameiginlegum auðlindum er góð útfærsla fyrir fjölkjarna umhverfi, gefið að örgjörvin sé hraður. Með þess konar reikniriti fyrir Ant Colony fengum við hraðaaukningu upp á 3.7 þætti fyrir vél með fjóra kjarna.

Aftur á móti kom í ljós að með því að skipta verkefninu upp í hluta sem tekur stuttan tíma að reikna og nota mikið sameiginlegar auðlindir milli reikninga, er ekki góð útfærsla fyrir fjölkjarna umhverfi. Það á sérstaklega við um hraða örgjörva. Með þess konar reikniriti fyrir Tabu Search minkaði hraðinn um 1.4 þátt fyrir vél með fjóra kjarna.

Það má bæta hraða upplýstra leitaraðferða með því að forrita sérstaklega fyrir fjölkjarna umhverfi án mikilla vandræða. Hversu mikla hraðaaukningu má vænta fer algjörlega eftir reikniritinu, útfærslunni (hvort notaðar séu sameiginlegar auðlindir eða lásar) og tölvunni sem notast skal við. Til að fá nokkuð örugga hraðaaukningu má keyra reikniritin sem eyju á hverjum kjarna en best er að velja rétt reiknirit og vélbúnað fyrir hvert vandamál fyrir sig.

Meta-Heuristics in Multi-core Environments

Abstract

Many real life problems are large NP-hard problems and thus for nontrivial instances it is almost impossible to find the best solution in a reasonable amount of time. Meta-heuristic algorithms are often used to get a good solution in a decent amount of time but they do not guarantee an optimal solution. In this paper we will take a look at how to increase the performance of meta-heuristic algorithms using multi core architectures and try to identify potential pitfalls along the way, to do that we will use three meta-heuristic algorithms and three different problems. The algorithms are Simulated Annealing, Ant Colony and Tabu Search and the problems are the Ackley function, Traveling Salesman problem and the Vehicle Routing problem.

We found that by using a delightfully parallel algorithm, where the whole algorithm can be run on a single thread with no dependence on other threads, for Simulated Annealing the average gain for a new core is a factor of 1.6.

We found that an implementation characterized by heavy calculations on each core with heavy reading and light writing in a shared memory is an ideal implementation for the multi core environment given that the CPU is both fast and has a large integrated cache memory. With such an algorithm we got a performance increase by a factor of around 3.7 for a quad core system using Ant colony meta-heuristics.

We found that an implementation characterized by relatively light computations on each core and the use of shared resources between computations is not a good implementation for the multi core environment. This was especially evident with a fast CPU. With such an algorithm we got a performance decrease by a factor of around 1.4 for a quad core system using Tabu search meta-heuristics.

By programming especially for multi core processors the performance of meta-heuristic algorithms can be improved without much additional cost or effort. The results depend on the algorithm chosen, the implementation (sharing memory structures, locks) and the system architecture. To get a solid performance gain, a delightfully parallel implementation is a very good choice but ideally one should select the correct algorithm and hardware for each problem and tune the algorithm for the selected hardware.

Key words: Meta-heuristics, multi-core, Simulated Annealing, Ant Colony, Tabu Search

The undersigned hereby certify that they recommend to the School of Science and Engineering at Reykjavík University for acceptance this thesis entitled “Meta-Heuristics in Multi-core environments” submitted by Ragnar M. Ragnarsson in partial fulfillment of the requirements for the degree of Master of Science.

_____

Supervisor, Dr. Eyjólfur Ingi Ásgeirsson

_____

Supervisor, Dr. Hlynur Stefánsson

_____

Examiner, Dr. Slawomir Marcin Koziel

Table of Contents

List of Tables

page

List of Figures

1 Introduction

The roots of OR (Operation Research) can be traced back to the Second World War when the allied forces needed to run military operations on scarce resources. The allied military called upon teams of scientists to try to apply scientific methods to the problem. These teams were quite successful and turned out to be an important factor in winning the Air Battle of Britain and the Battle of the North Atlantic [28].

Many of the scientists continued their research into OR after the war and its success triggered interest in the field outside of the military. By 1950 many of the tools and methods still used today were already developed, such as linear- and dynamic programming, queuing theory and inventory theory.

There was a major growth spurt in the OR world with the computer revolution. A large amount of computation is required to solve most OR problems and computers can be used to solve computational problems much faster than humans, especially repetitive calculations as is often the case in OR. With growing popularity of personal computers, OR has been put into the hands of more people than ever before [8].

Many real life problems are large NP-hard[1] problems and thus for nontrivial instances it is almost impossible to find the best solution in a reasonable amount of time. Meta-heuristic algorithms are often used to get a good solution in a decent amount of time but they do not guarantee an optimal solution. Meta-heuristic algorithms have become ever more popular during the recent years, in part due to the computer revolution [7].

Meta-heuristic algorithms are fairly easy to implement in a single core environment, i.e. one CPU (Central Processing Unit) using integrated cache memory and random access memory (RAM). The cache memory is integrated in the CPU and is faster than the RAM memory which is connected to the CPU through a bus (a high speed channel) which is much slower than the CPU. Though meta-heuristic algorithms are easy to implement on single core computers they run rather slow, especially on personal computers, though modern PCs are more powerful than many supercomputers through the years. To get better performance, more focus has been put into parallel computing. Parallel computing is where a problem is broken up into smaller entities and two or more entities are worked on at the same time. In parallel computing there are two prevailing inter-processor communication methods; shared and distributed memory models [3]. In shared memory systems a collection of homogenous processors share the same main memory through busses (or crossbar for more than four processors). Symmetric multi-processors, SMP, are an example of a shared memory model and many consider chip-level multi core processors to belong to this model, Figure 1 shows a simple diagram of a SMP and Figure 3 shows a simple diagram of a quad-core processor. Distributed memory systems are on the other hand comprised of more than one heterogeneous stand-alone machines, each with its own central processor unit and memory set, connected together through a high speed network. Good examples of distributed memory systems are Beowulf clusters [4], Figure 2 shows a simple diagram of a Beowulf cluster. The two main APIs (Application Programming Interfaces) for the shared memory model are POSIX threads [26] and OpenMP [6], for the distributed model the MPI (Message Passing Interface) is by far the most commonly used [5].

[1] non-deterministic polynomial-time hard

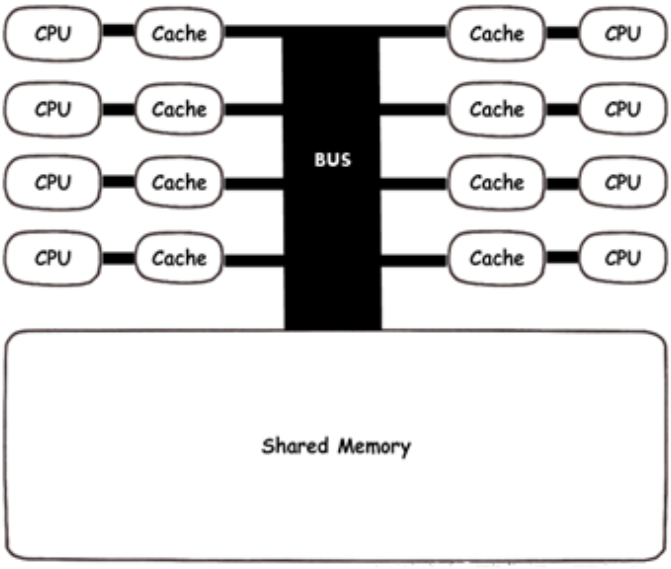

Figure 1 A diagram of a SMP with eight CPUs connected to a shared memory through a high speed bus or a crossbar.

Both the shared memory and the distributed memory models have their advantages and disadvantages. When porting a sequential code to a shared memory model, parallelizable code needs to be identified and written again in APIs such as OpenMP. This can introduce complications such as race conditions, deadlocks or other problems often accompanied with shared memory models. It is even more complicated to write code for distributed memory models, since it usually involves writing efficient algorithms to divide tasks among processor and memory sets and synchronizing the results. The distributed model is often limited by slow network speeds and thus each task needs to be large enough to overcome the network latency. On the other hand, distributed systems are more scalable; adding more processors to such a system often simply involves adding another node (computer) to the current environment whereas doing so on shared memory systems increases the bus traffic and slows down memory access. In some cases it even requires expensive and complex hardware changes, maybe involving investing in a completely new system with room for more processors.

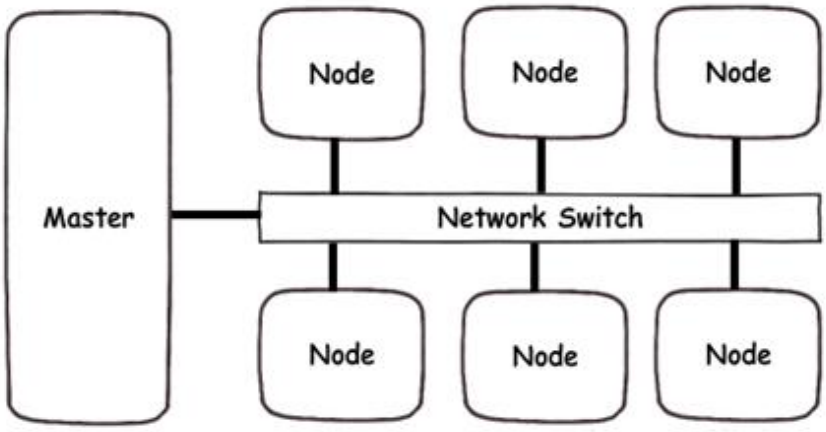

Figure 2 A diagram of a Beowulf cluster where both the master and the nodes are normal PCs connected through high speed networks. The master computer delegates the work, gathers the results from each node and compiles it to one final solution. The network can be anything from Ethernet to InfiniBand.

In 1965 Gordon Moore wrote an article for Electronics magazine titled “Cramming more components onto integrated circuits” [10] where he predicted, often called Moore’s Law, that the number of transistors on a chip would double each year into the foreseeing future. His prediction has, more or less, come true over the years. Around the year 2005 chip makers were having trouble with increasing the performance of chips; the small size of the transistors were causing heat and power issues. The answer was adding more cores on the same die, which gives a significantly better performance while only increasing power usage by a small percentage. This essentially threw the problem of ever increasing speed on to the software engineers but so far not many programmers have been utilizing this technology to any extent. In recent years, tools and support for parallel programming have been added to many software languages, such as Java, C++ and C#, giving programmers better opportunities to use the full power of the modern CPU without spending too much time in training and/or refactoring [1][2]. Since increasing speed is on software vendors, operating systems have to be scalable and use multi core technologies, today many operating systems are not optimized for multi core CPUs which can have an effect when writing multi core algorithm implementations [29].

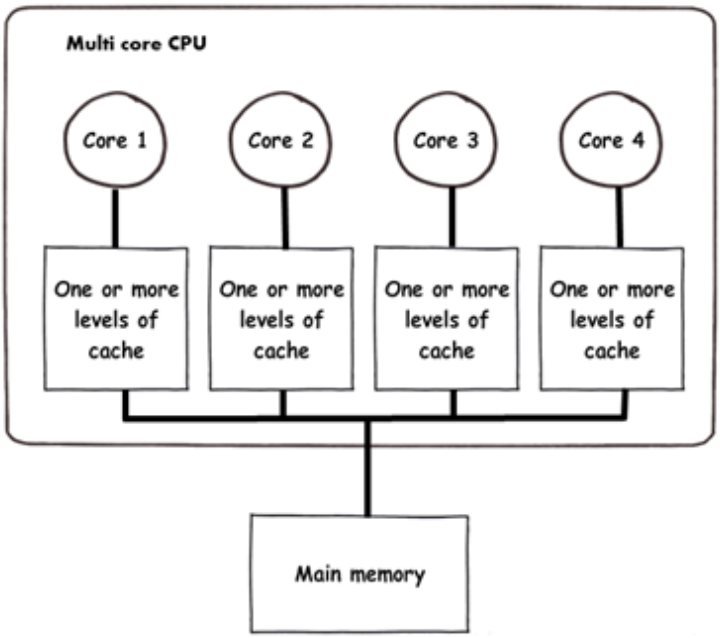

Figure 3 A diagram of a quad-core CPU where there are four cores, each with a dedicated cache memory. All the cores use the same main memory which is connected to the CPU with a high speed bus. Multi core CPUs can have different architectures as well, for example shared cache memory.

There are not many meta-heuristics papers focusing on the multi core architecture. Bui, Nguyen and Rizzo Jr. [9] showed results where the running times of an Ant Colony algorithm improved by around 40% with a dual core system and by a factor of up to 6 on eight core systems. In this paper we will take a look at how to increase performance with multi core architectures and try to identify potential pitfalls along the way.

2 Methods and algorithms

We selected three different meta-heuristic algorithms along with three different problems. When selecting problems and algorithms for this paper many candidates came to mind and some

interesting algorithms are being left out, such as Particle Swarm and Evolution algorithms which were left out due to insufficient time. Simulated Annealing was selected mainly because of its simplicity and popularity while both Ant Colony and Tabu Search have features, such as shared memory structures, that are interesting to implement in multi core environments.

2.1 Simulated Annealing and the Ackley Function

Simulated Annealing was described by Scott Kirkpatrick, C. Daniel Gelatt and Mario P. Vecchi in 1983 [18] and by Vlado Černý in 1985 [19]. The method is inspired by annealing in metallurgy which is a technique involving heating and controlled cooling of a material. Simulated Annealing is an iterative search method where the current solution is replaced by a random solution from the neighborhood of the current solution. The random solution is accepted with a probability that is based both on the relative values of the object function between the current solution and the objective value of the next solution, and a global parameter T, often called temperature, which is gradually decreasing during the process. The function for temperature decrease was very simple; the temperature value times an alpha value, where alpha is close to one, $\tau = T * \alpha$, $\alpha \in (0.9, 0.99)$.

The problem we used for the Simulated Annealing method is called the Ackley function, the objective is to find a minimum value of the Ackley function. For this paper the main focus is on comparing the single- and multi core implementations and not necessarily finding the absolute minimum of the function. Formally, the Ackley function can be described as a vector $\vec{x} = \{x_1, x_2, x_3, \dots, x_n\}$ where $x_i \in (-32.768, 32.768)$ that minimizes the following equation.

$$F(\vec{x}) = -20 \cdot e^{-0.2 \cdot \sqrt{\frac{1}{n} \cdot \sum_{i=1}^{n} x_i^2}} - e^{\frac{1}{n} \cdot \sum_{i=1}^{n} \cos(2\pi \cdot x_i)} + 20 + e \quad (1)$$

where n stands for the number of parameters used. In this problem only two parameters were used; x_1 and x_2 [11].

The function can be seen in Figure 4, it resembles a giant egg tray with a steep egg tray shaped

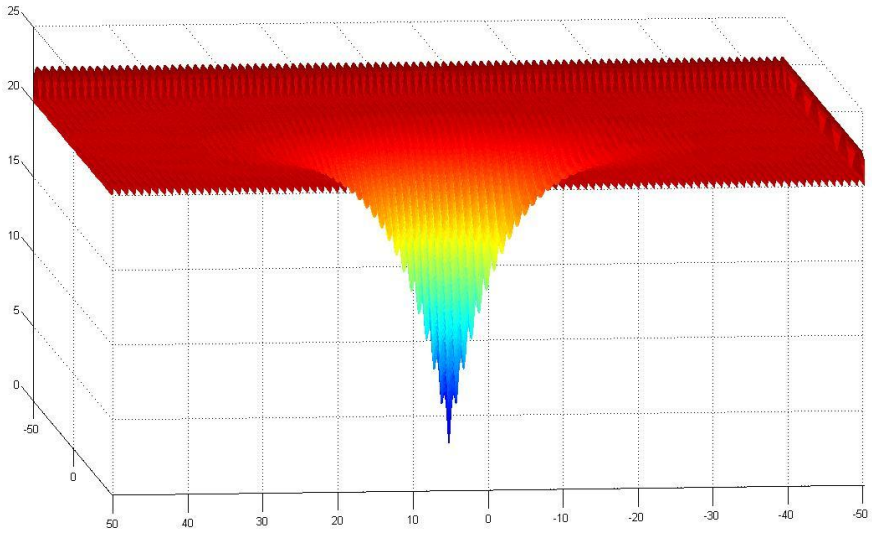

Figure 4 The Ackley function resembles a giant egg tray with a steep egg tray shaped vortex around the zero point. This figure shows only a small portion of the function around the zero point.

vortex around the zero point. The solution of the function as used in this paper is two dimensional. The neighborhood is defined as a small square with the current solution in the middle.

2.2 Ant Colony and the Traveling Salesman Problem

Ant colony optimization has been popular in recent years. The ant colony algorithm is based on the idea of mimicking ant colony behavior, and was first proposed by Marco Dorigo in 1992 in his Ph.D. thesis [12][13]. At the time it was called Ant System and was the result of research into computational intelligence approaches to combinatorial optimization.

We can look at the behavior of ants as follows. When searching for food ants start to wander around randomly until they find a food source. Once they have found a food source they return to the colony and on the way back they leave behind a pheromone trace. Other ants who stumble onto such a pheromone trail are likely to follow it. If the ants find the food source they will also return to the colony and thereby reinforce the pheromone trail. With time the pheromone trace will evaporate, that will lead to shorter trails having more pheromone than longer ones. Evaporations can also help avoid convergence to local optima.

Ant colony optimization tries to mimic this behavior by using artificial ants to travel through a graph updating the pheromone trail between iterations. Attractiveness and pheromones are then used to select the most attractive route [15][17]. In the implementation used in this paper the inverse of the distance is used as the attractiveness parameter and only the best route from given iteration is used to update the pheromone. The following equation is then used to calculate the probability of selecting an edge.

$$p_{i,j} = \frac{\left(\tau_{i,j}^{\alpha}\right)\left(v_{i,j}^{\beta}\right)}{\sum\left(\tau_{i,j}^{\alpha}\right)\left(v_{i,j}^{\beta}\right)} \qquad (2)$$

where $p_{i,j}$ is the probability of selecting route (or edge) (i,j), $\tau_{i,j}$ is the amount of pheromone on edge (i,j) and $v_{i,j}$ is the attractiveness or visibility of edge (i,j). Here $v_{i,j}$ is $1/d_{i,j}$ where d_{ij} is the distance between node i and j. The α and β are parameters to control the influence of $\tau_{i,j}$ and $v_{i,j}$ respectably.

The TSP (Traveling Salesman Problem) was first formulated in 1930 as a mathematical problem although the problem itself is considerably older [14]. It is a very popular problem of finding the shortest route between a set of cities by only visiting each city once. The TSP is used as a benchmark for many optimization methods [19]. In the traveling salesman problem we are given a complete undirected graph $G = (\mathcal{N}, E)$ with a nonnegative cost c_e associated with each edge $e \in E$. The goal is to find a tour (or a Hamiltonian cycle) of G with minimum cost. A mathematical formulation of the TSP is given in the following equation.

$$\min \sum_{e \in T} c_e \qquad (3)$$

$$s.t. \;\; T \; is \; a \; tour \; of \; G$$

The ant colony case of the TSP has a fixed number of ants in each city per iteration. A starting city is chosen and each ant from that city will traverse the grid creating a path along the way, to select a city to visit next, equation 2 is used. When all ants have completed their routes the pheromone is updated, and only the shortest route is used to update the pheromone table. The TSP instance used

in this paper was an instance based on the shortest Euclidian distance between the 200 largest cities in the USA as shown in Figure 5.

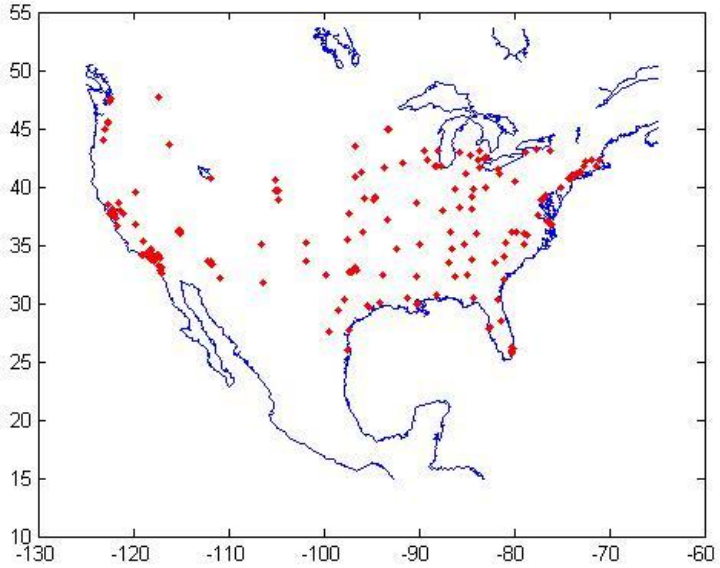

Figure 5 The 200 largest cities in the United States of America.

2.3 Tabu Search and the Vehicle Routing Problem

Tabu search is attributed to Fred W. Glover; it is a mathematical optimization method involving a memory structure. Tabu search belongs in the class of local search techniques, after finding local optima the solution is marked as tabu (taboo) so that the algorithm does not visit that possibility in the near future [27]. Tabu search is an iterative search method where the current solution is replaced by a random solution from the neighborhood of the current solution. The solutions in the neighborhood are selected by a memory structure, i.e. the tabu list.

The vehicle routing problem was first introduced by Dantzig and Ramser in 1959 [20]. It often involves delivering goods from a central location to customers; the goal is to minimize the cost of distributing the goods. In this paper a slightly different version is used, where there is no central depot so the routes are not overlapping. The vehicle routing problem is related to the traveling salesman problem, but instead of having only one salesman, we have multiple vehicles that must visit all the customers (or cities). Since we do not have any depots, the goal is to split the vertices (or customers) between the vehicles and minimize the length of the maximum cost tour over all the vehicles. The input is a complete undirected graph $G = (V, E)$ with a nonnegative cost c_e associated with each edge $e \in E$. Let $V_1, \dots, V_k$ be a partition of the vertices in V, $\bigcup_{i=1}^{k} V_i = V$ and $V_i \cap V_j = \emptyset$, where k is the number of vehicles, and let $G_i = (V_i, E_i)$ be the subgraph induced by the subset of the vertices V_i. The objective of the vehicle routing problem is to find a partition $V_1, \dots, V_k$ of the vertices, find a tour (or a Hamiltonian cycle) for each induced subgraph $G_i = (V_i, E_i)$ and minimize the total maximum tour length over the tours for each subgraph:

$$\min \max_{i=1}^{k} \left(\sum_{e \in T_i} c_e\right) \qquad (4)$$

$$s.t.\ \ V_1, \dots, V_k \ is\ a\ partition\ of\ V$$

$$T_i\ is\ a\ tour\ of\ G_i = (V_i, E_i)$$

This paper uses no central depot for the vehicle routing problem; consequently each route is totally independent and in many ways similar to the traveling salesman problem. The number of routes is determined beforehand and the objective is to minimize the length of the longest route. The neighborhood was computed by taking a random part of the current route and inverting it. There was also a small probability of a city from one route switching places with a city in another route and a small probability for a city to be removed from one route and inserted into another route. After evaluating the neighborhood the best route was selected from the neighborhood and added to a shared tabu list. For the VR problem we used the same instance as for the TS problem and can be seen in Figure 5.

2.4 Implementation

Although multi core processors have been mainstream for quite some time, not many software developers are programming to use their full power. In recent years companies like Intel, Microsoft and the Java open community have been creating tools to make it easier for the everyday programmer to start programming for the multi core CPUs [21][22][23].

The .NET4 platform from Microsoft comes with a set of tools for parallel programming, in this paper the .NET4 platform is used to program both the single- and multi core implementations for all the algorithms. Using such a high level language may not give the best performance but it makes the coding easier.

3 Experiments and analysis

To get a good comparison for all algorithms two computers were used, they will be called computer A and B in this paper. Computer A runs a dual core AMD CPU and computer B runs a quad core Intel CPU. Both computers are fairly basic although the CPUs come from two different vendors and computer B is considerably more powerful. These two systems have very different architectures and one of the key differences is that the AMD processor has integrated cache memory for each core while the Intel processor has one shared cache memory for all cores [24][25]. This key difference between the CPUs may have a significant effect on the results depending on which algorithm is being used. A more detailed description for both computers can be found in Appendix A. Computer A with a dual core CPU will be used to generate results for both the single core implementations and the dual core implementations. Computer B with a quad core CPU will be used to generate results for both the single core and the quad core implementations. To get accurate results, each algorithm is run as often as possible in a reasonable timeframe, the number of iterations for each algorithm depends on the running time of the algorithm. Also, more iterations were run on computer B since it was considerably faster than computer A.

3.1 Simulated Annealing and the Ackley Function

Simulated Annealing was selected because of its popularity and simplicity. It can be implemented in a delightfully parallel manner which means that the whole algorithm can be run on a single thread with no dependence on other threads [16]. The Ackley function was selected to test the algorithm because of its relative simplicity and scalability. To get a good comparison the algorithm was run for

15000 iterations on computer A and 100000 iterations on computer B. Table 1 gives an overview of the end results, according to the results the average gain for a new core is a factor of 1.6. A statistical test, so called t-Test, was performed; it concluded that there is not a statistical difference in the quality of the solution in the two implementations, i.e. single or multi core.

Table 1 Results for computer A and B for the Simulated Annealing algorithm running the Ackley function. According to the results the average gain for a new core is a factor of 1.6.

Computer A	Single core	Multi core
Time for 15000 iterations	7h 51m	4h 39m
Average running time pr. iteration	1,9s	1,1s
Computer B		
Time for 100000 iterations	28h 47m	9h 9m
Average running time pr. iteration	1,0s	0,3s

3.2 Ant Colony and the Traveling Salesman Problem

While the Simulated Annealing algorithm was implemented as delightfully parallel, the Ant Colony algorithm was implemented more as a true parallel algorithm with shared memory structures. The work is divided into chunks which takes a relatively long time to calculate followed by a short use of shared resources. The Ant Colony algorithm was run for 100 iterations for Computer A and 1000 iterations for Computer B. Table 2 gives an overview of the overall results for the Ant Colony algorithm.

Table 2 Results for computer A and B for the Ant Colony algorithm running on both single- and multi core implementations.

Computer A	Single core	Multi core
Time for 100 iterations	2h 5m	1h 51m
Average running time pr. iteration	1m 15s	1m 7s
Computer B		
Time for 1000 iterations	7h 32m	2h
Average running time pr. iteration	27,1s	7,2s

3.3 Tabu Search and the Vehicle Routing Problem

The last algorithm selected for this paper is Tabu Search which was implemented with a shared tabu list. The work is divided into chunks which takes a relatively short time to calculate followed by a high use of shared resources. The Tabu Search algorithm was run for 100 iterations for Computer A and 1000 iterations for Computer B. Table 3 gives an overview of the overall results for the Tabu Search algorithm.

Table 3 Results for computer A and B for the Tabu Search algorithm running on both single- and multi core implementations.

Computer A	Single core	Multi core
Time for 100 iterations	1h 3m	51m
Average running time pr. iteration	38s	31s
Computer B		
Time for 1000 iterations	4h 57m	6h 56m
Average running time pr. iteration	17,8s	25,0s

4 Overall results

4.1 Simulated Annealing and the Ackley Function

The Simulated Annealing algorithm is delightfully parallel, often called embarrassingly parallel, which means that the whole algorithm can be run on a single thread with no dependence on other threads [16]. This is in fact the best way to implement the algorithm in parallel. The only difference between the single core and the multi core implementation was that only one core was used to run the algorithm in the single core implementation but as many cores as are available are used in the multi core. A typical search path can be viewed in Figure 6, as can be seen it spans a good portion of the search space leaving few areas unsearched. The Simulated Annealing algorithm had to be very thorough in order to get sufficient workload on the CPU to get a good comparison between the single- and multi core implementations.

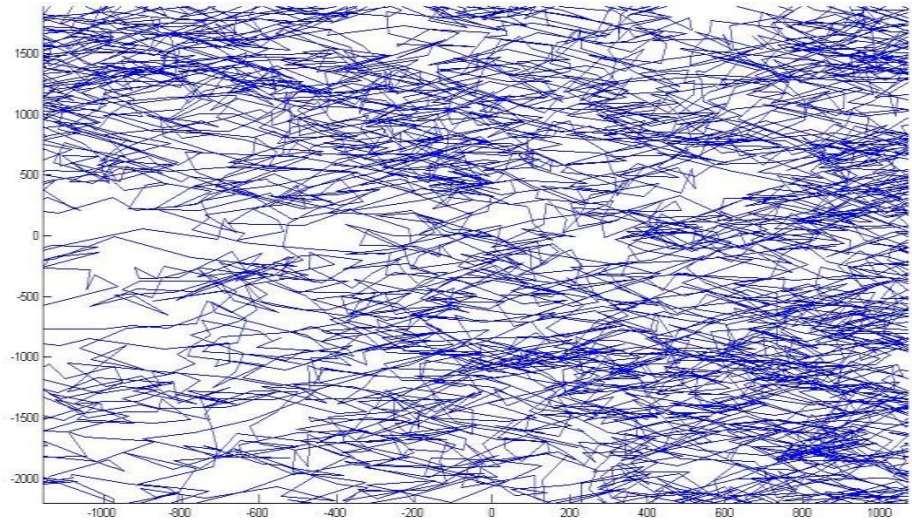

Figure 6 This is a typical search path for the Simulated Annealing algorithm over the Ackley function. The image zooms in around the zero point.

Running this example on computer A (see appendix A) gives a clear indication of how much performance gain can be realized by using multi core processors. By using Visual Studio to get a visualization of the behavior of both the multi and single core implementations we get the images given in Figure 7 and 8. Each figure is of 100 iterations.

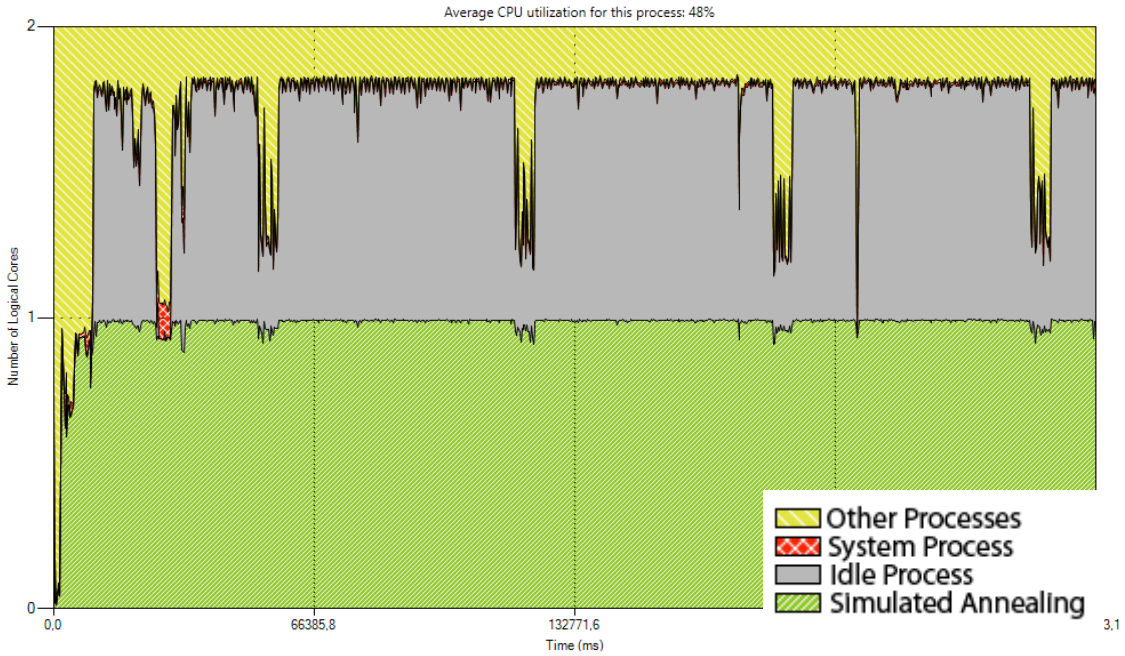

Figure 7 In this figure one can see how one core is almost completely used for the single core implementation of the Simulated Annealing algorithm while the other is mostly idle.

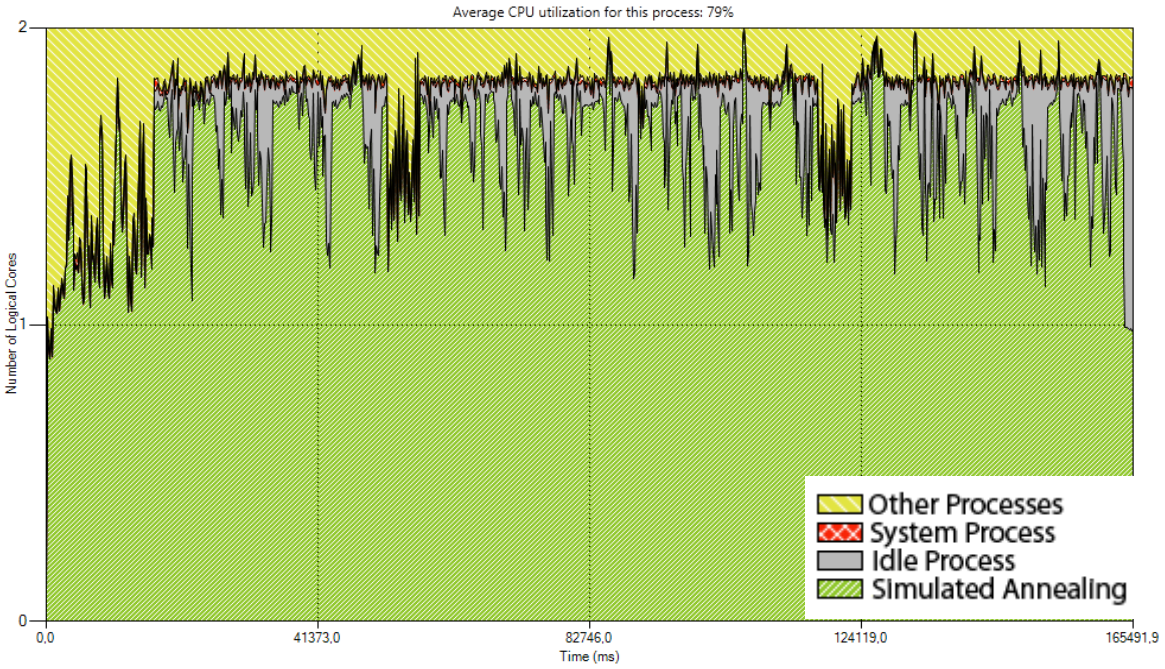

Figure 8 In this figure one can see how both cores are used to run the multi core implementation of the Simulated Annealing algorithm and the idle process is run much less than in Figure 4.

Figures 7 and 8 show clearly how the computer utilizes each core both in the single- and multi core implementations. The yellow part depicts other processes of which Visual Studio plays the biggest part; Visual Studio had to run to generate the Figures. System processes are shown as red but whose importance is negligible. The two most interesting processes are the idle process and the simulated annealing program depict with gray and green respectably.

As was to be expected with the single core implementation one core is almost completely devoted to the simulated annealing program while the other is not used at all. At the same time the other core is

used to completely carry other processes although they do not count for much. One interesting point is the 48% usage of the processor for the single core implementation as opposed to 79% in the multi core, this account for the percentage the simulated annealing program uses. The number is not doubled mainly because of the Visual Studio process and other processes, although the processor could have been better utilized as the amount of the idle process in Figure 7 gives evidence to.

It is obvious by the numbers given in Table 1 that computer B is much faster than computer A, which does not come as a surprise, but it is also obvious that the multi core version is much faster in both cases. Figure 9 shows the average running time of iteration in milliseconds for both the single- and multi core implementations of the Simulated Annealing algorithm on both computer A and B.

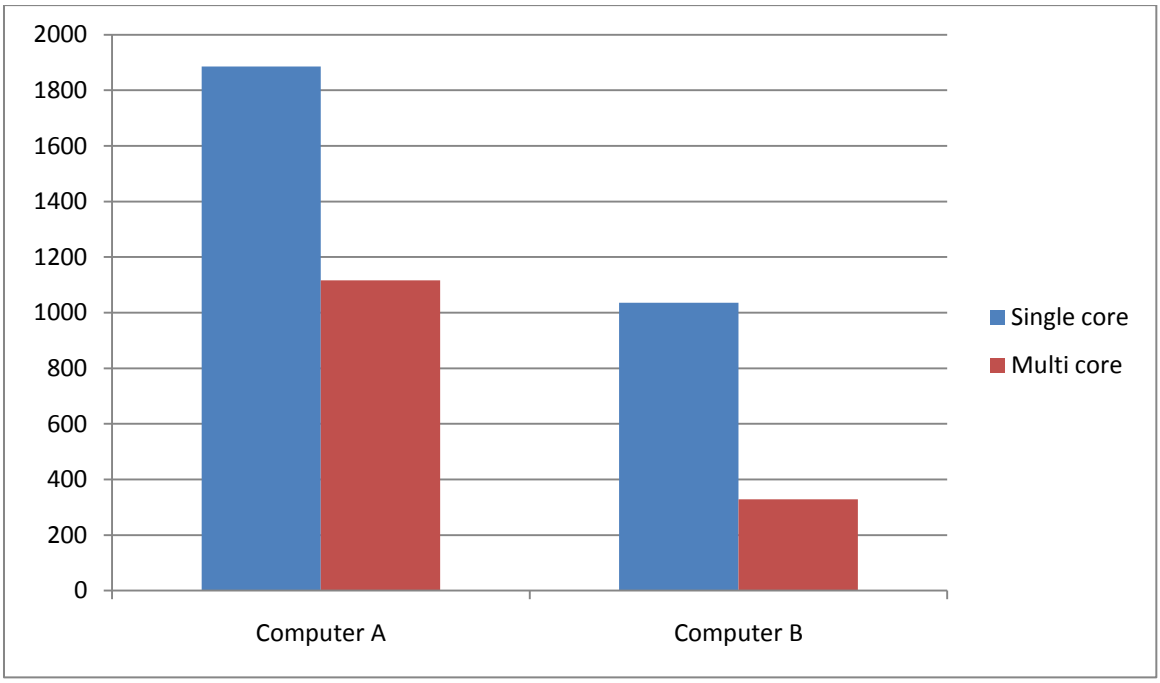

Figure 9 The average running time of an iteration in milliseconds for both the single- and multi core implementations of the Simulated Annealing algorithm on computers A and B.

The two computers used to test the algorithms are very different which makes it difficult to compare the result for the single core, dual core and the quad core implementations. To do so the results for the single core implementations were scaled to one and the multi core implementation for each machine was scaled accordingly, as shown in Figure 10. According to the scaled result the dual core implementation is just about 1.7 times as fast as the single core; the quad core on the other hand is about 3.14 times as fast as the single core.

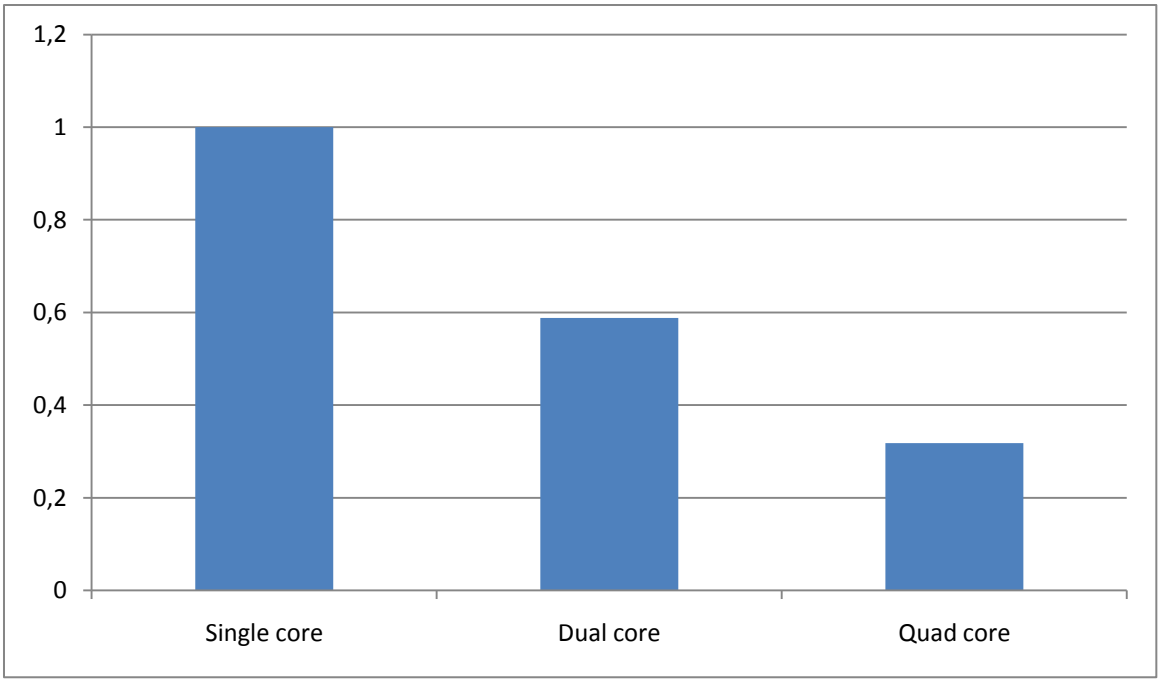

Figure 10 The scaled running time for the single- dual- and quad core implementations of the Simulated Annealing algorithm. According to the scaled result the dual core implementation is just about 1.7 times as fast as the single core; the quad core on the other hand is about 3.14 times as fast as the single core.

4.2 Ant Colony and the Traveling Salesman Problem

The Ant Colony algorithm can effectively be parallelized in many ways. The implementation used in this paper runs each ant in parallel as opposed to just running an independent instance on each core. All cores use the same pheromone table which means that shared resources are used. When using shared resources there is a greater risk of conflicts between the threads, such as one thread waiting for another, which is why such an algorithm is trickier to implement.

As can be seen in Table 2 the dual core implementation does not give the same performance improvements as the Simulated Annealing implementation, but the quad core implementation is if anything, even faster. In either case this is a solid improvement. The difference between the systems can be seen in Figure 11 as a scaled comparison. Figure 12 gives an even better view of the difference between the single- and the multi core implementations on each computer. The big speed difference between the dual core and quad core CPU's is most likely something to do with their design and clock speed.

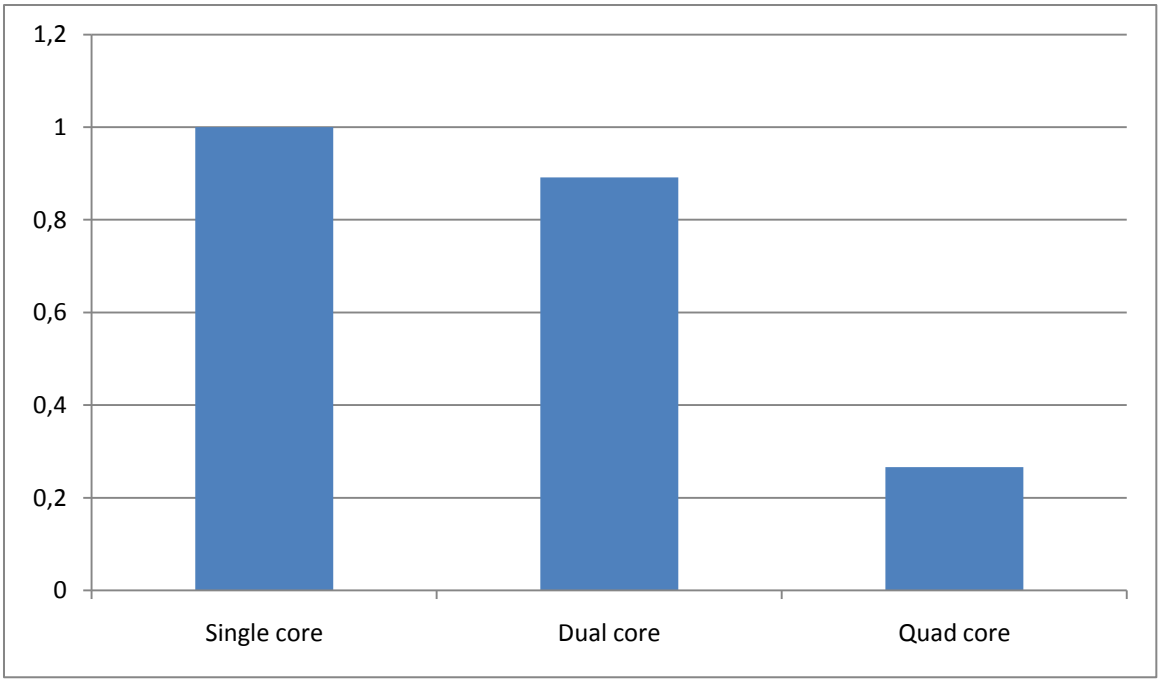

Figure 11 The scaled running time for the single- dual- and quad core implementations of the Ant Colony algorithm.

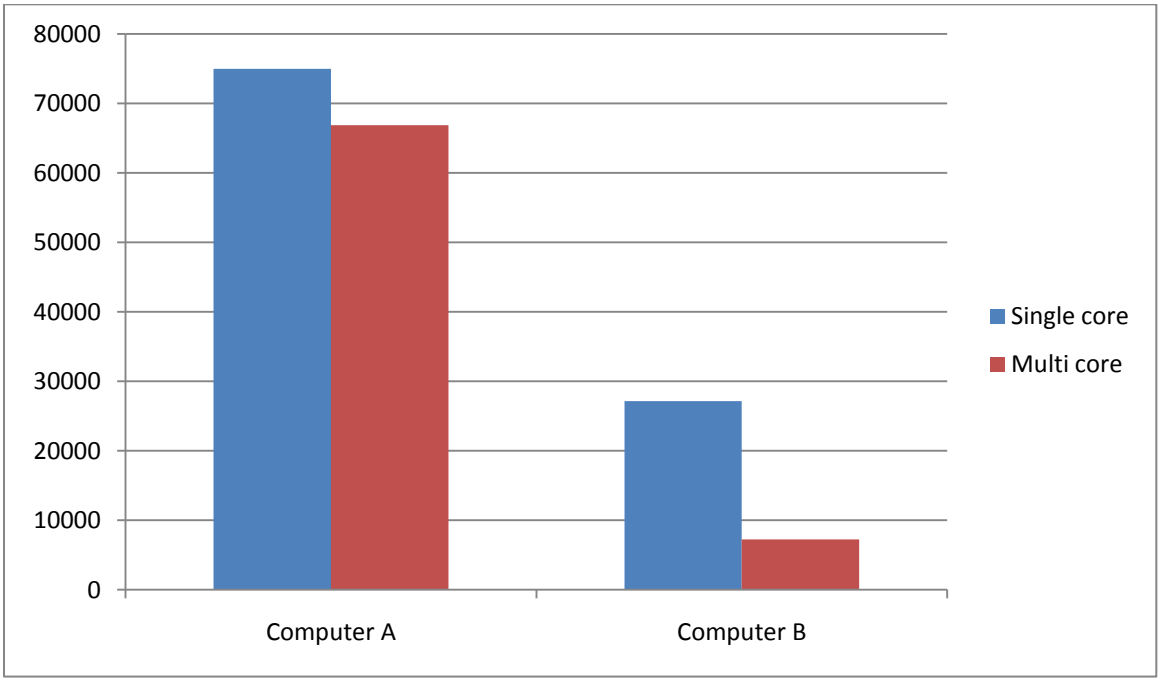

Figure 12 The average running time of an iteration in milliseconds for both the single- and multi core implementations of the Ant Colony algorithm on both computer A and B.

4.3 Tabu Search and the Vehicle Routing Problem

Tabu Search can be implemented as delightfully parallel just as well but for this paper the parallel implementation shared one tabu list with all cores to test the multi core use of a shared memory.

As Table 3 gives evidence to, there is a clear speed difference on the single- and multi core implementation but in contrast to the other two problems we have looked at the multi core

implementation is slower on computer B. Since all cores are using the same memory structure both to read and write, some sort of locking mechanism needs to be in place. Locking works by allowing only one thread to use a block of code at a time. Locking is always time consuming and in this case it is one of the reasons why the multi core implementation is so much slower. To get a clear picture of how much impact the locking mechanism has, an implementation with no locking mechanism was tried as can be seen in Figure 14.

The speed difference is quite startling, as can be seen in Figure 13 where both single core implementations have been scaled to 1 for each computer. The dual core implementation on computer A is about 20% faster than the single core implementation but the quad core implementation on computer B is about 40% slower.

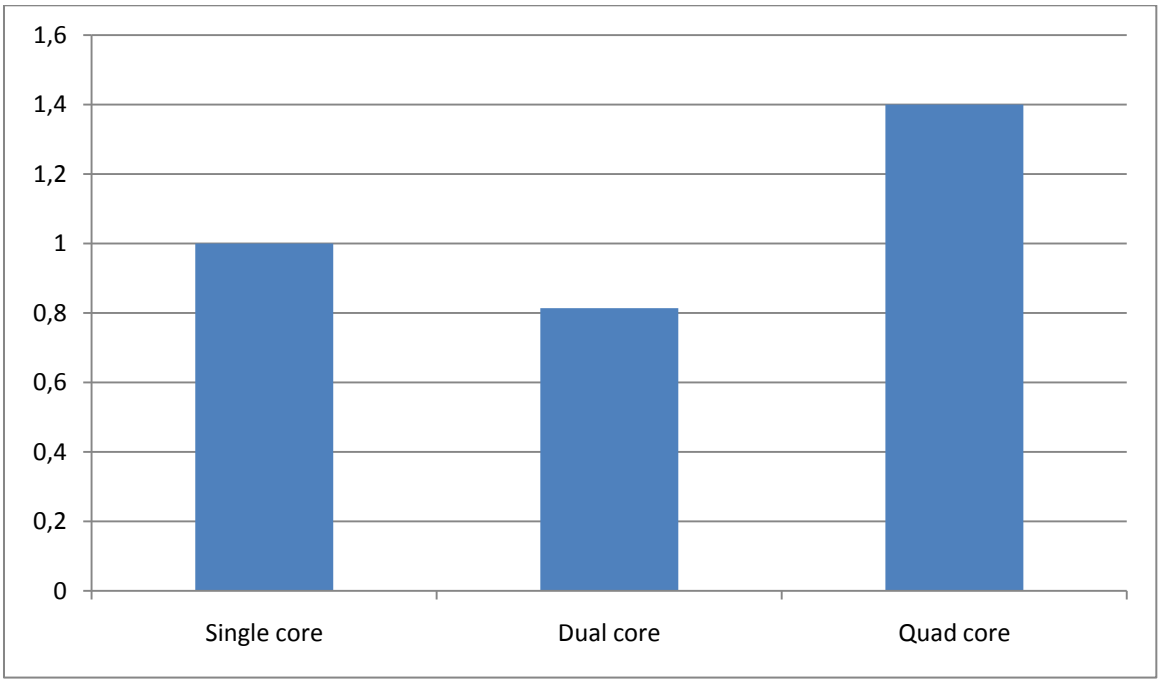

Figure 13 The scaled running time for the single- dual- and quad core implementations of the Tabu Search algorithm. The dual core implementation is about 20% faster than the single core implementation but the quad core implementation is about 40% slower. These results are very interesting; one would expect the quad core implementation to be much faster than the single core implementation.

As can be seen in Figure 14 computer B is much faster than computer A, while there is some speed increase in the multi core implementation on computer A there is actually a speed decrease on computer B. One reason for this is as mentioned earlier due to the locking mechanism, but the same type of locking mechanism is also used in the multi core implementation on computer A, it is in fact the same code. As mentioned earlier there is a key difference between the two CPUs, the CPU on computer A has a cache memory for each core while the CPU for computer B has a shared cache memory for all cores, this may be causing more locks on computer B which slows the process down. This may also be causing computer B to use the main memory. If for example the tabu list is too large to fit into cache memory, the main memory will be used, that might be one reason behind the performance decrease on computer B. The tabu list was connected to the size of the CPU, thus the tabu list on computer B was larger than the tabu list on computer A, it might even have been too large for the cache memory on computer B. A very important point is that the CPUs are from rival competitors and have a very different architecture and instruction sets.

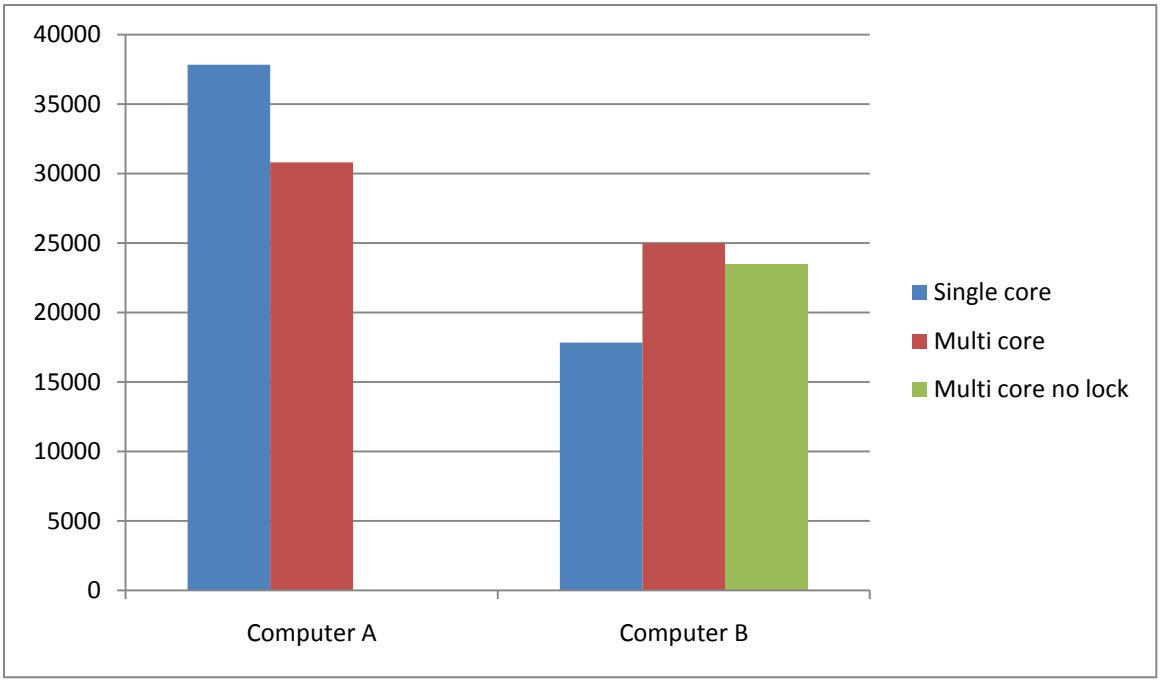

Figure 14 The average running time of an iteration in milliseconds for both the single- and multi core implementations of the Tabu Search algorithm on both computer A and B. To get a better comparison the algorithm was also tried without the locking mechanism.

4.4 Multi core implementations

There is a considerable difference between systems for the Ant Colony algorithm, although there is a performance increase on both computer A and B, computer B shows a considerably more increase than computer A. This means this implementation is much better suited for the kind of system represented by computer B. The implementation is characterized by heavy calculations on each core with heavy reading and light writing in a shared memory. This implementation is ideal for the multi core environment given that the CPU is both fast and has a large integrated cache memory, which computer B has.

Both the Simulated Annealing and the Ant Colony multi core implementations give a solid improvement over their single core counterpart but the Tabu Search implementation on the other hand does not, the reason being the relatively light computations on each core and uses shared resources between computations. There was a difference between computer A and B, while there were some performance gain on computer A, computer B was considerably slower for the multi core implementation, the reason being that computer B was faster with the computations and had to spend more time waiting for the locking mechanism. This implementation is not good for the multi core environment and the delightfully parallel implementation would most likely have performed better for the Tabu Search algorithm.

To get a better understanding of the problem the Tabu Search algorithm was implemented without the locking mechanism as well. As Figure 14 clearly shows there was some speed increase but not enough to explain the poor performance of the algorithm. There are some possible reasons for the poor performance of the algorithm, one being the relatively small amount of work being done on each core compared to the time it takes to work on the tabu list. Another reason might be related to the hardware architecture but that is beyond the scope of this text.

5 Conclusions and future work

In this paper we looked at three different algorithms and three different multi core implementations. First we looked at the delightfully parallel implementation which means that the whole algorithm can run on a single core with no dependencies to other cores, this design pattern gives a solid performance gain as long as the computations being performed are relatively time consuming. The second multi core implementation can be described as a shared memory implementation with almost no locking mechanisms but with time consuming computations on each core. Finally third implementation that was looked at can be described as a shared memory implementation depending heavily on locking mechanisms and relatively little computations on each core. The third algorithm was also implemented without the locking mechanism; it was for the most part the same algorithm apart from the locking mechanism.

After writing and testing the three algorithms it is quite evident that using the parallel toolkits available in the high level languages can give a drastic performance increase. In many cases it does not take much work to implement the parallel code but it has to be done correctly and the CPU architecture does matter a great deal as can be seen in both the Ant Colony and Tabu Search algorithms. One must be careful when using locking mechanisms; both AC and TS algorithms use one memory structure between cores but where no locking mechanism was required for the AC implementation it had considerable effect on the TS parallel implementation on computer B.

By programming especially for multi core processors the performance of meta-heuristic algorithms can be improved without much additional cost or effort. The results depend on the algorithm chosen, the implementation (sharing memory structures, locks) and the system architecture. To get a solid performance gain, a delightfully parallel implementation is a very good choice but ideally one should select the correct algorithm and hardware for each problem and tune the algorithm for the selected hardware.

Interesting future work would be to compare algorithm performance to hardware and find out the right combinations for popular problems. Intuitively one tabu list for all cores should give better solutions and be faster, this was not the case for this paper. It would be interesting work to find the reason why better solutions were not realized and optimize the algorithm for multi core architecture.

6 References

[1] D. Geer, “Chip Makers Turn to Multicore Processors,” Computer, Vol. 38, No. 5, May 2005, pp. 11–13.
[2] J. Parkhurst, J. Darringer, and B. Grundmann, “From Single Core to Multi-Core: Preparing for a New Exponential,” Proc. of the 2006 IEEE/ACM International Conference on Computer-Aided Design, 2006, pp. 67–72.
[3] D. Patterson and J. Hennessy, “Computer Organization and Design (2nd Edition),” Morgan Kaufmann Publishers, 1998.
[4] The Beowulf Project. http://www.beowulf.org. Last accessed March 2009.
[5] MPI - The Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi/. Last accessed March 2009.
[6] OpenMP Architecture Review Board. http://www.openmp.org/specs/. Last accessed March 2009.

[7] Z. Michalewicz and D. B. Fogel, “How to Solve It: Modern Heuristics (2nd Edition),” Springer, 2004.
[8] F. S. Hillier and G. J. Lieberman, “Introduction to Operations Research (8th Edition),” McGraw-Hill, 2005.
[9] T. N. Bui, T. Nguyen, and J. R. Rizzo Jr., “Parallel Shared Memory Strategies for Ant-Based Optimization Algorithms,” Proc. of the 2009 Genetic And Evolutionary Computation Conference, 2009, pp. 1–8.
[10] G. E. Moore, “Cramming More Components onto Integrated Circuits,” Electronics, 1965.
[11] The TRACKER Project. http://tracer.lcc.uma.es/problems/ackley/ackley.com. Last accessed February 23 2010.
[12] A. Colorni, M. Doringo, and V. Maniezzo, “Distributed Optimization by Ant Colonies,” European Conference on Artificial Life, Elsevier Publishing 1992, pp. 134–142.
[13] M. Doringo, V. Maniezzo and A. Colorni, “Ant System: An Autocatalytic Optimizing Process,” Technical Report, Politecnico di Milano, Italy, 1992
[14] History of the TSP. http://www.tsp.gatech.edu/history/index.html. Last accessed august 24 2010.
[15] About Ant colony optimization. http://iridia.ulb.ac.be/~mdorigo/ACO/about.html. Last accessed May 11 2010.
[16] S. Toub, “Patterns of Parallel Programming,” Microsoft Corporation, 2009.
[17] V. Maniezzo, L. M. Gambardella, F. de Luigi, “Ant Colony Optimization,” New optimization techniques in engineering, Springer, pp. 102–121.
[18] S. Kirkpatrick, C. D. Gelatt, M. P. Vecci, “Optimization by Simulated Annealing,” Science. New Series 220, American Association for the Advancement of Science 1983, pp. 671-680.
[19] V. Černý, “A Thermodynamical Approach to the Travelling Salesman Problem: an efficient simulation algorithm,” Journal of Optimization Theory and Applications 1985, pp. 41-51.
[20] G. B. Dantzig, J. H. Ramser, “The Truck Dispatching Problem,” Management Science 6, INFORMS 1959, pp. 80-91.
[21] Intel Parallel Studio. http://software.intel.com/en-us/intel-parallel-studio-home/. Last accessed July 5 2010.
[22] Java Parallel Processing Framework. http://www.jppf.org/. Last accessed July 5 2010.
[23] .NET Parallel Computing. http://msdn.microsoft.com/en-us/concurrency/default.aspx. Last accessed July 5 2010.
[24] Softpedia. http://news.softpedia.com/news/AMD-Unveils-Energy-Efficient-Athlon-X2-4850e-Dual-Core-CPU-80170.shtml. Last accessed July 15 2010.
[25] Intel processors. http://ark.intel.com/Product.aspx?id=41447. Last accessed September 04 2010.
[26] POSIX threads explained. http://www.ibm.com/developerworks/linux/library/l-posix1.html. Last accessed august 22 2010.
[27] An introduction to Tabu Search.
http://www.ifi.uio.no/infheur/Bakgrunn/Intro_to_TS_Gendreau.htm. Last accessed August 25 2010.
[28] The history of Operations Research.
http://www.britannica.com/EBchecked/topic/682073/operations-research/68171/History#ref22348. Last accessed September 13 2010.
[29] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev,
M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich, “An Analysis of Linux Scalability to Many Cores,” MIT CSAIL 2010

Appendix A

A.1 Computer A

Computer A has the following parameters:

Processor:	AMD Athlon™ Dual Core Processor 4850e 2.50 GHz
Installed memory (RAM):	2.00 GB
Operating System:	Microsoft Windows 7 Enterprise
System type:	32-bit Operating System

A.2 Computer B

Computer B has the following parameters:

Processor:	Intel® Core™ i7 Quad Core Processor 2.8 GHz
Installed memory (RAM):	4.00 GB
Operating System:	Microsoft Windows 7 Professional
System type:	64-bit Operating System

Appendix B

B.1 The Simulated Annealing code

```
public class SA
  {
    #region Data field

    Random rnd = new Random(DateTime.Now.Millisecond);
    BestSolution bestSolution;
    int nRepetativeIterationCounter = 0;
    int nIterationCounter = 0;
    double proba;
    double delta;
    double solution;

    #endregion

    #region Public stuff

    public List<List<double>> parameterPath = new List<List<double>>();
    public List<double> solutionPath = new List<double>();

    #endregion

    #region Methods

    public CommonObjects.Result StartAnnealingParam(List<double> parameterList, CommonObjects.Interface.IData
Data)
    {
      return AnnealingParam(parameterList, Data, 0.999, 1000.0, 0.001, true);
    }

    public CommonObjects.Result StartAnnealingParam(List<double> parameterList, CommonObjects.Interface.IData
Data, double alpha, double temperature, double epsilon, bool getParamList)
    {
      return AnnealingParam(parameterList, Data, alpha, temperature, epsilon, getParamList);
    }
```

```
    private CommonObjects.Result AnnealingParam(List<double> parameterList, CommonObjects.Interface.IData
Data, double alpha, double temperature, double epsilon, bool getParamList)
    {
      if (Data == null) throw new ArgumentNullException("Data not initialized!");

      List<double> currentParameters = parameterList;
      List<double> nextParameters;

      solution = Data.ComputeSolution(currentParameters);
      bestSolution = new BestSolution(solution, currentParameters[0], currentParameters[1]);
      Stopwatch sw = new Stopwatch();

      sw.Start();
      //while the temperature didnt reach epsilon
      while (temperature > epsilon)
      {
        ++nIterationCounter;
        if (getParamList)
        {
          //keep track of the param path
          parameterPath.Add(new List<double>(currentParameters));
        }
        //get the a random neighbour
        nextParameters = Data.ComputeNextNeighbour(currentParameters);
        //compute the distance of the new permuted configuration
        delta = Data.ComputeSolution(nextParameters) - solution;
        //if the new solution is better accept it and assign it
        if (delta < 0)
        {
          currentParameters = new List<double>(nextParameters);
          solution = delta + solution;
        }

        else
        {
          proba = rnd.NextDouble();
          //if the new solution is worse accept it but with a probability level
          //if the probability is less than E to the power -delta/temperature.
          //otherwise the old value is kept
          if (proba < Math.Exp(-delta / temperature))
          {
            currentParameters = new List<double>(nextParameters);
            solution = delta + solution;
          }
        }

        //cooling proces on every iteration
        temperature = Data.ComputeTemperature(temperature, alpha, nIterationCounter);

        //if same distance is five times in a row stop
        if (Math.Abs(bestSolution.solution - solution) < epsilon)
          nRepetativeIterationCounter++;
        else
        {
          nRepetativeIterationCounter = 0;

          if (bestSolution.solution > solution)
            bestSolution = new BestSolution(solution, currentParameters[0], currentParameters[1]);
        }

        if (nRepetativeIterationCounter == 1000)
        {
          sw.Stop();
          return new CommonObjects.Result(bestSolution.solution, new List<double>(new double[]
{bestSolution.x1,bestSolution.x2}), parameterPath, sw.Elapsed);
        }
```

```
        }

        sw.Stop();
        return new CommonObjects.Result(bestSolution.solution, new List<double>(new double[] { bestSolution.x1,
bestSolution.x2 }), parameterPath, sw.Elapsed);
    }

    #endregion
  }

  public class BestSolution
  {
    public double solution;
    public double x1;
    public double x2;

    public BestSolution(double nSolution, double nX1, double nX2)
    {
      solution = nSolution;
      x1 = nX1;
      x2 = nX2;
    }
  }
```

B.2 Single core implementation

```
static void Main(string[] args)
    {
      int SIZE = 10;
      bool WriteParamPath = false;
      bool WriteReport = false;
      bool WriteFullReport = false;
      string ParamPath = string.Empty;
      string ReportPath = string.Empty;
      Stopwatch sw = new Stopwatch();
      Random rand = new Random(DateTime.Now.Millisecond);
      List<double> initialSolution = new List<double>(new double[] { 32264, 27800 });


      if (!CommonObjects.CommunicationManager.StartProgram("Singlecore implementation", out SIZE, out
WriteReport, out ReportPath, out WriteFullReport, out WriteParamPath, out ParamPath))
        return;

      List<CommonObjects.Result> resultList = new List<CommonObjects.Result>(SIZE);

      sw.Start();
      for (int i = 0; i < SIZE; i++)
      {
        SimulatedAnnealing.SA annealing = new SimulatedAnnealing.SA();
        CommonObjects.Ackley data = new CommonObjects.Ackley();

        CommonObjects.Result result = annealing.StartAnnealingParam(initialSolution, data, 0.99999, 100000,
0.0000001, WriteParamPath);
        if (!WriteParamPath) result.ParameterList = null;
        resultList.Add(result);

        if (WriteParamPath)
        {
          StreamWriter writer = new StreamWriter(string.Format("{0}\\solutionPath{1}.txt", ParamPath, i));
          foreach (var item in resultList[i].ParameterList)
          {
            writer.WriteLine(string.Concat(item[0].ToString().Replace(",", ".").Replace("E", "e"), " ",
item[1].ToString().Replace(",", ".").Replace("E", "e")));
          }
          writer.Flush();
```

```
            writer.Close(); writer.Dispose();
          }

          annealing = null;
          data = null;
          result = null;
        }
        sw.Stop();

        Console.WriteLine(string.Format("Time to compute: {0}ms", sw.ElapsedMilliseconds));

        if (WriteReport)
        {
          CommonObjects.CommunicationManager.WriteReport(WriteFullReport, ReportPath, resultList, sw);
        }
```

B.3 Multi core implementation

```
static void Main(string[] args)
    {
        int SIZE = 10;
        bool WriteParamPath = false;
        bool WriteReport = false;
        bool WriteFullReport = false;
        string ParamPath = string.Empty;
        string ReportPath = string.Empty;
        Stopwatch sw = new Stopwatch();
        Random rand = new Random(DateTime.Now.Millisecond);
        List<double> initialSolution = new List<double>(new double[] { 32264, 27800 });

        if (!CommonObjects.CommunicationManager.StartProgram("Multicore implementation", out SIZE, out
WriteReport, out ReportPath, out WriteFullReport, out WriteParamPath, out ParamPath))
          return;

        List<CommonObjects.Result> resultList = new List<CommonObjects.Result>(SIZE);
        Task[] tasks = new Task[SIZE];

        sw.Start();
        for (int n = 0; n < SIZE; n++)
        {
          int i = n;
          tasks[n] = Task.Factory.StartNew(() =>
            {
              SimulatedAnnealing.SA annealing = new SimulatedAnnealing.SA();
              CommonObjects.Ackley data = new CommonObjects.Ackley();

              CommonObjects.Result result = annealing.StartAnnealingParam(initialSolution, data, 0.99999, 100000,
0.0000001, WriteParamPath);
              if (!WriteParamPath) result.ParameterList = null;
              resultList.Add(result);

              if (WriteParamPath)
              {
                StreamWriter writer = new StreamWriter(string.Format("solutionPath{0}.txt", i));
                foreach (var item in resultList[i].ParameterList)
                {
                  writer.WriteLine(string.Concat(item[0].ToString().Replace(",", ".").Replace("E", "e"), " ",
item[1].ToString().Replace(",", ".").Replace("E", "e")));
                }
                writer.Flush();
                writer.Close(); writer.Dispose();
              }

              annealing = null;
```

```
                data = null;
                result = null;
            });
        }

        Task.WaitAll(tasks);
        sw.Stop();

Console.WriteLine(string.Format("Time to compute: {0}ms", sw.ElapsedMilliseconds));

        if (WriteReport)
        {
            CommonObjects.CommunicationManager.WriteReport(WriteFullReport, ReportPath, resultList, sw);

        }

        }
    }
```

Appendix C

C.1 The Ant Colony code

```
public class AC
  {
    #region Datafeild

    int nNumberOfCities = 0;
    double[,] nPheremones;
    double[,] nVisibility;
    int nTotalNumberOfAnts = 0;
    int[,] nAntTours;
    CommonObjects.AntTour[] nAntTourLenght;
    int nIterationCounter = 0;
    int[] nStartingCities;
    double PheromoneConstant = 10;
    CommonObjects.AntTour[] BestSolutions;

    #endregion

    #region Properties

    public int NumberOfIterations { get; set; }

    public int AntsPerCity { get; set; }

    public double Alpha { get; set; }

    public double Beta { get; set; }

    public double RateOfEvaporation { get; set; }

    #endregion

    public AC(int nNumberOfIterations, int nAntsPerCity, double nAlpha, double nBeta, double nRateOfEvaporation,
double nPheromoneConstant)
    {
        NumberOfIterations = nNumberOfIterations;
        AntsPerCity = nAntsPerCity;
        Alpha = nAlpha;
        Beta = nBeta;
        RateOfEvaporation = nRateOfEvaporation;
        PheromoneConstant = nPheromoneConstant;
```

```
    }

    public CommonObjects.AntTour[] StartAntColony(double nInitialPheromone, double[,] nDistances)
    {
      nNumberOfCities = nDistances.GetLength(1);
      nPheremones = new double[nNumberOfCities, nNumberOfCities]; for (int counti = 0; counti <
nPheremones.GetLength(1); counti++) { for (int countj = 0; countj < nPheremones.GetLength(1); countj++) {
nPheremones[counti, countj] = nInitialPheromone; } }
      nVisibility = new double[nNumberOfCities, nNumberOfCities]; for (int counti = 0; counti <
nVisibility.GetLength(1); counti++) { for (int countj = 0; countj < nVisibility.GetLength(1); countj++) {
nVisibility[counti, countj] = 1.0 / nDistances[counti, countj]; } }
      nTotalNumberOfAnts = AntsPerCity * nNumberOfCities;
      nAntTourLenght = new CommonObjects.AntTour[nTotalNumberOfAnts];
      nStartingCities = new int[nTotalNumberOfAnts]; for (int i = 0; i < nTotalNumberOfAnts; i++) { nStartingCities[i]
= Convert.ToInt32(Math.Floor(Convert.ToDouble(i) / AntsPerCity)); }
      BestSolutions = new CommonObjects.AntTour[NumberOfIterations];

      while (nIterationCounter < NumberOfIterations)
      {
        nAntTours = new int[nTotalNumberOfAnts, nNumberOfCities];
        for (int i = 0; i < nTotalNumberOfAnts; i++)
        {
          // Find a tour for ant i
          int[] nCities = new int[nNumberOfCities]; for (int j = 0; j < nNumberOfCities; j++) { nCities[j] = j; }
          nCities[nStartingCities[i]] = -1;

          nAntTours[i, 0] = nStartingCities[i];
          for (int j = 1; j < nNumberOfCities; j++)
          {
            int nNextCity = FindNextCityForAntColony(nAntTours[i, j], nCities);
            nAntTours[i, j] = nNextCity;
            nCities[nNextCity] = -1;
          }

          nAntTourLenght[i] = GetTotalPath(nDistances, i);
        }

        double tempShortestTour = nAntTourLenght[0].Length;
        int tempQuickestAnt = 0;
        for (int i = 1; i < nAntTourLenght.Length; i++)
        {
          if (tempShortestTour > nAntTourLenght[i].Length)
          {
            tempShortestTour = nAntTourLenght[i].Length;
            tempQuickestAnt = i;
          }
        }

        BestSolutions[nIterationCounter] = nAntTourLenght[tempQuickestAnt];
        BestSolutions[nIterationCounter].Iteration = nIterationCounter;

        // Update pheremone
        UpdatePheromones(nAntTourLenght[tempQuickestAnt]);
        //UpdatePheromones();
        nIterationCounter++;
      }

      return BestSolutions;
    }

    public CommonObjects.AntTour[] StartAntColonyMC(double nInitialPheromone, double[,] nDistances)
    {
      nNumberOfCities = nDistances.GetLength(1);
      nPheremones = new double[nNumberOfCities, nNumberOfCities]; for (int counti = 0; counti <
nPheremones.GetLength(1); counti++) { for (int countj = 0; countj < nPheremones.GetLength(1); countj++) {
nPheremones[counti, countj] = nInitialPheromone; } }
```

```
        nVisibility = new double[nNumberOfCities, nNumberOfCities]; for (int counti = 0; counti <
nVisibility.GetLength(1); counti++) { for (int countj = 0; countj < nVisibility.GetLength(1); countj++) {
nVisibility[counti, countj] = 1.0 / nDistances[counti, countj]; } }
        nTotalNumberOfAnts = AntsPerCity * nNumberOfCities;
        nAntTourLenght = new CommonObjects.AntTour[nTotalNumberOfAnts];
        nStartingCities = new int[nTotalNumberOfAnts]; for (int i = 0; i < nTotalNumberOfAnts; i++) { nStartingCities[i]
= Convert.ToInt32(Math.Floor(Convert.ToDouble(i) / AntsPerCity)); }
        BestSolutions = new CommonObjects.AntTour[NumberOfIterations];

        while (nIterationCounter < NumberOfIterations)
        {
          nAntTours = new int[nTotalNumberOfAnts, nNumberOfCities];
          Parallel.For(0, nTotalNumberOfAnts, i =>
            {
              // Find a tour for ant i
              int[] nCities = new int[nNumberOfCities]; for (int j = 0; j < nNumberOfCities; j++) { nCities[j] = j; }
              nCities[nStartingCities[i]] = -1;

              nAntTours[i, 0] = nStartingCities[i];
              for (int j = 1; j < nNumberOfCities; j++)
              {
                int nNextCity = FindNextCityForAntColony(nAntTours[i, j], nCities);
                nAntTours[i, j] = nNextCity;
                nCities[nNextCity] = -1;
              }

              nAntTourLenght[i] = GetTotalPath(nDistances, i);
            });

          double tempShortestTour = nAntTourLenght[0].Length;
          int tempQuickestAnt = 0;
          for (int i = 1; i < nAntTourLenght.Length; i++)
          {
            if (tempShortestTour > nAntTourLenght[i].Length)
            {
              tempShortestTour = nAntTourLenght[i].Length;
              tempQuickestAnt = i;
            }
          }

          BestSolutions[nIterationCounter] = nAntTourLenght[tempQuickestAnt];
          BestSolutions[nIterationCounter].Iteration = nIterationCounter;

          // Update pheremone
          UpdatePheromones(nAntTourLenght[tempQuickestAnt]);
          nIterationCounter++;
        }

        return BestSolutions;
      }

      private void UpdatePheromones(CommonObjects.AntTour bestSolution)
      {
        int nPheremonLength = nPheremones.GetLength(1);
        for (int i = 0; i < nPheremonLength; i++)
        {
          for (int j = 0; j < nPheremonLength; j++)
          {
            nPheremones[i, j] = (1 - RateOfEvaporation) * nPheremones[i, j];
          }
        }

        int c1;
        int c2;
        int ant = bestSolution.Ant;
        for (int j = 0; j < bestSolution.VisitCityOrder.Count - 1; j++)
```

```
    {
      c1 = nAntTours[ant, bestSolution.VisitCityOrder[j]];
      c2 = nAntTours[ant, bestSolution.VisitCityOrder[j + 1]];
      nPheremones[c1, c2] = nPheremones[c1, c2] + PheromoneConstant / nAntTourLenght[ant].Length;
      nPheremones[c2, c1] = nPheremones[c1, c2];
    }

    c1 = nAntTours[ant, bestSolution.VisitCityOrder[bestSolution.VisitCityOrder.Count - 1]];
    c2 = nAntTours[ant, bestSolution.VisitCityOrder[0]];
    nPheremones[c1, c2] = nPheremones[c1, c2] + PheromoneConstant / nAntTourLenght[ant].Length;
    nPheremones[c2, c1] = nPheremones[c1, c2];
  }

  private CommonObjects.AntTour GetTotalPath(double[,] nDistances, int index)
  {
    int[] nTempAntTour = new int[nNumberOfCities];
    for (int i = 0; i < nNumberOfCities; i++)
    {
      nTempAntTour[i] = nAntTours[index, i];
    }

    List<int> nAntPath = new List<int>(nNumberOfCities);
    double nTotalPathLength = nDistances[nTempAntTour[nNumberOfCities - 1], nTempAntTour[0]];
    for (int i = 0; i < nNumberOfCities -  1; i++)
    {
      nTotalPathLength += nDistances[nTempAntTour[i], nTempAntTour[i + 1]];
      nAntPath.Add(nTempAntTour[i]);
    }
    nAntPath.Add(nTempAntTour[nNumberOfCities - 1]);

    return new CommonObjects.AntTour(index, nTotalPathLength, nAntPath);
  }

  private int FindNextCityForAntColony(int nCurrentCity, int[] nCities)
  {
    var AvailableCitiesTemp = from ac in nCities where ac >= 0 select ac;
    int AvailableCitiesCount = AvailableCitiesTemp.Count();
    int[] AvailableCities = AvailableCitiesTemp.ToList<int>().ToArray();

    // Extract the visibility and pheromone for the cities that we haven't visited.
    double[] v = new double[AvailableCitiesCount];
    double[] p = new double[AvailableCitiesCount];
    for (int i = 0; i < AvailableCitiesCount; i++)
    {
      v[i] = nVisibility[nCurrentCity, AvailableCities[i]];
    }
    for (int i = 0; i < AvailableCitiesCount; i++)
    {
      p[i] = nPheremones[nCurrentCity, AvailableCities[i]];
    }

    // Calculate the probability vector and scale it.
    double[] x = new double[AvailableCitiesCount];
    double sumX = 0;
    for (int i = 0; i < AvailableCitiesCount; i++)
    {
      x[i] = Math.Pow(v[i], Alpha) * Math.Pow(p[i], Beta);
      sumX += x[i];
    }

    for (int i = 0; i < AvailableCitiesCount; i++)
    {
      x[i] = (x[i] / sumX);
    }

    double rand = (new Random(DateTime.Now.Millisecond)).NextDouble();
```

```
        int index = 0;

        if (x.Length == 0)
            throw new Exception("The vector x should not be empty.");

        while (x[index] < rand)
        {
            rand = rand - x[index];
            index++;
        }

        return nCities[AvailableCities[index]];
    }
  }
```

C.2 Multi core implementation

```
static void Main(string[] args)
    {
        int SIZE = 25000;
        int AntsPerCity = 20;
        double Alpha = 1;
        double Beta = 2;
        double RateOfEvaporation = 0.1;
        double InitialPheromone = 20;
        double PheromoneConstant = 1;
        string InputPath = @"C:\TSP_50.csv";
        string ReportPath = @"C:\Solutions\AntColony\TSP_AC_MC_25000_20.txt";
        Stopwatch sw = new Stopwatch();

        for (int i = 0; i < 10; i++)
        {
            double[,] DistanceMatrix = CommonObjects.CommunicationManager.ReadTSP(InputPath);
            CommonObjects.AntTour[] tours;

            sw.Start();
            AntColony.AC ant_colony = new AntColony.AC(SIZE, AntsPerCity, Alpha, Beta, RateOfEvaporation,
PheromoneConstant);
            tours = ant_colony.StartAntColonyMC(InitialPheromone, DistanceMatrix);
            sw.Stop();


CommonObjects.CommunicationManager.WriteTSPSolution(string.Format(@"C:\Solutions\AntColony\TSP_AC_MC_2
5000_20_{0}.txt", i), tours, sw);

            Console.WriteLine(string.Format("Time to compute: {0}ms", sw.ElapsedMilliseconds));
        }
    }
```

C.3 Single core implementation

```
static void Main(string[] args)
    {
        int SIZE = 300;
        int AntsPerCity = 200;
        double Alpha = 1;
        double Beta = 2;
        double RateOfEvaporation = 0.1;
        double InitialPheromone = 10;
        double PheromoneConstant = 1;
        string InputPath = @"C: \TSP_25.csv";
        string ReportPath = @"C: \Solutions\tempTSP_solution_25_300_200.txt";
        Stopwatch sw = new Stopwatch();
```

```
        if (!CommonObjects.CommunicationManager.StartTSPProgram("Single-core implementation", out SIZE, out
AntsPerCity, out Alpha, out Beta, out RateOfEvaporation, out InitialPheromone, out PheromoneConstant, out
InputPath, out ReportPath))
            return;

        double[,] DistanceMatrix = CommonObjects.CommunicationManager.ReadTSP(InputPath);
        CommonObjects.AntTour[] tours;

        sw.Start();
        AntColony.AC ant_colony = new AntColony.AC(SIZE, AntsPerCity, Alpha, Beta, RateOfEvaporation,
PheromoneConstant);
        tours = ant_colony.StartAntColony(InitialPheromone, DistanceMatrix);
        sw.Stop();

        CommonObjects.CommunicationManager.WriteTSPSolution(ReportPath, tours, sw);

        Console.WriteLine(string.Format("Time to compute: {0}ms", sw.ElapsedMilliseconds));
    }
```

Appendix D

D.1 The Tabu Search code

```
public class TS
  {
    public CommonObjects.TS_Result TabuSearch(List<int> parameterList, CommonObjects.Interface.IDataTS Data,
int numberOfVehicles)
    {
      if (Data == null) throw new ArgumentNullException("Data not initialized!");

      List<List<int>> currentParameters = CreateVRP(parameterList, numberOfVehicles);
      List<CommonObjects.BestSolution> solutionList = new List<CommonObjects.BestSolution>();
      Random rand = new Random((int)DateTime.Now.Ticks);

      double solution = Data.ComputeSolution(currentParameters);
      CommonObjects.BestSolution bestSolution = new CommonObjects.BestSolution(solution,
GetCurrentCityOrder(currentParameters), 0);
      solutionList.Add(new CommonObjects.BestSolution(bestSolution.solution, bestSolution.solutionList,
bestSolution.iteration));
      Stopwatch sw = new Stopwatch();

      int solutionCounter = 0;
      int iterationCounter = 1;

      sw.Start();
      while(true)
      {
        currentParameters = new List<List<int>>(Data.ComputeNextNeighbour(currentParameters));
        solution = Data.ComputeSolution(currentParameters);
        if (solution < bestSolution.solution)
        {
          solutionCounter = 0;
          bestSolution = new CommonObjects.BestSolution(solution, GetCurrentCityOrder(currentParameters),
iterationCounter);
          solutionList.Add(new CommonObjects.BestSolution(bestSolution.solution, bestSolution.solutionList,
bestSolution.iteration));
        }

        solutionCounter++;

        if (solutionCounter % 100000 == 0)
        {
```

```
            List<int> randomParameterList = new List<int>(parameterList);
            int index = 0;

            for (int i = 0; i < 10; i++)
            {
              index = rand.Next(randomParameterList.Count - 2);
              randomParameterList.Reverse(index, randomParameterList.Count - index);
            }

            currentParameters = CreateVRP(randomParameterList, numberOfVehicles);
          }

          if (solutionCounter > 1000000) break;

          iterationCounter++;
        }
        sw.Stop();

        return new CommonObjects.TS_Result(solutionList, sw.ElapsedMilliseconds);
      }

      private List<List<int>> CreateVRP(List<int> currentCityOrder, int numberOfVehicles)
      {
        List<List<int>> vrp = new List<List<int>>(numberOfVehicles);
        int numberOfCities = (int)Math.Ceiling((double)currentCityOrder.Count / (double)numberOfVehicles);
        for (int i = 0; i < numberOfVehicles; i++)
        {
          int startIndex = i * numberOfCities;
          int endIndex = (startIndex + numberOfCities) < currentCityOrder.Count ? startIndex + numberOfCities :
currentCityOrder.Count;
          List<int> path = new List<int>(endIndex - startIndex);

          for (int j = startIndex; j < endIndex; j++)
          {
            path.Add(currentCityOrder[j]);
          }

          vrp.Add(path);
        }

        return vrp;
      }

      private List<int> GetCurrentCityOrder(List<List<int>> vrp)
      {
        //Just iterate throug all the vrp vectors in order and
        //put them all in one vector. This creates one vector
        //representing all routes partitioned by -1
        List<int> currentCityOrder = new List<int>();
        foreach (List<int> list in vrp)
        {
          foreach (int item in list)
          {
            currentCityOrder.Add(item);
          }

          currentCityOrder.Add(-1);
        }

        currentCityOrder.RemoveAt(currentCityOrder.Count -1);
        return currentCityOrder;
      }
    }

public class DataTS : CommonObjects.Interface.IDataTS
    {
```

```
    public double[,] Distances;
    Queue<List<List<int>>> TabuList;
    Random rand = new Random((int)DateTime.Now.Ticks);
    int CAPACITY = 10;

    public DataTS(double[,] nDistances, ref Queue<List<List<int>>> nTabuList)
    {
      Distances = nDistances;
      CAPACITY = Convert.ToInt32(Math.Pow(Distances.GetLength(1), 2)) * 2;
      TabuList = nTabuList;
    }

    public double ComputeSolution(List<List<int>> vrp)
    {
      double result = 0;
      double totalPath = 0;
      foreach (List<int> path in vrp)
      {
        totalPath = GetTotalPath(path);
        if (result < totalPath) { result = totalPath; }
      }

      return result;
    }

    public List<List<int>> ComputeNextNeighbour(List<List<int>> vrp)
    {
      List<List<int>> bestCityOrder = new List<List<int>>(vrp);
      List<List<int>> tempCityOrder = new List<List<int>>(vrp);
      double bestSolution = ComputeSolution(bestCityOrder);
      double tempSolution = 0;

      if (rand.NextDouble() > 0.999)
      {
        for (int i = 0; i < tempCityOrder.Count - 1; i++)
        {
          int indexA = rand.Next(tempCityOrder[i].Count);
          int indexB = rand.Next(tempCityOrder[i + 1].Count);
          int temp = tempCityOrder[i][indexA];
          tempCityOrder[i][indexA] = tempCityOrder[i + 1][indexB];
          tempCityOrder[i + 1][indexB] = temp;
        }
      }

      if (rand.NextDouble() > 0.999)
      {
        int routeToAddTo = rand.Next(tempCityOrder.Count);

        if (tempCityOrder[(routeToAddTo + 1) % tempCityOrder.Count].Count > 3)
        {
          int cityToAdd = rand.Next(tempCityOrder[(routeToAddTo + 1) % tempCityOrder.Count].Count);
          tempCityOrder[routeToAddTo].Add(tempCityOrder[(routeToAddTo + 1) %
tempCityOrder.Count][cityToAdd]);
          tempCityOrder[(routeToAddTo + 1) % tempCityOrder.Count].RemoveAt(cityToAdd);
        }
      }

      for (int i = 0; i < tempCityOrder.Count; i++)
      {
        for (int counter = 0; counter < Math.Pow(tempCityOrder.Count,2); counter++)
        {
          List<int> tempCityRoute = new List<int>(tempCityOrder[i]);
          int indexA = rand.Next(tempCityRoute.Count);
          int indexB = rand.Next(tempCityRoute.Count);
          if (indexA > indexB)
          {
```

```
            int temp = indexA;
            indexA = indexB;
            indexB = temp;
          }
          int k = indexA;
          if (vrp[i].Count > 0 && tempCityRoute.Count > 0)
          {
            for (int j = indexB; j >= indexA; j--)
            {
              tempCityRoute[j] = tempCityOrder[i][k];
              k++;
            }
          }

          tempCityOrder[i] = tempCityRoute;
          bool isTabu = true;
          bool tabuListContains = false;
          foreach (List<int> list in tempCityOrder)
          {
            lock (TabuList)
            {
              tabuListContains = TabuListContains(list);
            }
            if (!tabuListContains)
            {
              isTabu = false;
            }
          }

          if (!isTabu)
          {
            tempSolution = ComputeSolution(tempCityOrder);

            if (tempSolution < bestSolution)
            {
              bestSolution = tempSolution;
              bestCityOrder = new List<List<int>>(tempCityOrder);
            }

            lock (TabuList)
            {
              if (TabuList.Count == CAPACITY)
                TabuList.Dequeue();
              TabuList.Enqueue(new List<List<int>>(tempCityOrder));
            }
          }
        }
      }

      return bestCityOrder;
    }

    private double GetTotalPath(List<int> currentCityOrder)
    {
      int nNumberOfCities = currentCityOrder.Count;
      double nTotalPathLength = 0;

      if (nNumberOfCities > 0)
      {
        nTotalPathLength = Distances[currentCityOrder[nNumberOfCities - 1], currentCityOrder[0]];
        for (int i = 0; i < nNumberOfCities - 1; i++)
        {
          nTotalPathLength += Distances[currentCityOrder[i], currentCityOrder[i + 1]];
        }
      }
```

```
        return nTotalPathLength;
    }

    private bool TabuListContains(List<int> vrpList)
    {
        int size = vrpList.Count;
        if (size <= 0)
            throw new ArgumentNullException(); //There must be a list to compare to the Tabu list

        int index = 0;
        bool contains = false;
        //Check all vrp lists in the tabu list
        foreach (List<List<int>> tabuVrp in TabuList)
        {
            if (tabuVrp != null)
            {
                foreach (List<int> tabuList in tabuVrp)
                {
                    if (vrpList.Count != tabuList.Count) return false;

                    contains = true;
                    index = tabuList.IndexOf(vrpList[0]);

                    if (index == -1) return false; //This is not the same route

                    for (int i = 0; i < size; i++)
                    {
                        if (vrpList[i] != tabuList[(i + index) % size])
                        {
                            contains = false;
                            continue;
                        }
                    }

                    if (contains)
                        return true;
                }
            }
        }

        return false;
    }
}
```

D.2 Multi core implementation

```
static void Main(string[] args)
    {
        int SIZE = 50;
        string InputPath = @"C \TSP_25.csv";
        string ReportPath = @"C: \tempTSP_TabuSC_solution_10.txt";
        Stopwatch sw = new Stopwatch();

        double[,] DistanceMatrix = CommonObjects.CommunicationManager.ReadTSP(InputPath);

        List<int> cities = new List<int>(DistanceMatrix.GetLength(1));
        for (int i = 0; i < cities.Capacity; i++)
        {
            cities.Add(i);
        }

        Queue<List<List<int>>> TabuList = new
Queue<List<List<int>>>(Convert.ToInt32(Math.Pow(DistanceMatrix.GetLength(1), 2)) * 10000);
        List<CommonObjects.BestSolution> solutionList = new List<CommonObjects.BestSolution>();
```

```
        Parallel.For(0, SIZE, i =>
        {
            TabuSearch.TS ts = new TabuSearch.TS();
            TabuSearch.DataTS data = new TabuSearch.DataTS(DistanceMatrix, ref TabuList);
            solutionList = ts.TabuSearch(cities, data, 2).SolutionList;
        });

        CommonObjects.BestSolution theBestSolution = null;
        foreach (CommonObjects.BestSolution solution in solutionList)
        {
            if (theBestSolution == null || theBestSolution.solution > solution.solution)
                theBestSolution = solution;

            Console.WriteLine(solution.ToString());
            Console.WriteLine();
        }

        Console.WriteLine("The best solution!!");
        Console.WriteLine(theBestSolution.ToString());
        Console.WriteLine();
    }
```

D.3 Single core implementation

```
static void Main(string[] args)
    {
        int SIZE = 10;
        string InputPath = @"C:\TSP_25.csv";
        string ReportPath = @"C:\tempTSP_TabuSC_solution_10.txt";
        Stopwatch sw = new Stopwatch();

        double[,] DistanceMatrix = CommonObjects.CommunicationManager.ReadTSP(InputPath);

        List<int> cities = new List<int>(DistanceMatrix.GetLength(1));
        for (int i = 0; i < cities.Capacity; i++)
        {
            cities.Add(i);
        }

        List<CommonObjects.BestSolution> solutionList = new List<CommonObjects.BestSolution>();
        for (int i = 0; i < SIZE; i++)
        {
            TabuSearch.TS ts = new TabuSearch.TS();
            Queue<List<List<int>>> TabuList = new
Queue<List<List<int>>>(Convert.ToInt32(Math.Pow(DistanceMatrix.GetLength(1), 2)) * 2);
            TabuSearch.DataTS data = new TabuSearch.DataTS(DistanceMatrix, ref TabuList);
            solutionList = ts.TabuSearch(cities, data, 2).SolutionList;
        }

        CommonObjects.BestSolution theBestSolution = null;
        foreach (CommonObjects.BestSolution solution in solutionList)
        {
            if (theBestSolution == null || theBestSolution.solution > solution.solution)
                theBestSolution = solution;

            Console.WriteLine(solution.ToString());
            Console.WriteLine();
        }

        Console.WriteLine("The best solution!!");
        Console.WriteLine(theBestSolution.ToString());
        Console.WriteLine();
    }
```

Appendix E

E.1 Common objects

E.1.1 Interfaces

```
public interface IData
  {
    double ComputeSolution(List<double> argument);

    List<double> ComputeNextNeighbour(List<double> currentSolution);

    double ComputeTemperature(double temperature, double alpha, int iteration);
  }


public interface IDataAC
  {
    decimal ComputeSolution(decimal[] argument);

    decimal[] ComputeNextNeighbour(decimal[] currentSolution);
  }


public interface IDataTS
  {
    double ComputeSolution(List<List<int>> currentCityOrder);

    List<List<int>> ComputeNextNeighbour(List<List<int>> currentCityOrder);
  }
```

E.1.2 Objects

```
public class Ackley : Interface.IData
  {
    private const int n = 2;
    private const double a = 20;
    private const double b = 0.2;
    private const double c = 2 * Math.PI;
    public const double SEARCHSPACE = 32768;
    public const double STEPSIZE = 200;
    Random rand = new Random(DateTime.Now.Millisecond);

    public double ComputeSolution(List<double> argument)
    {
      double sum1 = 0;
      for (int i = 0; i < n; i++)
      {
        double temp = argument[i];
        sum1 += Math.Pow(argument[i], 2);
      }

      double sum2 = 0;
      for (int i = 0; i < n; i++)
      {
        double temp = argument[i];
        sum2 += Math.Cos(c * argument[i]);
      }

      return ((-a) * Math.Exp((-b) * Math.Sqrt(sum1 / n))) - Math.Exp(sum2 / n) + a + Math.E;
    }

    public List<double> ComputeNextNeighbour(List<double> currentParameters)
```

```
    {
      for (int i = 0; i < currentParameters.Count; i++)
      {
        int des = 1;
        if (rand.NextDouble() > 0.5)
          des = -1;

        double temp = STEPSIZE * des * rand.NextDouble();

        currentParameters[i] += temp;

        if (currentParameters[i] > SEARCHSPACE) { currentParameters[i] = SEARCHSPACE; }
        if (currentParameters[i] < -SEARCHSPACE) { currentParameters[i] = -SEARCHSPACE; }
      }

      return currentParameters;
    }

    public double ComputeTemperature(double temperature, double alpha, int iteration)
    {
      return temperature *= alpha;
    }
  }


public class AntTour
  {
    public AntTour(int nAnt, double nLength)
    {
      Ant = nAnt;
      Length = nLength;
      VisitCityOrder = new List<int>(1000);
    }

    public AntTour(int nAnt, double nLength, List<int> nVisitCityOrder)
    {
      Ant = nAnt;
      Length = nLength;
      VisitCityOrder = nVisitCityOrder;
    }

    public int Ant { get; set; }
    public int Iteration { get; set; }
    public double Length { get; set; }
    public List<int> VisitCityOrder { get; set; }
  }


public class BestSolution
  {
    public double solution;
    public List<int> solutionList;
    public int iteration;

    public BestSolution(double nSolution, List<int> nSolutionList, int nIteration)
    {
      solution = nSolution;
      solutionList = new List<int>(nSolutionList);
      iteration = nIteration;
    }

    public override string ToString()
    {
      StringBuilder builder = new StringBuilder();
      foreach (int item in solutionList)
```

```
        {
            builder.AppendFormat("{0} ", item);
        }
        return string.Format("Solution = {0}\nIteration number {1}\n{2}", solution, iteration, builder.ToString());
    }
  }

public class Result
  {
    public Result(double bestResult, List<double> bestParameters, List<List<double>> parameterlist, TimeSpan
elapsedTime)
    {
        BestResult = bestResult;
        BestParameters = bestParameters;
        ParameterList = parameterlist;
        ElapsedTime = elapsedTime;
    }

    public double BestResult { get; set; }

    public List<double> BestParameters { get; set; }

    public List<List<double>> ParameterList { get; set; }

    public TimeSpan ElapsedTime { get; set; }
  }

public class TS_Result
  {
    public TS_Result(List<BestSolution> solutionlist, long timeElapsedMilliSecond)
    {
        SolutionList = solutionlist;
        TimeElepsedMillisecond = timeElapsedMilliSecond;
    }

    public BestSolution Bestsolution
    {
        get
        {
            BestSolution theBestSolution = null;
            foreach (BestSolution solution in SolutionList)
            {
                if (theBestSolution == null || theBestSolution.solution > solution.solution)
                    theBestSolution = solution;
            }

            return theBestSolution;
        }
    }
    public List<BestSolution> SolutionList { get; set; }
    public long TimeElepsedMillisecond { get; set; }
  }

public class CommunicationManager
  {
    public static bool StartProgram(string sMessage, out int nSize, out bool bReport, out string sReportPath, out bool
bFullReport, out bool bParamPath, out string sParamPath)
    {
        nSize = 0;
        bReport = false;
        sReportPath = string.Empty;
        bParamPath = false;
        sParamPath = string.Empty;
        bFullReport = false;

        Console.WriteLine(string.Format("******** {0} ********", sMessage));
```

```
        Console.WriteLine();

        Console.WriteLine("-Set the number of iterations by using the command SIZE=<some arbritrary number>");
        Console.WriteLine("-To write parameterpath use the command PARAMPATH=<the folder where to write the
parameter paths>");
        Console.WriteLine("-To write a report use the command REPORT=<the folder where to write the report>");
        Console.WriteLine("-To write a full report use the command FULLREPORT instead of the REPORT command");
        string commandString = Console.ReadLine();

        try
        {
           string[] commands = commandString.Split(' ');
           foreach (string command in commands)
           {
              string[] commandParts = command.Split('=');

              switch (commandParts[0].Trim().ToUpper())
              {
                 case "SIZE":
                    nSize = Convert.ToInt32(commandParts[1].Trim());
                    break;
                 case "PARAMPATH":
                    bParamPath = true;
                    sParamPath = commandParts[1].Trim();
                    break;
                 case "REPORT":
                    bReport = true;
                    sReportPath = commandParts[1].Trim();
                    break;
                 case "FULLREPORT":
                    bReport = true;
                    bFullReport = true;
                    sReportPath = commandParts[1].Trim();
                    break;
                 default:
                    break;
              }
           }
        }
        catch (Exception ex)
        {
           return false;
        }

        Console.WriteLine();
        Console.WriteLine(string.Format("Number of iterations: {0}", nSize));
        Console.WriteLine();

        return true;
     }

     public static void WriteReport(bool bFullreport, string sReportPath, List<CommonObjects.Result> resultList,
Stopwatch sw)
     {
        StreamWriter writer = new StreamWriter(sReportPath);
        writer.WriteLine(string.Format("Total time to compute: {0}ms", sw.ElapsedMilliseconds));
        CommonObjects.Ackley data = new CommonObjects.Ackley();
        writer.WriteLine(string.Format("Best possible solution: {0}", data.ComputeSolution(new List<double>(new
double[] { 0, 0 }))));
        writer.WriteLine();
        writer.WriteLine("_____");
        writer.WriteLine();

        CommonObjects.Result bestResult = null;
        foreach (var item in resultList)
        {
```

```
            if (bestResult == null || bestResult.BestResult > item.BestResult)
                bestResult = item;
        }

        writer.WriteLine(string.Format("Best solution found: {0}", bestResult.BestResult));
        writer.WriteLine(string.Format("Best params X:{0} Y:{1}", bestResult.BestParameters[0],
bestResult.BestParameters[1]));
        writer.WriteLine(string.Format("Time: {0}ms", bestResult.ElapsedTime.Milliseconds));
        writer.WriteLine();
        writer.WriteLine("_____");
        writer.WriteLine();

        if (bFullreport)
        {
            writer.WriteLine("Best_solution X_param Y_param Time");
            foreach (var item in resultList)
            {
                writer.WriteLine(string.Format("{0} {1} {2} {3}", item.BestResult, item.BestParameters[0],
item.BestParameters[1], item.ElapsedTime.Milliseconds));
            }
        }
        writer.Flush();
        writer.Close(); writer.Dispose();
    }

    public static bool StartTSPProgram(string sMessage, out int nNumberOfIterations, out int nAntsPerCity, out
double nAlpha, out double nBeta, out double nRate, out double nInitialPheromon, out double nConstant, out string
sInputPath, out string sReportPath)
    {
        nNumberOfIterations = 0;
        nAntsPerCity = 0;
        nAlpha = 0;
        nBeta = 0;
        nRate = 0;
        nInitialPheromon = 0;
        nConstant = 0;
        sReportPath = string.Empty;
        sInputPath = string.Empty;

        Console.WriteLine(string.Format("******** {0} ********", sMessage));
        Console.WriteLine();

        Console.WriteLine("-Set the number of iterations by using the command SIZE=<some arbritrary number>");
        Console.WriteLine("-Set the number of ants per city by using the command ANTS=<some arbritrary number>");
        Console.WriteLine("-Set the value of alpha by using the command ALPHA=<some arbritrary number>");
        Console.WriteLine("-Set the value of beta by using the command BETA=<some arbritrary number>");
        Console.WriteLine("-Set the value of rate of evaporation by using the command RATE=<some arbritrary
number>");
        Console.WriteLine("-Set the value of initial pheromon by using the command INIT=<some arbritrary
number>");
        Console.WriteLine("-Set the value of pheromon constant by using the command CONSTANT=<some arbritrary
number>");
        Console.WriteLine("-Use the command INPUTPATH=<the path to the input file>");
        Console.WriteLine("-Use the command REPORT=<the folder where to write the report>");
        string commandString = Console.ReadLine();

        try
        {
            string[] commands = commandString.Split(' ');
            foreach (string command in commands)
            {
                string[] commandParts = command.Split('=');

                switch (commandParts[0].Trim().ToUpper())
                {
                    case "SIZE":
```

```
                nNumberOfIterations = Convert.ToInt32(commandParts[1].Trim());
                break;
            case "INPUTPATH":
                sInputPath = commandParts[1].Trim();
                break;
            case "REPORT":
                sReportPath = commandParts[1].Trim();
                break;
            case "ANTS":
                nAntsPerCity = Convert.ToInt32(commandParts[1].Trim());
                break;
            case "ALPHA":
                nAlpha = Convert.ToDouble(commandParts[1].Trim());
                break;
            case "BETA":
                nBeta = Convert.ToDouble(commandParts[1].Trim());
                break;
            case "RATE":
                nRate = Convert.ToDouble(commandParts[1].Trim());
                break;
            case "INIT":
                nInitialPheromon = Convert.ToDouble(commandParts[1].Trim());
                break;
            case "CONSTANT":
                nConstant = Convert.ToDouble(commandParts[1].Trim());
                break;
            default:
                break;
        }
      }
    }
    catch (Exception ex)
    {
      return false;
    }

    Console.WriteLine();
    Console.WriteLine(string.Format("Number of iterations: {0}", nNumberOfIterations));
    Console.WriteLine();

    return true;
}

public static double[,] ReadTSP(string sUrl)
{
    StreamReader reader = new StreamReader(sUrl);

    List<string> lines = new List<string>(1000);
    while (!reader.EndOfStream)
    {
      lines.Add(reader.ReadLine());
    }

    double[,] matrix = new double[lines.Count, lines.Count];
    string[] tempItems; int lineIndex = 0; string tempLine;
    foreach (string line in lines)
    {
      tempLine = line.Replace(".", ",");
      tempItems = tempLine.Split(new string[] { ";" }, StringSplitOptions.RemoveEmptyEntries);
      for (int i = 0; i < lines.Count; i++)
      {
        matrix[lineIndex, i] = Convert.ToDouble(tempItems[i]);
      }

      lineIndex++;
    }
```

```
    return matrix;
}

public static void WriteTSPSolution(string sPath, AntTour[] antTours, Stopwatch sw)
{
    StreamWriter writer = new StreamWriter(sPath);
    int index = 0;
    double bestLength = antTours[index].Length;
    for (int i = 1; i < antTours.Length; i++)
    {
        if (antTours[i].Length < bestLength)
        {
            bestLength = antTours[i].Length;
            index = i;
        }
    }

    writer.WriteLine(string.Format("Total time to compute: {0}ms", sw.ElapsedMilliseconds));
    writer.WriteLine(string.Format("Best possible solution: {0}", bestLength));
    writer.WriteLine(string.Format("That was ant number: {0}", antTours[index].Ant));
    writer.WriteLine(string.Format("That was iteration number: {0}", antTours[index].Iteration));
    writer.WriteLine("_____");
    foreach (int city in antTours[index].VisitCityOrder)
    {
        writer.WriteLine(city);
    }

    writer.Flush();
    writer.Close(); writer.Dispose();
}

public static void WriteTSPSolution(string sPath, AntTour[] antTours, double[,] Pheromons, Stopwatch sw)
{
    StreamWriter writer = new StreamWriter(sPath);
    int index = 0;
    double bestLength = antTours[index].Length;
    for (int i = 1; i < antTours.Length; i++)
    {
        if (antTours[i].Length < bestLength)
        {
            bestLength = antTours[i].Length;
            index = i;
        }
    }

    writer.WriteLine(string.Format("Total time to compute: {0}ms", sw.ElapsedMilliseconds));
    writer.WriteLine(string.Format("Best possible solution: {0}", bestLength));
    writer.WriteLine(string.Format("That was ant number: {0}", antTours[index].Ant));
    writer.WriteLine(string.Format("That was iteration number: {0}", antTours[index].Iteration));
    writer.WriteLine("_____");
    foreach (int city in antTours[index].VisitCityOrder)
    {
        writer.WriteLine(city);
    }
    writer.WriteLine("_____");
    writer.WriteLine("Pheromons");
    for (int i = 0; i < Pheromons.GetLength(1); i++)
    {
        for (int j = 0; j < Pheromons.GetLength(1); j++)
        {
            writer.Write(string.Format("{0}\t", Pheromons[i, j]));
        }
        writer.WriteLine();
    }
```

```
        writer.Flush();
        writer.Close(); writer.Dispose();
    }

    public static void WriteVRPSolution(string sPath, TS_Result result)
    {
        StringBuilder builder = new StringBuilder();
        foreach (int item in result.Bestsolution.solutionList)
        {
            builder.AppendFormat("{0} ", item);
        }

        StreamWriter writer = new StreamWriter(sPath);

        writer.WriteLine(string.Format("Total time to compute: {0}ms", result.TimeElepsedMillisecond));
        writer.WriteLine(string.Format("Solution = {0}", result.Bestsolution.solution));
        writer.WriteLine(string.Format("Iteration number {0}", result.Bestsolution.iteration));
        writer.WriteLine("_____");
        writer.WriteLine();
        writer.WriteLine();
        writer.WriteLine(builder.ToString());
        writer.Flush();
        writer.Close(); writer.Dispose();
    }
}
```


HÁSKÓLINN Í REYKJAVÍK
REYKJAVIK UNIVERSITY