EE451 Project Report
Accelerating Artificial Bee Colony (ABC) Optimization Algorithm Using a Multi-Swarm Approach on CPU

Tingqi (Ting) Wang and Yang Shen

Introduction

The Artificial Bee Colony Algorithm (ABC) is a meta-heuristic swarm-based numerical optimization algorithm introduced by Dervis Karaboga in 2005 inspired by the intelligent foraging behavior of honey bee swarms. The ABC algorithm is highly flexible, robust, versatile, and simple to implement, thus allowing it to have wide applications in many different fields: ranging from design optimization in engineering to minimize material usage and cost to achieve desired performance, to optimization of investment portfolio by balancing risk and reward, to clustering in image processing to segment images and compress color spaces.

The ABC algorithm implements a population-based optimization method, the food source locations (i.e. solutions of function to be optimized) are modified by artificial bees for the purpose of finding the most fit food source which means with the highest amount of nectar (i.e highest or lowest fitness value depending maximize or minimize problem).

The ABC algorithm is a serial algorithm that does not perform well as the solution space and the dimension of the problem is increased due to the significantly high numbers of iterations. Therefore, the parallelization of the algorithm will enable the algorithm to have improved performance runtime in large-scale optimization problems. In addition, the performance of the ABC algorithm may be restricted since only one swarm takes the whole responsibility of searching for the best food source among all the surrounding locations. Thus could lead to no update for the best food source location in several iterations, harming performance. As a result, contrary to existing parallel implementation of the ABC algorithm, we decide to adopt a multi-swarm approach when parallelizing the algorithm to enable multiple swarms to search the solution space in order to allow better exploration of the solution space as well as increase the algorithm's scalability and decrease the runtime.

In our project, we aim to achieve the following:

- Implement a parallel version of the ABC algorithm using a multi-swarm approach in C using Pthreads to run on the CPU

- Test the performance of the CPU-parallel implementation of the ABC algorithm on four benchmark functions to test the robustness and effectiveness in finding the global minimum under different optimization landscape scenarios.
- Evaluate the runtime and solution results of running the original serial implementation in comparison to our parallel implementation of the algorithm on four benchmark functions in a CPU context.
- Identity the benefits and limitations of using a multi-swarm approach to parallelize the algorithm.
- Analyze the impact on runtime and solution quality (fitness value) of key parameters of the algorithm.
-

In the ABC algorithm, the honey bees/agents can be separated into three groups: employed bees, onlooker bees, and scout bees. The tasks assigned to them are quite different but consecutive.

Algorithm

The ABC algorithm consists of three main stages:

Employed Bee Stage:
Conceptually, employed bees are responsible for searching the food around food sources and share the corresponding location and nectar amount information to the onlooker bees.

In the program, the employed bee produce new solutions (food source positions) $\upsilon_{i,j}$ in the neighborhood of $x_{i,j}$ (entire solution space) for the employed bees using the formula

$$\upsilon_{i,j} = x_{i,j} + \Phi_{ij}(x_{i,j} - x_{k,j})$$

Where k is a solution in the neighborhood of i, Φ is a random number in the range [-1,1] and j is bounded by the number of dimensions of the solution. Then the bee applies a greedy selection between x_i and v_i.

Onlooker Bee Stage:
Conceptually, the onlooker bees select good ones from the food sources found by employed bees depending on the nectar amount.

In the program, the fitness value decides the possibility of being chosen by an onlooker bee for the corresponding solution. Equation for calculating the fitness value of the solution (Figure 1), but note the given function is used for minimization and should be changed when using other functions:

$$fit_m(\vec{x_m}) = \begin{cases} \dfrac{1}{1 + f_m(\vec{x_m})} & \text{if } f_m(\vec{x_m}) \geq 0 \\ 1 + abs(f_m(\vec{x_m})) & \text{if } f_m(\vec{x_m}) < 0 \end{cases}$$

Figure 1 (Dervis Karaboga 2010).

The probability is then calculated using the fitness value. Equation for calculating the probability of the solution being chosen (Figure 2):

$$p_m = \frac{fit_m(\vec{x_m})}{\sum_{m=1}^{SN} fit_m(\vec{x_m})} .$$

Figure 2 (Dervis Karaboga 2010).

The onlooker bees then improve the solution chosen based on P_m in a similar manner as the employed bee.

Scout Bee Stage:

Conceptually, for those employed bees whose food sources are not being selected, they transform into scout bees, abandon the food source in their hand and continue searching for the new ones.

In the program, once a solution has not improved for a given max number of iterations, the solution is dropped and a new solution is randomly produced using the following solution:

$$X_{i,j} = min_j + rand(0,1)*(max_j - min_j)$$

Where i is the solution number and j is the dimension of the solution (i.e. the number of parameters the optimization problem has).

The algorithm runs until a termination criteria is satisfied.

The pseudocode for the serial implementation of the single-swarm ABC algorithm are presented below:

Initialisation:
Initialize the initial population and Evaluate fitness;
Calculate the initial cost function value, f(Sol);
Set best solution, Solbest ← Sol;
Set maximum number of iteration, NumOfIte;
Set the population size;
//where population size = OnlookerBee = EmployeedBee;
iteration ← 0;
Improvement:
do while (iteration < NumOfIte)
for i=1: EmployeedBee
Select a random solution and apply random neighborhood structure;
Sort the solutions in ascending order based on the Penalty cost;
Determine the probability for each solution, based on the following formula :

$$p_i = \frac{\sum \left(1 / fit_i \right)^{-1}}{fit_i}$$

end for

for i=1: OnlookerBee
Sol* ← select the solution who has the higher probability;
Sol** ← Apply a random Nbs on Sol*;
if (Sol** < Solbest)
Solbest=Sol**;
end if
end for
Scoutbee determines the abandoned food source and replace it with the new food source.
iteration++

end do

Context

We decided to utilize pthread for implementing the CPU parallelization version using a multi-swarm approach. As a result, there are a few challenges we had to tackle. The first one is that we need to consider how and when to place the communication between swarms during the whole searching process as each swarm works independently. Also, upon this communication, while we are synchronizing the best solution found by each group, there should be an efficient

way to minimize the influence on the algorithm's overall performance as synchronization may introduce processor idling.

Moreover, since the serial version of the ABC algorithm works for only one swarm, for a multi-swarm approach, each swarm should execute the ABC algorithm separately and combine their results when it finishes running, which means that each swarm should have its own input data and variables. This results in us having to consider the structuring of the data matrix for the purpose of eliminating the dependencies between swarms.

Parallelization Strategy

The initial randomized parameters and their corresponding function values, the solution fitness values, and the probability for each function value will be stored in separate matrices. Each employed bee is responsible for one row in each separate matrix (each row is one solution for the optimization problem) and after being determined not chosen, re-randomize one of the function parameters in its owned row, recalculate and update the new function value so that a whole new solution generated and will participate in next compare and choose round.

Instead of putting all the burden on one swarm for finding the best food source, we implement a multi-swarm approach using pthreads so that each swarm group is responsible for only a part of the solution space assigned to them. We partition the input matrix into several sections depending on the number of swarms and eliminate any data dependencies that exist.

Each partition will be assigned to one thread/processor for parallel execution which means that each swarm will only change their function parameters in the partition of the matrix which belongs to it and every swarm will begin searching and replacing asynchronously.

Also, we implement the communications between swarms in a unidirectional ring topology after a given number of iterations has passed so that in every fixed amount of iterations every swarm can get the updated location for the best food source found by other swarms at the global level and replace it with their worst solution.

Hypothesis

The ABC algorithm consists of three main stages: Employed Bee, Onlooker Bee, and Scout Bee. In all three stages, they can run independently, hence concurrently, after partitioning the input matrices and there will not be any dependencies. However, there is still a need for synchronization at certain points for communication which could result in some damage to runtime. But when we implement a multi-swarm method, the same amount of solutions can be

generated and compared in a much shorter time. As a result, we can say that in the same amount of time, the parallel version can explore more solutions compared with the serial version which enables a higher solution quality.

Thus, our hypothesis is:

The parallel implementation of the ABC algorithm that adopts a multi-swarm approach will achieve higher performance in computation speed and achieve better solution quality when using the same execution time as the serial version of the ABC algorithm.

Experimental Setup

- **Datasets**

In the field of optimization algorithms, benchmark functions are standard experimental objects for evaluating the performance. They provide a clear vision of how well the algorithm can perform when solving different problem scenarios.

The four benchmark functions we will use to evaluate the ABC algorithm's performance are:

Sphere function: Can help check whether the ABC algorithm can find the global minimum at the origin where the function value is zero.

Rosenbrock functions: Similar check to the Sphere function but it has a curved valley which provides a more challenging problem space.

Griewank function: Test that the ABC algorithm can escape local minima and continue searching for global minimum.

Rastrigin function: Can test the ABC algorithm's performance in multi-modal optimization problems.

The range for all four tests are [-5.12, 5.12] and the exact solution should be 0 for all parameters/dimensions.

- **Serial Implementation (baseline)**

The serial implementation of the ABC algorithm that we will run as our baseline is adapted from the original C implementation developed by Karaboga. The program is run on a single Epyc-7513 CPU at 2.6 GHz on the Epyc-64 partition of the HPC.

- **Pthread Parallel Implementation**

We used Pthreads to implement a parallelized multi-swarm ABC algorithm to examine the speedup of parallelization. As described in the Parallelization Strategy section, each thread/processor/swarm will be in charge of searching through a section of the total solution space. We will be running the parallelized algorithm on 2, 4, and 16 threads/processors on the Epyc-7513 CPU at 2.6 GHz on the Epyc-64 partition of the HPC.

- **Algorithm Execution**

We design our program to run 30 independent iterations of the ABC algorithm, both serially and parallelly, to then find the average of runtime and fitness value of produced solution. Doing so allows us to obtain reliable and robust output data for our analysis.

Result and Analysis

- **Evaluating the impact of MaxCycle**

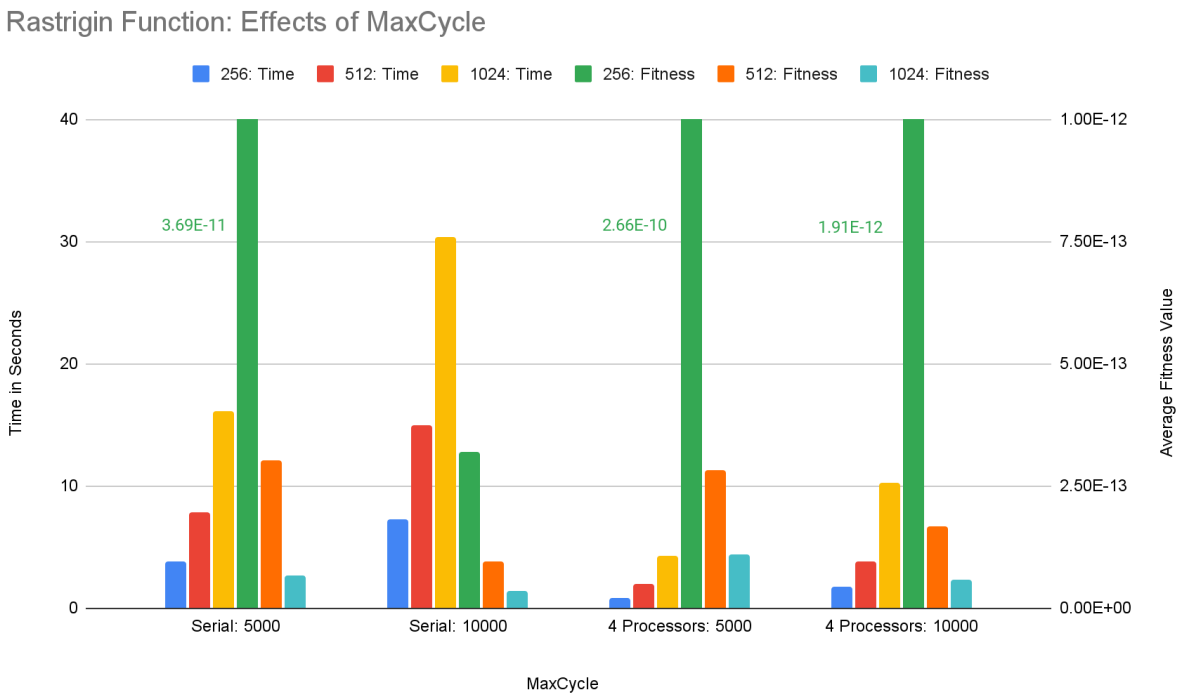

Figure 3. Comparing the runtime and average fitness values of different MaxCycle values

Figure 3. depicts the runtime and average fitness value of optimizing the Rastrigin function using the serial and parallel (4 processors) versions of the ABC algorithm with maxCycles being 5000 and 10000 on all three colony sizes. The maxCycle value dictates how many iterations the algorithm runs for before terminating. Looking at the graph above, we can see that the runtime of both versions is higher for maxCycle being 10000 in comparison to maxCycle being 5000, which demonstrates an expected positive relationship between runtime and maxCycle. We can also see that the solution quality/fitness value is lower (lower is better given we are minimizing the functions) when we have a higher maxCycle for both serial and parallel. There is also a positive relationship between solution quality and maxCycle, which is again expected because with more iterations, the algorithm will be able to further explore the solution space and further improve the final solution.

We can also observe that doubling the maxCycle from 5000 to 10000 has led to an increase in runtime by approximately a factor of 2 for both versions across all colony sizes, but the gain in solution fitness value was significantly greater than a factor of 2, exhibiting trends similar to exponential improvement. As a result, increasing the maxCycle is an important component to achieve high quality solutions for the ABC algorithm, and parallelization enables the algorithm to run on very large maxCycle by reducing the runtime. Thus making the algorithm more applicable to applications requiring high precision solutions.

- **Evaluating the effect of migrationGap**

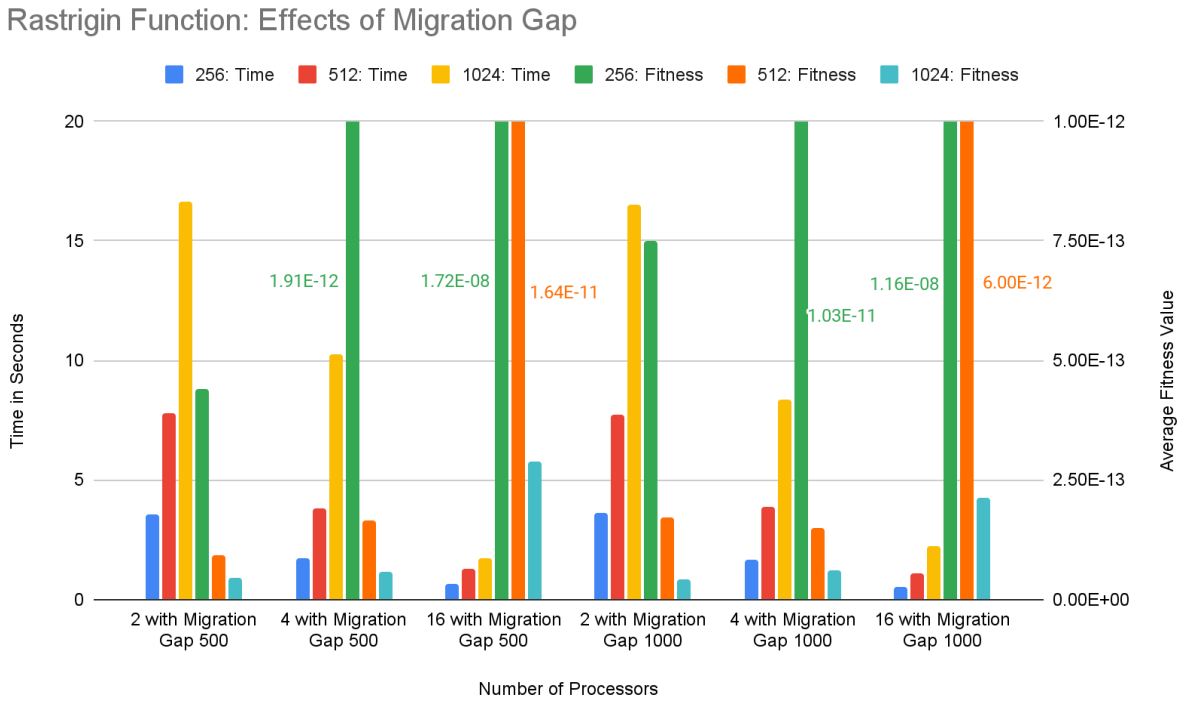

Figure 4. Comparing the runtime and average fitness value of different migrationGap

Figure 4. illustrates the runtime and average fitness value of running the parallel implementation on the Rastrigin function using 2, 4, and 16 processors with migrationGap of 500 and 1000. MigrationGap is the parameter that determines the number of iterations before a communication between all swarms takes place. Looking at the runtime between a migrationGap of 500 to 1000, we can see that the runtime is on the whole higher for a migrationGap of 500. This is due to the fact that a lower migrationGap will mean a higher number of times the program needs to synchronize in order to communicate without corrupting the data. Synchronization and the process of communicating data between swarms results in hindered performance and higher runtime.

On the other hand, looking at the fitness values, we can see that with a higher migrationGap of 1000, the fitness values tend to be lower in comparison to those values with migrationGap of 500. The reason behind this is that with a lower migrationGap and in turn a higher frequency of communication, the swarms can more frequently use the best solutions in other swarms to help improve their search by focusing on potentially promising areas of the solution space and potentially helps to avoid local optima to search for the global optimum.

As a result, the migrationGap is a crucial parameter that needs to be carefully considered when implementing the parallelized ABC algorithm to achieve low runtime and high solution quality.

- **Evaluating algorithm performance in different scenarios**

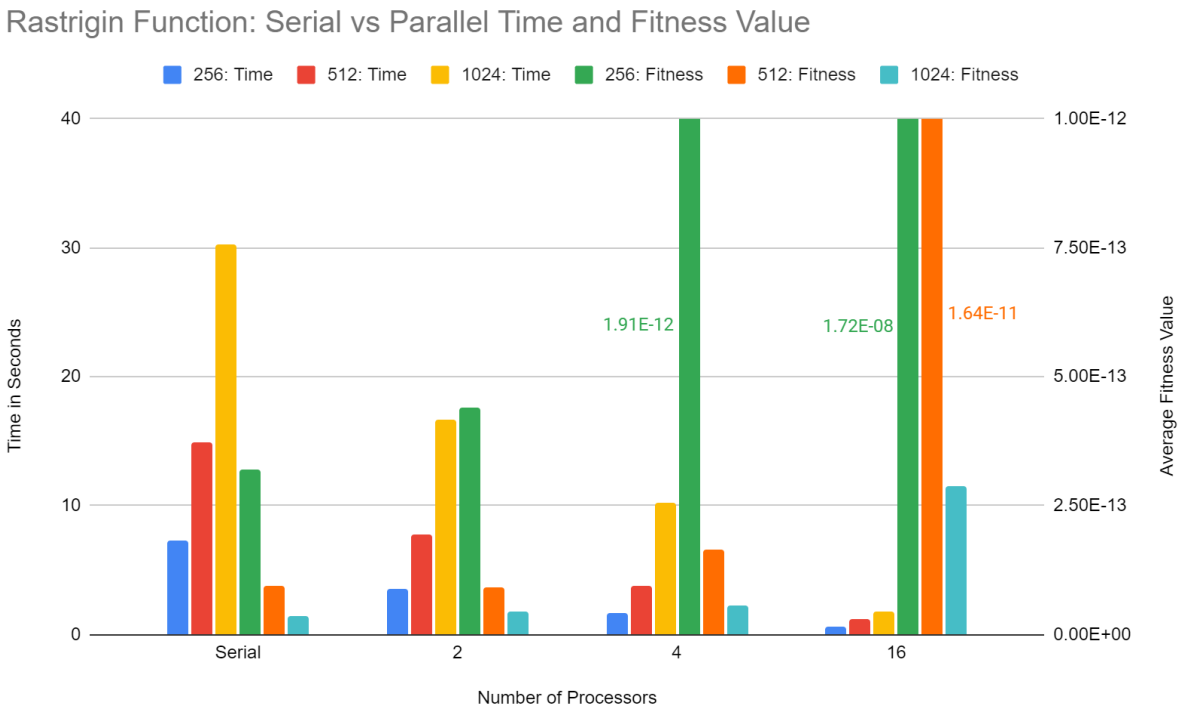

Figure 5. Comparing the runtime and average fitness value of different processor numbers in Rastrigin Function

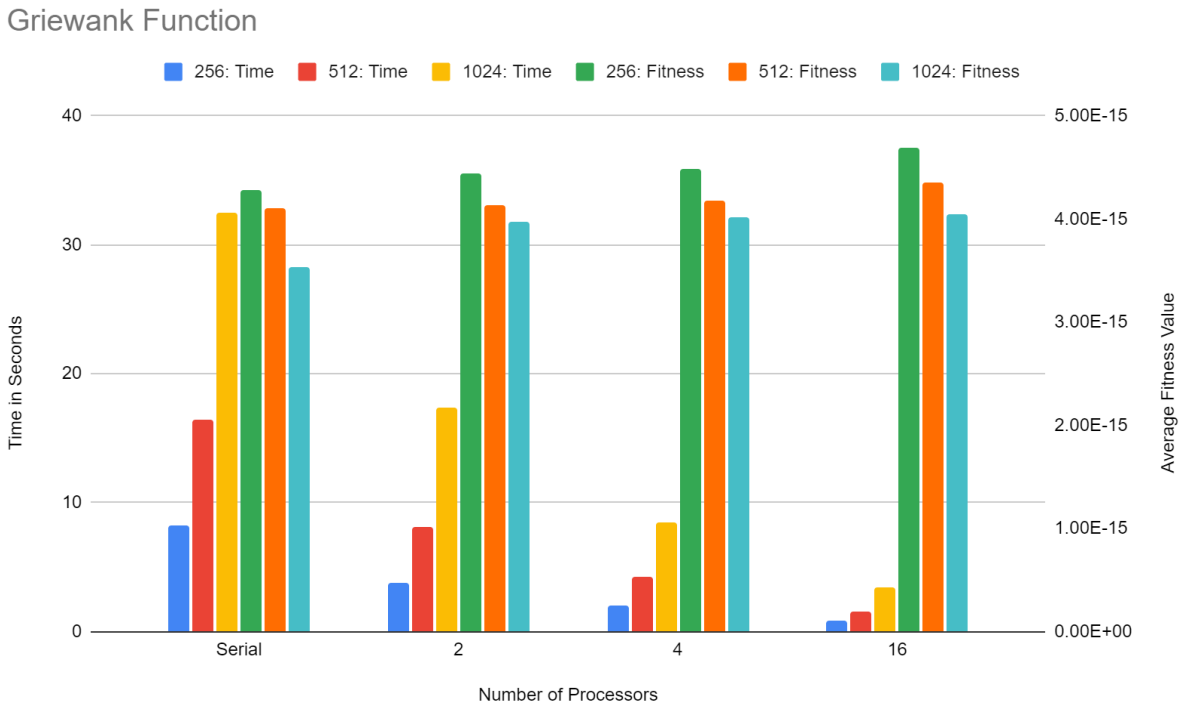

Figure 6. Comparing the runtime and average fitness value of different processor numbers in Griewank Function

As shown above in Figure 5-8, one graph for each function, we compare the runtime and solution fitness by increasing the processor numbers or the colony size in different benchmark functions.

For the runtime, with a growing number of processors, we can observe that the runtime keeps decreasing with the unchanging colony size.Same, with a growing number of colony size, it is clear that the runtime keeps increasing with the unchanging processor number. The main reason is that the workload was splitted by each processor or shared by putting in more bees.

For the solution fitness, as we can see in the fitness columns of four graphs for each benchmark function above (Figure 5-8), With the same amount of colony size, the fitness value is increasing as we increase the number of processors since we do the communication between processors which means each swarm would know the best food source found by other group and replace its own if necessary. This truly helps improve the fitness value. Also, with the same amount of processors, the fitness value is decreasing as we increase the colony size which will lead to the expansion of the food source searching scope. As a result, more solutions will be discovered and compared so the fitness value will decrease.

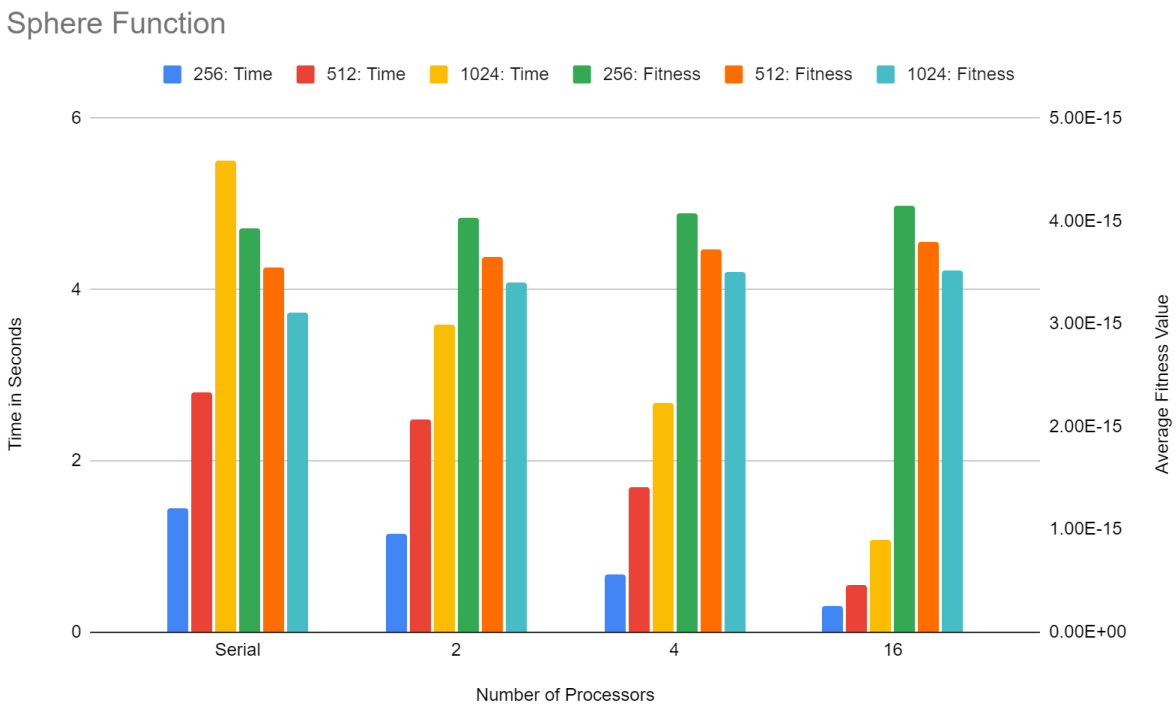

Figure 7. Comparing the runtime and average fitness value of different processor numbers in Sphere Function

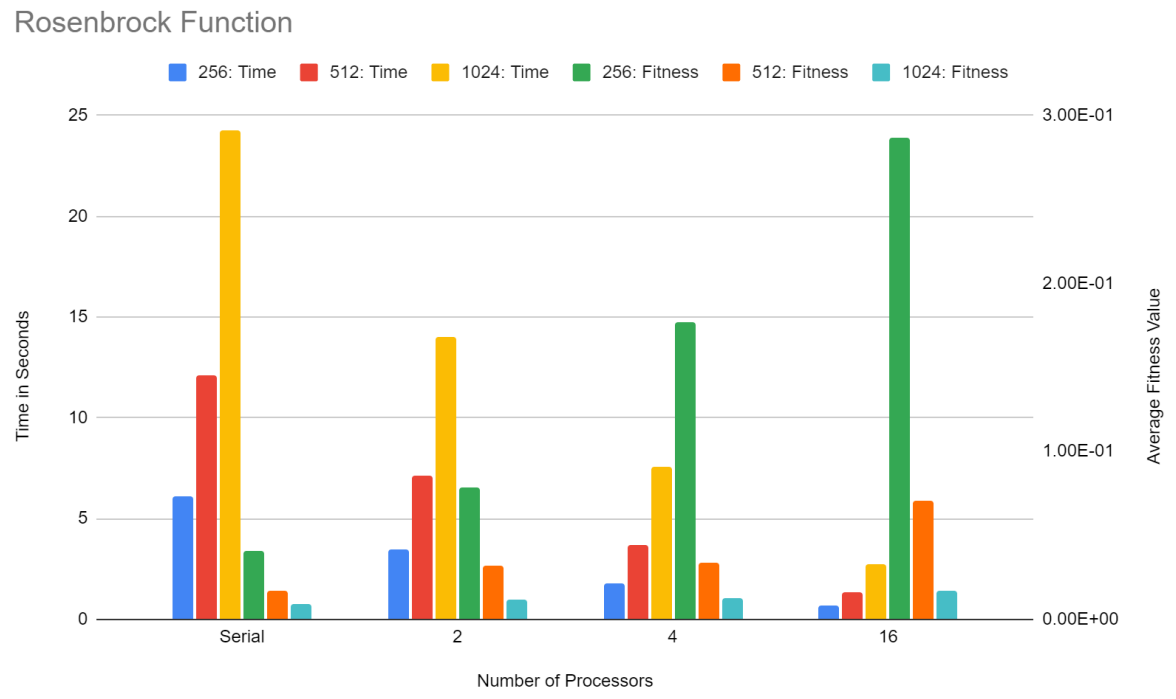

Figure 8. Comparing the runtime and average fitness value of different processor numbers in Rosenbrock Function

- **Evaluating speedup achieved by parallel implementation**

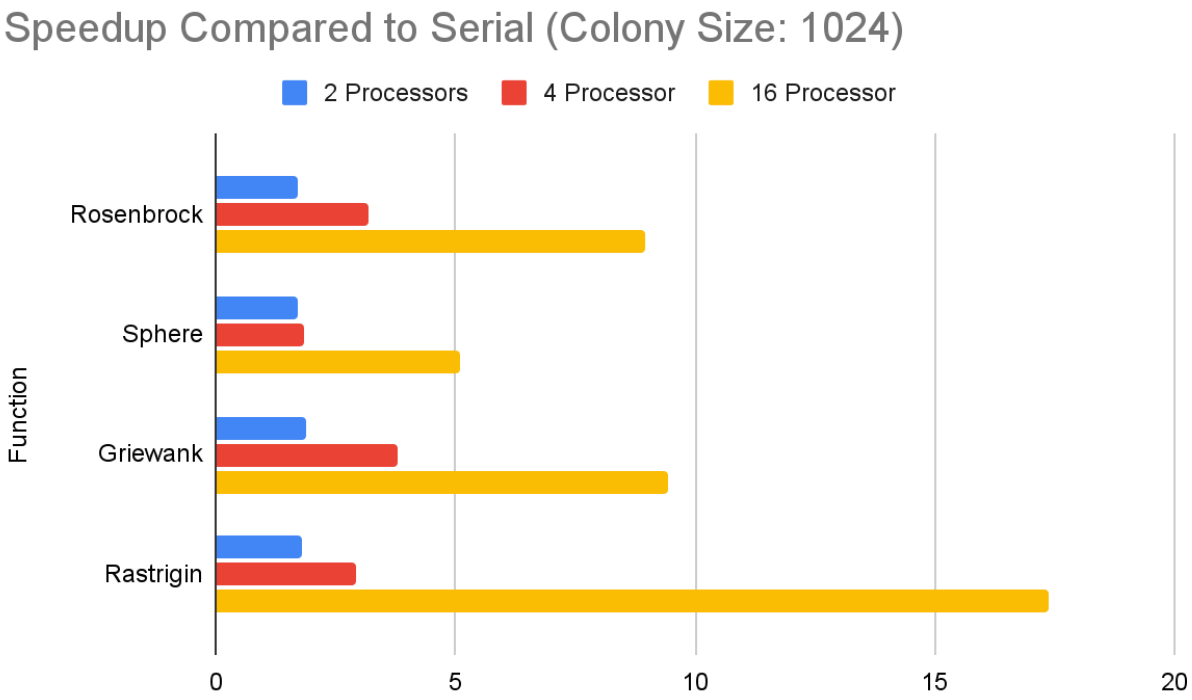

Figure 9. Comparing the speedup of parallel implementation for each benchmark function

The speedup is obtained by:

$$Speedup = \frac{Serial\ ABC\ runtime}{Parallelized\ ABC\ runtime}$$

Figure 9. demonstrates the speedup obtained by the parallelized algorithm when using 2, 4, and 16 processors on the four benchmark functions with migrationGap being 500 and maxCyles being 10000. Although we see differences in the extent of increase in speedup between different functions due to the different solution space each function exhibits, we can see that as the number of processors increases, the speedup increases for all four functions.

However, we can see that the speedup for the Rastrigin function is greater than 16x which is greater than a 16 times increase in the number of processors used. The likely reason behind the superlinear speedup obtained is the more efficient usage of cache in each processor. This is because as we split up the problem amongst multiple processors (in this case 16 processors), each processor deals with smaller datasets that can better fit into cache, which will result in significant speed up of computation and overall performance.

On the whole, the parallelized ABC algorithm we have implemented does indeed decrease the runtime quite proportionally to the number of processors used. It is important to note that the reason it is not exactly proportional is described by Amdahl's law where the serial initialization in the program will bound the overall speedup.

Conclusion

In this project, we implement a multi-swarm ABC algorithm using pthread in optimization problems and test the performance of this algorithm on the CPU via four benchmark functions which offer a variety of optimization landscapes. We evaluate the performance of our algorithm by looking at the runtime and solution fitness of our algorithm compared to the original serial implementation. We have seen evidence that our parallel implementation has achieved quite a persuasive speedup, from which we can conclude that within the same amount of time the serial version runs, our parallel implementation can achieve a better solution quality by exploring significantly more food sources with a higher colony size or number of cycles.

References

[1]Narasimhan, H. (2009). Parallel artificial bee colony (PABC) algorithm. 2009 World Congress on Nature & Biologically Inspired Computing (NaBIC). https://doi.org/10.1109/nabic.2009.5393726

[2] Intelligent Systems Research Group. Artificial Bee Colony (ABC) algorithm homepage. Artificial Bee Colony (ABC) Algorithm Homepage. https://abc.erciyes.edu.tr/

[3] Alzaqebah, Malek & Abdullah, Salwani. (2011). Artificial bee colony search algorithm for examination timetabling problems. International Journal of the Physical Sciences. 6. 4264-4272.

[4] Mehta, S. (2022, May 21). Artificial Bee Colony and its applications to optimization problems. Analytics India Magazine. https://analyticsindiamag.com/artificial-bee-colony-and-its-applications-to-optimization-problems/

[5] Wikipedia. (2023, January 6). Artificial Bee Colony algorithm. https://en.wikipedia.org/wiki/Artificial_bee_colony_algorithm

[6] Karaboga, D., Gorkemli, B., Ozturk, C., & Karaboga, N. (2012, March 11). A comprehensive survey: Artificial Bee Colony (ABC) algorithm and Applications - Artificial Intelligence Review. SpringerLink. https://link.springer.com/article/10.1007/s10462-012-9328-

[7]Subotic, M., Stanarevic, N., & Tuba, M. (2011). Different approaches in parallelization of the artificial bee ... - naun. Different approaches in parallelization of the artificial bee colony algorithm. https://www.naun.org/main/NAUN/ijmmas/20-484.pdf

[8] Karaboga, D. (2010). Artificial Bee Colony algorithm. Scholarpedia. http://www.scholarpedia.org/article/Artificial_bee_colony_algorithm

[9] Project GitHub Repo: https://github.com/TQW0909/EE451_Project.git