15-418: Project Final Report (Ethan Wang and Kaashvi Sehgal

TITLE: Parallelizing the Artificial Bee Colony (ABC) Algorithm

SUMMARY:
The Artificial Bee Colony (ABC) algorithm is an optimization algorithm which mimics the foraging behavior of bees. We will parallelize this algorithm on multi-core CPU platforms using Open-MP and Open-MPI and analyze their performances.

BACKGROUND:
An explanation of the algorithm:
The Artificial Bee Colony (ABC) algorithm is a well known and commonly used algorithm when it comes to optimization problems. It was created based on how honey bee swarms have intelligent foraging behavior. In the basic version of the model, the bees are split into 3 different groups: employed bees, onlookers bees, and scout bees. A main assumption is that there is one food source for each employed bee. The employed bees carry information regarding the food source and then share this information to other bees through their dancing. If an employed bees' food source is abandoned, they will become a scout and search for a new food source. The onlooker bees gain information regarding the food sources and choose them through the dances.

Basic algorithm looks like:

1. Initialize food sources for each employed bee
2. Repeat/loop the following
    - Every employed bee goes to its associated food source and determines how much honey is at the food source then goes to dance in the hive
    - Each onlooker bee watches the dances and selects a food source through a greedy strategy and goes to that source. After it picks a neighbor around it, it determines how much honey is in the neighbor.
    - Figure out which are the abandoned food sources and replace them with new food sources found by the scout bees.
    - Store information about best food source
3. Loop until requirements for best food source are met or until a certain number of iterations

Some further details about the algorithm:

- Important data structures:
    - foodSource = {params[MAX_NUM_PARAM], functionVal, nectarVal, numTrials, onlookerProb}. The food source data structure is arguably the most important data structure for our algorithm. It defines a possible solution to the optimization problem. The array "params" stores the parameters used in the optimization function (aka the solution). FunctionVal, is the output of the optimization function when we plug in the parameters. NectarVal is the fitness of the current solution. NumTrials is how many tries the solution has had to improve upon its value.

Finally, onlookerProb is the probability assigned to the current solution that it is chosen by an onlooker bee to continue to improve the solution.

- Algorithm Inputs:
    - Number of iterations
        - How many iterations should the algorithm perform to get the best possible solution.
    - Maximum number of trials
        - How many tries does each solution get to improve, before we start with a brand new random solution. Used to avoid local minimas when converging.
    - Number of threads
    - Optimization problem (optimization function and number of parameters)
        - Which problem do you want to optimize, how many parameters does the problem take.
    - Colony Size:
        - How many bees in the colony. Also used to determine the number of food sources (solutions).
    - Number of food sources:
        - Number of possible solutions to try for the problem. Determined by colony size/2.
- Algorithm Outputs:
    - Optimized parameters solutions
        - The best possible solution parameters found that minimizes the optimization problem.
    - Resulting optimization output
        - The minimized output resulting from the best solution found.
    - Time taken to compute solution
- Parallelization possibilities:
    - Originally, we thought this algorithm would be parallelized in phases. However, after taking a further look at the code and trying out strategies that will be described below in the approach, we ended up parallelizing iterations rather than parallelizing the separate phases. We ended up trying out both approaches and will present the results of what was the better strategy below.
- Workload:
    - A lot of the workload can be data parallel, since iterations can be done in parallel to find the best possible solution, each thread can perform the optimization using their own set of data.

APPROACH:

How we implemented the algorithm:

- Initialize food sources for each employed bee
    - For each food source, randomly get starting values for each parameter between lower and upper bound of the optimization problem.

- Plug this into the optimization problem and get the function val (output of optimization problem based on current parameters)
- Get the fitness (nectar) value using the following formula where $f_m(x_m)$ is the function value.

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

- Set the number of trials and onlooker probability to 0.

- Repeat/loop the following (for number of iterations/number of threads times)
  1. Employed Bees Phase
     a. Perform the following for each food source.
     b. Choose a random parameter from the solution to adjust.
     c. Using a neighboring food source's value, and a randomly chosen adjustment value, adjust current parameters value only if the fitness of the new random solution is better than it was before.
     d. Increase the number of trials if a better solution was not found. If a better solution was found, set trials to 0.
  2. Onlooker Bees Phase
     a. Perform the following for each food source.
     b. Calculate the onlooker probability by comparing the fitness of the solution to the sum of the fitnesses of all the solutions using this formula.

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

     c. For the number of food sources available, if the onlooker probability is larger than a randomly chosen probability, then choose a random parameter from the solution to adjust.
     d. Using a neighboring food source's value, and a randomly chosen adjustment value, adjust current parameters value only if the fitness of the new random solution is better than it was before.
     e. Increase the number of trials if a better solution was not found. If a better solution was found, set trials to 0.
  3. Store information about best food source
     a. For sequential version: Search through all food sources in current iteration and find the best solution. If the minimum is better than the global best solution from all the previous iterations, store that as the best possible solution.
     b. For parallel version: Each thread searches through all food sources for their current solutions and saves the minimum solution. Then the root thread compares all the solutions from the separate threads and updates

the global minimum solution if needed. The list of food sources is updated to be the list of food sources from the thread that resulted with the best possible solution for this iteration. (All threads are sent this list of food sources to start with on the next iteration).

4. Scout Bees Phase
    a. Replace any food sources that have elapsed their maximum number of trials with a brand new random solution in order to avoid converging to local minimas.

Technologies Used:

For all our versions we used c++. For our parallel implementations, we used openMP for one and openMPI for another.

Parallelization information:

For the parallelization as mentioned earlier we tried two different strategies for the openMP implementation.

1. The first being parallelizing across phases (employed bees). This strategy involved parallelizing across the food sources loop in the phases. Basically each thread would implement the adjustment of a solution. Since the threads were updating different items in the array, there was no need for copies or atomic operations of the foodSource array. There wasn't much of a difference in implementation between this and the sequential version.
    a. Unfortunately this strategy didn't work out that well as the work was already very limited for the employed bees phase, so parallelizing it didn't really make much of a difference in the computation time. Furthermore, it would lead to slightly stale data as threads would be updating neighboring food sources at the same time. Thus we instead went with our second strategy.
2. The second being a slightly more complicated parallel implementation across iterations. Changes were made from the sequential version as rather than threads being split across the same iteration doing parts of a task, we adjusted the algorithm slightly such that each thread did a single iteration, and once each thread had completed one iteration, their solutions would be compared to find the best solution and set of food sources. That specific thread's food source list would then be labeled as the final list and the threads would start the next iteration using that list (but all of them updating their version separately). To be able to do this, we had a massive array of foodSources such that each thread could have their own list of foodSources to edit before a singular thread came together to find the best list.

For the openMPI implementation we simply parallelized across iterations like openMP but made some adjustments to allow message passing instead.

- Similar to the openMP version, we maintained the same parallelization axis, where it was across iterations rather than a specific phase. However, we had to make some adjustments such that information could be passed to the root thread that would determine which thread had the best solution. Each thread would message pass their

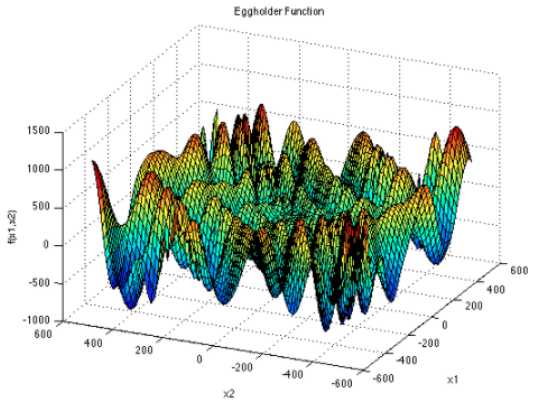
minimum solution and which specific solution had that minimum, such that the root thread would be able to determine which thread had the best solution. Once it has determined this, it broadcasts this to all the threads. Then the thread with the best solution will update the overall global solution if necessary, and perform the scout bees phase. Once it has updated its list, it will broadcast the final updated list to all the threads such that the threads can begin their next iteration with the updated list.

RESULTS:

For the below tests, we will be benchmarking our implementations running on the PSC machines. We will be optimizing the eggholder function on the domain [-512, 512], where the optimum lies at (512, 404.2319).

When plugged in, f((512, 404.2319)) = -959.6407.

EGGHOLDER FUNCTION



$$f(\mathbf{x}) = -(x_2 + 47)\sin\left(\sqrt{\left|x_2 + \frac{x_1}{2} + 47\right|}\right) - x_1\sin\left(\sqrt{|x_1 - (x_2 + 47)|}\right)$$

When it came to this algorithm we had 2 main aspects that we were analyzing when it came to determining how successful our implementation was.

- Speedup
  - We would measure speedup using the computation time it took to complete running the algorithm using a different number of threads.
- Accuracy of solution
  - The accuracy of our solution was determined by how well the algorithm could determine the minima of a specific optimization function, that being the Eggholder function above. We measured the accuracy by determining the percentage error rate in comparison to the ideal minima of the function. This was calculated by $percent\ error\ = \frac{|actual - ideal|}{ideal} * 100$. This would allow us to see how accurate the found solution was to the ideal expected solution.

For our results, we adjusted the main parameter that affected how much work was being done in order to see how parallelization would vary across threads. Furthermore, this would allow us to see if we hit barriers to speedup when colony size is small but number of threads is maximal.

We kept the number of iterations the same such that we could see how the work varied given a greater number of threads. Additionally, this would allow us to see how it affects how optimal our solutions are.
The following are our raw data tables from when we gathered results.

Sequential

Using 1000 Iterations

Num Threads	Colony Size (# of food sources * 2)	Computation Time	Solution	Error Rate
1	100	0.028693	(512, 402)	0.6%
1	1,000	0.477035	(483, 433)	0.8%
1	10,000	4.791637	(512, 404)	0.01%

Open-MP

Using 1000 Iterations and (Colony size = (# of food sources * 2))

Num Threads	Colony Size	Computation Time	Solution	Error Rate
1	100	0.047329	(430, 445.36)	4.2%
2	100	0.046947	(453.8, 463)	7%
4	100	0.089784	(481, 430)	2.6%
8	100	0.105028	(435, 450)	2.8%
16	100	0.109899	(442, 456)	2.7%
32	100	0.070765	(445, 458)	3.3%
64	100	0.108065	(463, 418)	5.18%
128	100	0.121748	(512, 404.05)	~0%

Num Threads	Colony Size	Computation Time	Solution	Error Rate
1	1000	0.414990	(512, 404)	0.01%
2	1000	0.452005	(512, 404)	0.01%
4	1000	0.348033	(465, 421)	4.79%
8	1000	0.435684	(473, 428)	1.2%
16	1000	0.410644	(476, 427)	1%
32	1000	0.322983	(439, 454)	2.5%
64	1000	0.241572	(476, 431)	0.6%
128	1000	0.257324	(481, 431)	0.3%

Num Threads	Colony Size	Computation Time	Solution	Error Rate
1	10000	4.246774	(512, 404.2)	~0%
2	10000	3.294000	(477, 427)	0.8%
4	10000	1.652219	(512, 404.4)	~0%
8	10000	0.949860	(512, 403.26)	0.1%
16	10000	1.584100	(512, 404.15)	1%
32	10000	1.067125	(512, 404.2)	~0%
64	10000	0.719320	(512, 404.27)	~0%
128	10000	0.597256	(512, 404.2)	~0%

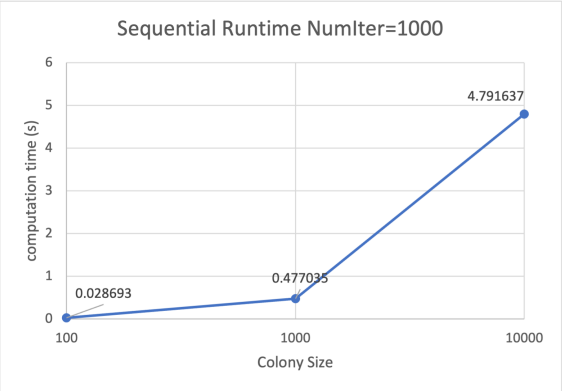
Open-MPI

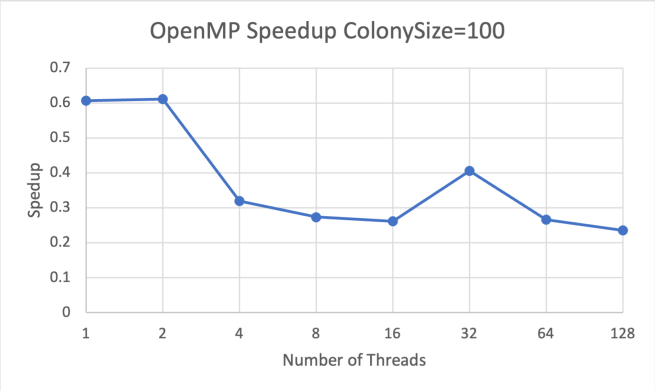
Using 1000 Iterations and (Colony size = (# of food sources * 2))

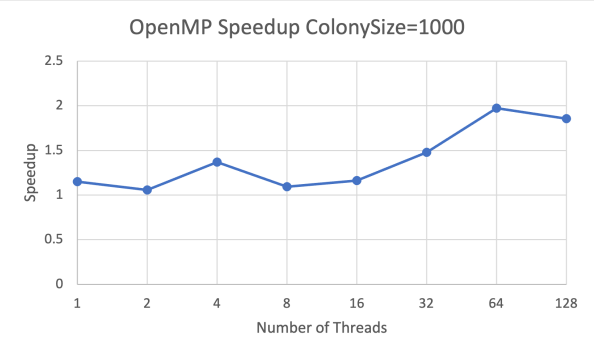
Num Threads	Colony Size	Computation Time	Solution	Error Rate
1	100	0.025506	(433, 448)	3.3%
2	100	0.014069	(512, 404)	0.24%
4	100	0.007857	(-458, 386)	7.6%
8	100	0.004052	(436, 450)	2.85%
16	100	0.002393	(471, 428)	2.14%
32	100	0.001404	(512, 404)	0.01%
64	100	0.001281	(512, 403)	0.14%
128	100	0.001964	(512, 404)	0.01%

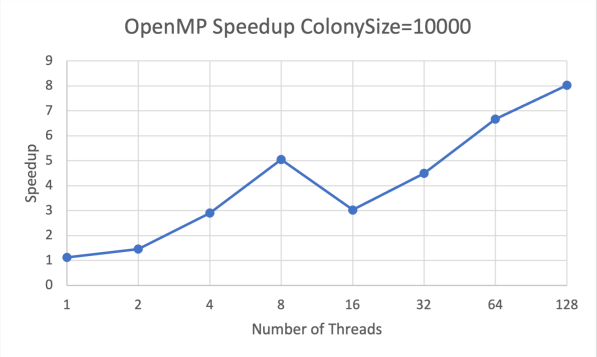
Num Threads	Colony Size	Computation Time	Solution	Error Rate
1	1000	0.416232	(440, 454)	2.5%
2	1000	0.210541	(480, 430)	2.34%
4	1000	0.111334	(470, 425)	2%
8	1000	0.057021	(478, 429)	0.89%
16	1000	0.029140	(509, 399)	1.7%
32	1000	0.015764	(512, 404.2)	~0%
64	1000	0.009399	(512, 404.2)	~0%
128	1000	0.007910	(481, 431)	0.3%

Num Threads	Colony Size	Computation Time	Solution	Error Rate
1	10000	4.260838	(471, 426)	1.8%
2	10000	2.159576	(512, 403)	0.18%
4	10000	1.121985	(512, 405)	0.2%
8	10000	0.568482	(512, 404.2)	~0%
16	10000	0.284037	(512, 404.3)	~0%
32	10000	0.144596	(512, 404.2)	~0%
64	10000	0.071906	(512, 404.2)	~0%
128	10000	0.041859	(512, 404.2)	~0%

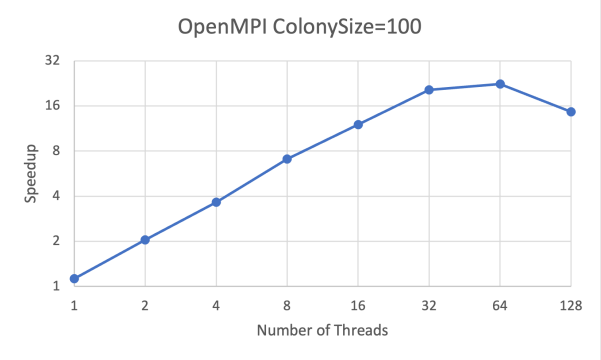


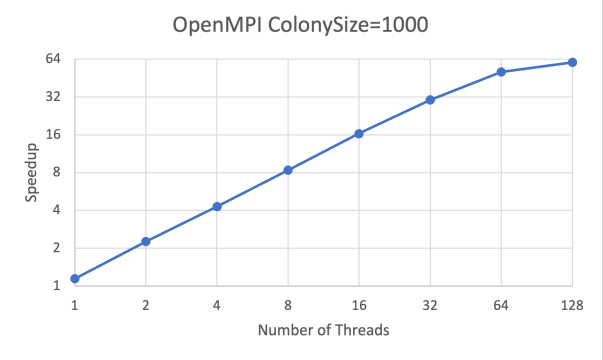


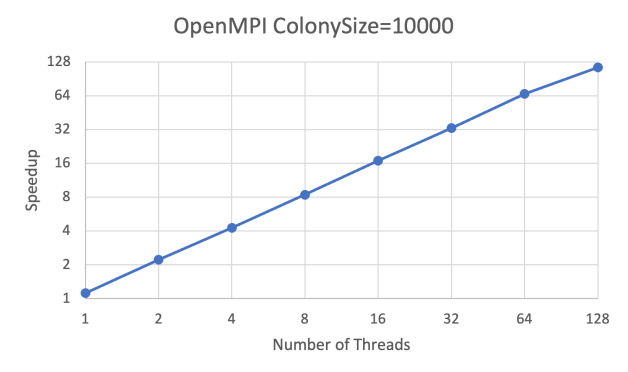




OpenMP graphs are on a linear scale and OpenMPI is on logarithmic scale







Results and Parallelization Analysis:

Sequential:
For our sequential algorithm, we saw that as we increase the colony size, the computation time increases. This is because the colony size is used to determine the number of food sources (possible solutions) to iterate on, and is what the main workload of the algorithm is defined by. As a result, it may seem like a good idea to parallelize across the food sources, however, this caused major issues with stale data and not as great optimizations as adjustments to food sources were made by neighboring food sources. If different threads were changing the neighboring food sources at the same time, it might lead to stale data being taken. Furthermore, the overhead of consistently spawning and synchronizing threads for each phase leads to synchronization dominating the computation time. Lots of overhead lead to terrible speedups. Instead, as mentioned earlier, we decided to parallelize across iterations in order to truly get the best optimization, runtime, and speedup possible. In this case since we were dividing number of iterations by number of threads, there is still some stale data, but the speedup is a trade-off for it.

OpenMP:
The speedup is measured against the computation time required for the sequential version. Since we wanted to see how speedup was affected when we adjusted the colony size, we tested 3 colony sizes of 100, 1000, and 10000.

With colony size 100, the workload is so small that computation finishes within 0.05 seconds. In fact, the parallel version with 1 thread was actually slower than the sequential version. This is due to the fact that there is overhead of performing some of the specific operations that were adjusted from the sequential version to make it parallelizable. Because of this, as we increase the threads we don't actually see much of a speedup. Rather, the computation time actually increases due to the overhead required to initialize a larger number of threads dominates the benefits of parallelization.

For the colony size of 1000, we definitely see improvements from colony size 100, where there is some speedup, but even then the speedup is small. Even up to 128 threads, we are only getting a maximum of around 2x speedup. The same thing that was occurring with the colony size of 100 is occurring here, the synchronization overhead is just majorly dominating overall computation time. When we broke down the computation time further and measured how long an iteration would take for a specific solution, it was very similar to the sequential version.

Finally, for the colony size of 10,000, we begin to see the benefits of parallelization. We began seeing a better speedup, however, it was still not that great. The main reason that this was occurring was because on every iteration, we would have to go through the entire process of initializing a set of threads to divide up the work, and then join them at the end of the iteration. However, the workload is much bigger for each iteration so computation time begins to outweigh the synchronization overhead.

OpenMPI:
In contrast to the OpenMP version, we see significant speedups occurring in all three colony sizes tested for the OpenMPI version.

Similarly to OpenMP, the colony size of 100, 1000 doesn't have near perfect speedup, as considering the workload is decently small, the sequential version takes only about 0.02 seconds, thus what we can see is that there is still overhead required to initialize the openMPI threads that prevents from speedup being perfect. However, there is still significant speedup.

For colony size of 100, we see the maximum speedup is for 64 threads, then it dips down for 128 threads. Once again this is caused due to overhead of initializing threads, and slightly due to message passing. Within our implementation, there is only minor message passing, except for one major broadcast of an array of foodSources. This also results in some additional time taken that increases as the number of threads increases. As a result, this also affects the speedup for the smaller computation time.

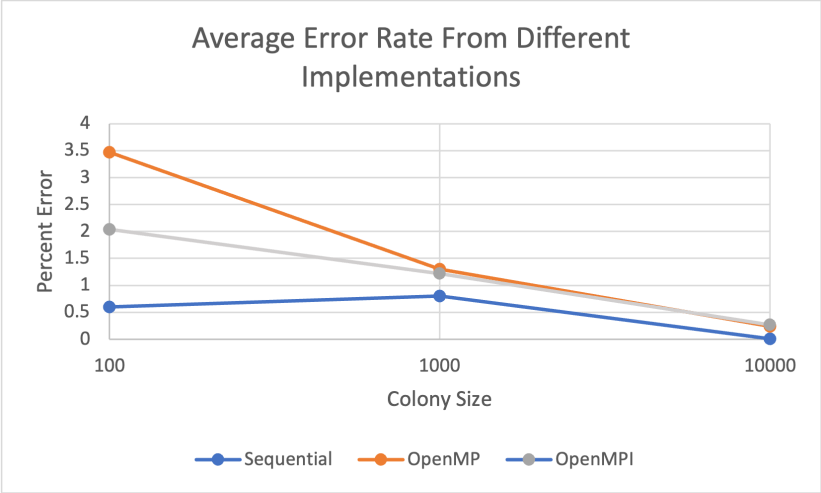
For colony size of 1000, we see improvements in the speedup as overall workload has now increased enough such that small things like thread initialization and message passing is not dominated for threads below 64. However we see the speedup hits a boundary of sorts for 64 and 128 threads. This is where it is likely that there cannot be much to be improved upon, as the computation time is already significantly low, the message passing and thread synchronization now dominates. Where the sequential version took about 0.5s, the parallel version with 128 threads took 0.00791 seconds.

Finally, for a colony size of 10,000, we see near perfect speedup. In contrast to the smaller colony sizes, the workload takes up a larger amount of time, which makes the thread synchronization and message passing no longer dominate the computation time.

Why OpenMPI was significantly better than OpenMP:
As we can see from our above analysis, there is a significant difference in the speedups between the two versions. This actually comes from how we are initializing threads. For the openMPI version, we only have to initialize the threads once. As we perform the iterations, since the threads are already initialized they just have to carry out the workload. Additionally, in our openMPI version there is only very little message passing that occurs in comparison to the rest of the computation, which means the overhead of synchronization is very low.

On the other hand, for the openMP version, every single time we must perform an iteration we have to reinitialize the threads and join them at the end of the iteration. The overhead caused by reinitializing the threads over and over again for each iteration massively affects the overall speedup. Maybe in different situations where we were not performing iterations, openMP is better than openMPI, but for our implementation of the ABC algorithm, it is clear that openMPI is the preferred format of parallelization that allows us accurate optimal solutions, while also providing massive speedup.

Error rate and Workload Analysis:



One of the major aspects when it came to perfecting the optimization problem was determining what size the colonies should be. The reason why this is important is because colony size determined how many food sources (aka possible solutions) we would be looking at for each iteration. What we can see is as colony size increases there is definitely a clear difference in the average error rate across all the implementations. This is because with a larger number of possible solutions, we are able to randomize more and adjust the solutions more to get closer to the optimal solution. Additionally, we can see that the sequential version is the most accurate across the board. This is because the sequential version never operates on stale data, whereas the parallel implementations are not as optimal. However, as colony size increases the error rate drastically decreases to the point where the parallel implementations almost perform as well as the sequential version in terms of accuracy.

REFERENCES

"Artificial Bee Colony (ABC) Algorithm Homepage." *Artificial Bee Colony (ABC) Algorithm Homepage*, https://abc.erciyes.edu.tr/.

"Artificial Bee Colony." *Artificial Bee Colony - an Overview | ScienceDirect Topics*, https://www.sciencedirect.com/topics/computer-science/artificial-bee-colony.

Karaboga, Dervis. "Artificial Bee Colony Algorithm." *Scholarpedia*, http://www.scholarpedia.org/article/Artificial_bee_colony_algorithm.

Ram, Naresh. "Implementing Artificial Bee Colony Algorithm to Solve Business Problems." *Medium*, Towards Data Science, 31 Mar. 2022, https://towardsdatascience.com/implementing-artificial-bee-colony-algorithm-to-solve-business-problems-cb754f3b9255.

Xu, Yunfeng, et al. “A Simple and Efficient Artificial Bee Colony Algorithm.” *Mathematical Problems in Engineering*, Hindawi, 12 Feb. 2013, https://www.hindawi.com/journals/mpe/2013/526315/.

LIST OF WORK BY EACH STUDENT, AND DISTRIBUTION OF TOTAL CREDIT

Work Done By Both
Sequential Implementation
OpenMP implementation
Report

Work Done By Ethan
Video and Presentation

Work Done By Kaashvi
OpenMPI Implementation

Overall distribution of Credit = 50%-50%.