A Comparative Study of OpenMP and MPI for Parallelizing the Grey Wolf Optimizer

15618: Parallel Computer Architecture and Programming

Spring 2023

Tong Jin tongj@andrew.cmu.edu

Yiding Chang yidingch@andrew.cmu.edu

Instructor: Zhihao Jia, Brian Railing

Project Link: https://github.com/tongjin0521/15x18-final-project

Contents

1 Summary

We conducted a comparative study of OpenMP and MPI for parallelizing the Grey Wolf Optimizer algorithm. Our project aimed to explore the performance of these two parallelization methods on different computing resources, including GHC machines and PSC machines. We implemented both methods and compared their performance in terms of speedup and efficiency. Our results showed that OpenMP was outperformed by MPI in terms of speedup and efficiency on both GHC and PSC machines. Overall, our study provides insights into the strengths and weaknesses of OpenMP and MPI for parallelizing computationally intensive algorithms like the Grey Wolf Optimizer.

2 Background

The Grey Wolf Optimizer (GWO) is a nature-inspired metaheuristic algorithm that imitates the hunting behavior of grey wolves to optimize problem-solving. The algorithm is computationally intensive and requires a decent number of iterations to achieve an optimal solution. GWO mimics wolves' hunting behavior and works on the same principle that wolves move towards their prey. In this context, the position of each wolf signifies a potential solution to the optimization problem. The wolves hunt based on the alpha wolf's position, the beta wolf's position, and so on, which are updated at every iteration.

According to Al-Tashi et al., [1], the feature selection method based on Grey Wolf Optimization is effective for coronary artery disease classification, where the Support Vector Machine classifier (SVM) is being used as the fitness function of GWO. Detailed updates for each agent can also be seen here.

The dataset is rows of data, where each row is an array of features (of dimension D as specified in the algorithm above) plus the predicted attribute, and all of them are

Algorithm 1 Grey Wolf Optimization (GWO)

Require: D: number of dimensions, N: population size, max_{iter}: maximum number of iterations, $\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\delta}$: initial search agents

Initialize the position of search agents
Initialize iteration counter $iter = 0$
while $iter < max_{iter}$ **do**
    for $i = 1$ to N **do**
        Evaluate the objective function $f(\boldsymbol{x})$ for the current search agent
        if $f(\boldsymbol{x}) < f(\boldsymbol{x})_\alpha$ **then**
            Update $\boldsymbol{\alpha}$ with the current search agent
        else if $f(\boldsymbol{x})_\alpha < f(\boldsymbol{x}) < f(\boldsymbol{x})_\beta$ **then**
            Update $\boldsymbol{\beta}$ with the current search agent
        else if $f(\boldsymbol{x})_\beta < f(\boldsymbol{x}) < f(\boldsymbol{x})_\delta$ **then**
            Update $\boldsymbol{\delta}$ with the current search agent
        end if
    end for
    for $i = 1$ to N **do**
        for $j = 1$ to D **do**
            Update the position of each search agent based on $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\delta}$
        end for
    end for
    Increment iteration counter $iter \leftarrow iter + 1$
end while
return $\boldsymbol{\alpha}$

interpreted as a double. Each feature has a real-life meaning to help coronary artery disease classification. The features are listed in the table 1, where the Chest pain type has four values: typical angina, atypical angina, non-anginal pain, and asymptomatic.

No	Attributes	Description
1	Age	Age in year
2	Sex	0 for female and 1 for male
3	Cp	Chest pain type
4	Trestbps	Resting blood sugar in mm Hg on admission to the hospital
5	Chol	Serum cholesterol in mg/dl
6	Fbd	(Fasting blood sugar $>$ 120 mg/dl) (1 = true; 0 = false)
7	Restecg	Resting ECG result
8	Thalach	Maximum heart rate achieved
9	Exang	Exercise-induced angina
10	Oldpeak	ST depression induced by exercise relative to rest
11	Slope	Slope or peak exercise ST segment
12	Ca	Number of major vessels colored by fluoroscopy
13	Thal	Defect type
14	num	The predicted attribute

Table 1: Attributes of the Dataset

The key data structure here is the structure of each search agent. It is an array of double. The array is of dimension D as specified in the algorithm above. Each element of the array is a double value ranging from 0 to 1, representing the probability of using this column of data in the SVM classifier. For example, if we have an array of {0,1,1,0}, then the second and third columns are to be used in the SVM when calculating the fitness value, and the first and fourth columns of data are discarded in this case.

There are two key operations on these data structures. The first one is using the objective function $f(\boldsymbol{x})$ to calculate the fitness value or score of this current search agent/wolf, which we are using SVM with the support of libsvm[3] to do. The second one is calculating $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\delta}$, and updating each agent based on these three values.

The algorithm's inputs are the dataset, and the initial positions of the search agents are assigned from a random number generator, which is uniformly distributed in the range [0,1). The output is $\boldsymbol{\alpha}$, as specified in the algorithm. $\boldsymbol{\alpha}$ is the best search agent

we have found, so we return it as the final output.

The most computationally expensive part is evaluating the objective function $f(\boldsymbol{x})$ for the current search agent, especially given the fact that the SVM is not guaranteed to converge, resulting in reaching the max number of iterations in SVM training. Also, calculating $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\delta}$ involves using each search agent's data, so this step is also not cheap, especially in terms of communication cost.

As is shown in the algorithm, each iteration of the GWO algorithm is based on the previous iteration. So there is no parallelism here. For each iteration, the GWO algorithm can be parallelized among search agents by employing parallelization on fitness evaluation functions and position update functions. This is because the fitness evaluation functions and position update functions are independent of each search agent. These functions can be concurrently executed by multiple threads or processes on various computing resources, such as high-performance computing clusters or multi-core CPUs. Parallelizing these functions reduces the computation time, allowing the algorithm to converge toward an optimal solution faster. However, we do need to mention here, within one iteration, all search agents need to communicate with each other to select $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\delta}$, in order to update their positions correspondingly. With that being said, each search agent is dependent on each other since they need to wait for the slowest one to finish calculating the fitness value.

3 Approach

Our project aims to compare the performance of two parallelization methods, OpenMP and MPI, in parallelizing the GWO algorithm. OpenMP is a shared memory parallelization technique that allows the parallel execution of code blocks within a single program on multi-core CPUs. In contrast, MPI is a message-passing parallelization technique that enables multiple processes to communicate and synchronize on different

computing resources. We are comparing the difference on both GHC machines and PSC machines, where the processor on the GHC is an Intel® Xeon® Processor E5-1660 v4 and the processor on PSC is an AMD EPYC 7742 64-Core Processor.

Therefore, to start with, we used C++ to implement our sequential version. There exists a simple Python implementation of the GWO algorithm here as our reference [2]. However, it does not utilize SVM when evaluating the objective function $f(\boldsymbol{x})$ for each search agent, and it does not involve any parallelization. So we needed to come up with our own C++ sequential version with SVM as the objective function with the help of libsvm[3].

The data structures and operations described above map to machine concepts like cores and threads, which are the basic building blocks of parallel computing. Each core or thread would be responsible for executing a separate task, like calculating the fitness value for a specific search agent, allowing multiple tasks to be executed simultaneously. OpenMP and MPI in the implementations below would allocate the available threads or cores on the target machines to execute the parallel sections of the code. The data structures involved in the computation would be each search agent.

As is shown in the algorithm, there are many iterations of optimization in GWO. However, each iteration of the GWO algorithm is based on the previous iteration. So each iteration needs to wait for the previous iteration to be finished, and there is no parallelism or speedup here. For various nodes in one iteration, they also need to synchronize to get prepared for the next iteration.

3.1 OpenMP

We utilized OpenMP statements to achieve parallelization and speedup. The problem was mapped to the target parallel machine(s) using OpenMP statements for parallelization and speedup. There are two places where we do parallelization.

1. In the first place, parallelization was achieved by distributing the task of calcu-

lating fitness values among each search agent. Each search agent's calculation is independent of the others, which allows us to take advantage of multiple cores or threads available on the target machine(s) to speed up the computation. The data structures involved in this step are the search agents and based on that, the fitness values are calculated. Each thread or core would work on a separate search agent, updating its corresponding fitness value. However, updating $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\delta}$ is not independent, and to prevent conflicts, *#pragma omp critical* was used to protect the section where these values are updated.

2. In the second place, the search agents are updated given $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\delta}$. Here, in the sequential version, we have two for loops. One is iterating over all the search agents, and the inner loop is iterating over each data element in the search agent. However, each data element within one search agent is independent, and each search agent is independent of each other as well. Therefore, we merged them into a big for loop, which is now iterating from 0 to Number of Search Agents * dim. We added OpenMP statements over the merged big loop to achieve better speedup since we now divide the work into smaller chunks. So now each thread or core is actually updating a single double value based on $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\delta}$.

We also did experiments with different scheduling policies, like static, dynamic, guided, and auto. The details can be seen in the results section.

3.2 MPI

We utilized MPI as well to achieve parallelization and speedup. There are basically two places to do communication.

1. The master node will gather all search agents' fitness values and their positions. This is because the master node needs all search agents' fitness values to know $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\delta}$, and save their corresponding positions.

2. After the master node sorts the fitness values of all wolves and get $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\delta}$, it will scatter the information across all search agents so that each search agent can update their positions according to $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\delta}$.

Here, we use `MPI_Isend` and `MPI_Recv` to achieve communication between search agents.

4 Results

To determine the effectiveness of each parallelization method in accelerating the GWO algorithm, the project evaluates their performance on different resources. By comparing the results obtained from the two methods, it is possible to determine which method is more suitable for parallelizing the GWO algorithm. Moreover, it can provide insights into the best practices for parallelizing other nature-inspired metaheuristic algorithms. The performance is measured by the execution time of various implementations given different configurations. The sequential version, as the baseline, is run on the GHC machine as a single-threaded CPU code.

We have run our code on both GHC machines and PSC machines, where the processor on the GHC is Intel® Xeon® Processor E5-1660 v4 and the processor on PSC is AMD EPYC 7742 64-Core Processor. The performance results related to various factors are shown in Fig 1 and 2. We can see similar results on both machines, so we are not comparing the performance between these two machines. Instead, we are comparing the results between various code implementations.

4.1 Random Results

Unlike most algorithm implementations, the GWO algorithm involves generating random numbers. Although we can set the seed of the random number generator, since all threads are running in parallel and grabbing multiple random values, we can not guarantee the detailed order of how the threads are executed, so for each run, each

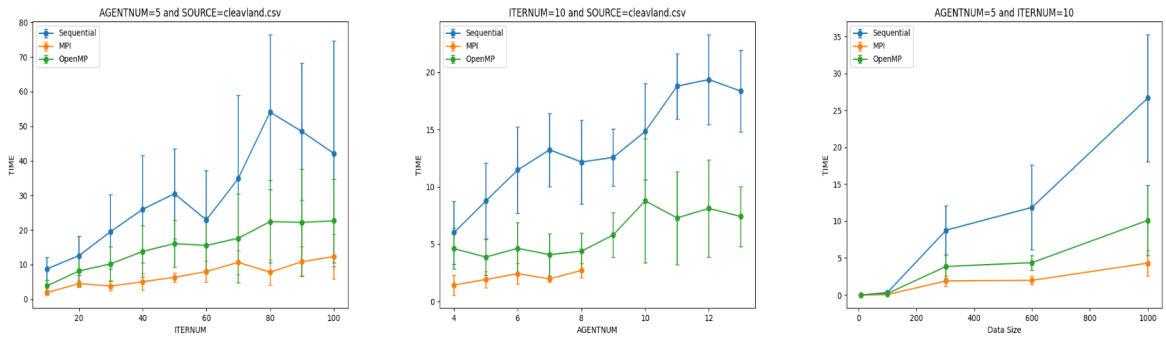


Figure 1: Performance on GHC Machine

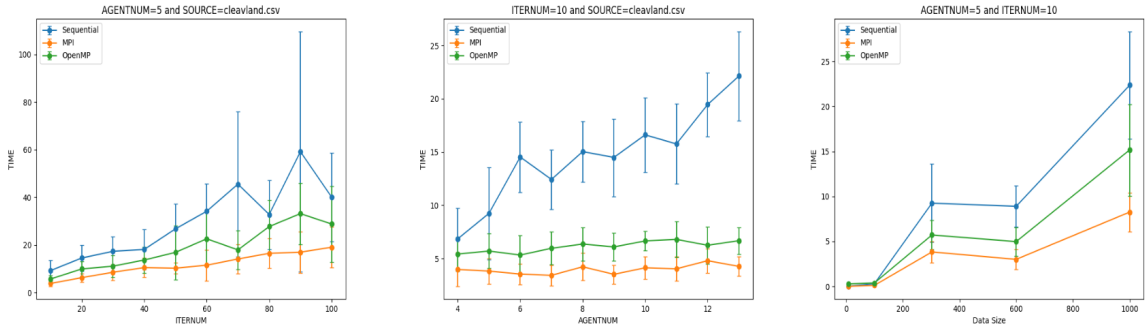


Figure 2: Performance on PSC Machine

thread has different random values, resulting in various performance. Therefore, our results are non-deterministic. Hence, we have run 10 iterations to get the mean and stand deviation of the performance. As is shown in the figures, the center point is the mean time for 10 iterations, and the error bar stands for the standard deviation among these 10 iterations.

4.2 Trends of error bar sizes

From the plots, we can see that when the required time increases, the error bars tend to increase in size. This is because as the execution becomes more time-consuming, the randomness gives the program more degrees of freedom to generate the results. The error bars also suffer from a few outliers, as can be seen in the "ITERNUM" plot for the PSC machines. We have made the decision to keep the outliers because we think it is a normal result of the randomness.

4.3 Effects of the Number of Iterations

In the analysis of the results, it was observed that the sequential version exhibits erratic behavior as the number of iterations increases, which is attributed to the randomness in the implementation. This is reflected in the larger error bars for a higher number of iterations, as explained in the preceding section. It is expected that the error would increase linearly with the number of iterations.

In contrast, both the OpenMP and MPI versions display a noticeable speedup due to parallelization. The execution time for these versions increases linearly with the number of iterations, with a little bit of turbulence due to randomness though. This is anticipated since more iterations imply more computational work. Moreover, it is noteworthy that the additional work cannot be parallelized with the preceding workload owing to the interdependence of iterations. Hence, the execution time gets increased normally.

4.4 Effects of the Number of Agents

Our experimental results show that the execution time for the sequential version increases as the number of agents increases, as expected since more agents lead to more iterations in the GWO algorithm for loops.

We also observed that the performance of the OpenMP and MPI versions was consistent on the PSC machine. Similarly, on the GHC machine, we observed consistent performance when the number of agents was less than or equal to 8. This is because the GHC machine has 8 cores, and each core is assigned to one agent, enabling parallel execution of the GWO algorithm. In contrast, the PSC machine has more cores than required, allowing for the allocation of additional threads to execute extra jobs in parallel with the previous work when the number of agents is less than the number of cores.

However, when the number of agents is greater than the number of cores, we observed an increased execution time due to some threads having to perform extra jobs compared to the others. This is the case for the GHC machine having more than 8 agents. Nonetheless, we did observe consistent performance when the number of agents was between 8 and 13, as the extra work was evenly distributed among the available cores. Specifically, assigning one node to do one extra job is equivalent to assigning three nodes to do three extra jobs, resulting in a consistent execution time for the GWO algorithm implementation.

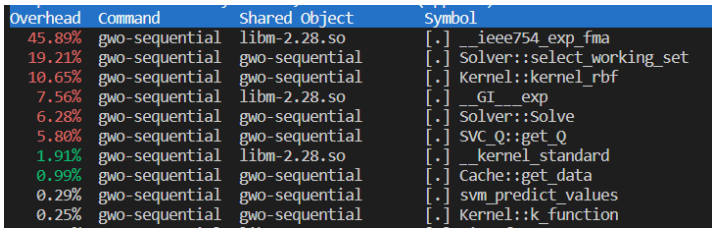
4.5 Effects of Data Size

The findings indicate a positive correlation between the size of the data and the execution time for all three implementations. As the data size increases (the data size is 9, 100, 303, 600, 1000), the SVM calculation requires more computation, which leads to a decrease in the processing speed of each node, consequently affecting the overall execution time. These trends were observed across all three versions and were found to be almost linear. However, some level of turbulence was detected due to the random nature of the data.

4.6 Bottlenecks – results of `perf`

Running `perf` on the sequential version of the algorithm resulted in the report shown in Figure 3. The most time-consuming functions are all from the C++ SVM implementation (The `__ieee754_exp_fma` function is an implementation of the exponential function, which is also used in the SVM fitness function.). This is in line with our expectations for SVM to be the most time-consuming step of the algorithm.

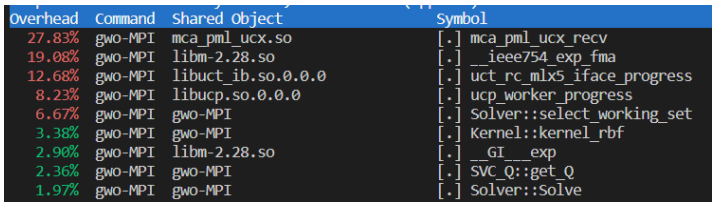
Running `perf` on the MPI version of the algorithm resulted in the report shown in Figure 4. The most time-consuming function is the receiving function of MPI, and the runner-ups are all from the C++ SVM implementation. This indicates that while the



Overhead	Command	Shared Object	Symbol
45.89%	gwo-sequential	libm-2.28.so	[.] __ieee754_exp_fma
19.21%	gwo-sequential	gwo-sequential	[.] Solver::select_working_set
10.65%	gwo-sequential	gwo-sequential	[.] Kernel::kernel_rbf
7.56%	gwo-sequential	libm-2.28.so	[.] __GI___exp
6.28%	gwo-sequential	gwo-sequential	[.] Solver::Solve
5.80%	gwo-sequential	gwo-sequential	[.] SVC_Q::get_Q
1.91%	gwo-sequential	libm-2.28.so	[.] __kernel_standard
0.99%	gwo-sequential	gwo-sequential	[.] Cache::get_data
0.29%	gwo-sequential	gwo-sequential	[.] svm_predict_values
0.25%	gwo-sequential	gwo-sequential	[.] Kernel::k_function

Figure 3: The `perf` results of the sequential version of the algorithm.

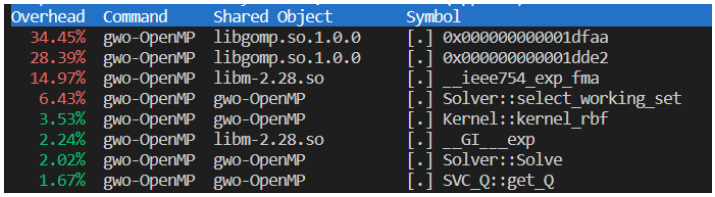
SVM implementation is time-consuming, the MPI workload is unbalanced among the processors such that a lot of time is spent on waiting for data to be transferred.



Overhead	Command	Shared Object	Symbol
27.83%	gwo-MPI	mca_pml_ucx.so	[.] mca_pml_ucx_recv
19.08%	gwo-MPI	libm-2.28.so	[.] __ieee754_exp_fma
12.68%	gwo-MPI	libuct_ib.so.0.0.0	[.] uct_rc_mlx5_iface_progress
8.23%	gwo-MPI	libucp.so.0.0.0	[.] ucp_worker_progress
6.67%	gwo-MPI	gwo-MPI	[.] Solver::select_working_set
3.38%	gwo-MPI	gwo-MPI	[.] Kernel::kernel_rbf
2.90%	gwo-MPI	libm-2.28.so	[.] __GI___exp
2.36%	gwo-MPI	gwo-MPI	[.] SVC_Q::get_Q
1.97%	gwo-MPI	gwo-MPI	[.] Solver::Solve

Figure 4: The `perf` results of the MPI version of the algorithm.

Running `perf` on the OpenMP version of the algorithm resulted in the report shown in Figure 5. The most time-consuming functions are from the OpenMP library, but we are unable to tell which ones in particular. The runner-ups are all from the C++ SVM implementation.



Overhead	Command	Shared Object	Symbol
34.45%	gwo-OpenMP	libgomp.so.1.0.0	[.] 0x000000000001dfaa
28.39%	gwo-OpenMP	libgomp.so.1.0.0	[.] 0x000000000001dde2
14.97%	gwo-OpenMP	libm-2.28.so	[.] __ieee754_exp_fma
6.43%	gwo-OpenMP	gwo-OpenMP	[.] Solver::select_working_set
3.53%	gwo-OpenMP	gwo-OpenMP	[.] Kernel::kernel_rbf
2.24%	gwo-OpenMP	libm-2.28.so	[.] __GI___exp
2.02%	gwo-OpenMP	gwo-OpenMP	[.] Solver::Solve
1.67%	gwo-OpenMP	gwo-OpenMP	[.] SVC_Q::get_Q

Figure 5: The `perf` results of the OpenMP version of the algorithm.

4.7 Comparison between different parallelization implementations

From all the plots attached, it can be seen that the MPI version outperforms the OpenMP one. We think this might be due to that the OpenMP implementation relies on the usage of shared memories, and the information of the best-performing wolves are

kept in shared memories and accessed by different processes concurrently, which causes frequent critical section accesses and modifications. This might lead to the OpenMP version performing poorly.

The MPI version, on the other hand, does not rely on any shared memory and instead relies on the master process telling the other processes about the information of the best-performing wolves. Different processes can communicate with the master process in parallel with asynchronous communication, so it will not cause a huge synchronization problem as might be the case for OpenMP. From this perspective, we conclude that MPI is the more fitting parallelization implementation under the circumstances of this problem.

4.8 Scheduling Policies

We tried replacing all scheduling policies with all static, all dynamic, all guided, and all auto. The result is shown in the figures 6. We do want to mention here that the results in the figures are based on only one run. Since we have random numbers in our implementation, each run will vary a lot, even given the same configurations. Therefore, it's not useful to look at specific values. Instead, we should look at the trend.

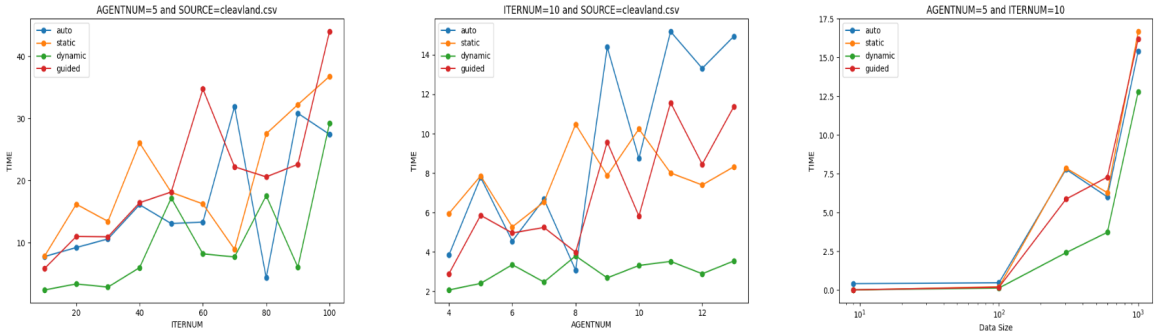


Figure 6: Effect of Different Scheduling Policies

Generally speaking, using dynamic scheduling can achieve the best performance, and others share similar results. Dynamic means each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.

With that being said, idle threads will try to steal tasks from other jobs, which will help improve the imbalance of tasks. This improvement is most obvious in unbalanced cases. At the same time, dynamic introduces overhead, decreasing the performance on some balanced tasks. However, for static, each thread has its own task, and will not steal. For guided, the difference with dynamic scheduling type is in the size of chunks. Guided policies assign chunks dynamically. However, at first, the chunk is too large in size, causing imbalance problems again. For auto, it delegates the decision of the scheduling to the compiler and/or runtime system. But it's hard for it to predict the execution time of the SVM, so it appears not much useful. Therefore, dynamic policies have the best result. So we should go with dynamic policies.

5 List of Work by Each Student, and Distribution of Total Credit

In accordance with the project requirements, the workload was distributed equally between the two team members, with Tong Jin being assigned to implement both the sequential and OpenMP versions, while Yiding Chang was responsible for developing the MPI version. To ensure a balanced contribution, we collaborated closely to generate and analyze the results obtained from each implementation. It is worth noting that an equal distribution of effort was maintained throughout the project, with both team members contributing equally to its success. The distribution should be 50% - 50% in the result.

References

[1] Qasem Al-Tashi, Helmi Rais, and Said Jadid, Feature Selection Method Based on Grey Wolf Optimization for Coronary Artery Disease Classification. International

Journal of Engineering and Advanced Technology (IJEAT), vol. 9, no. 6, pp. 5091-5097, Jun. 2020.

[2] KaushalSahu, Grey-Wolf-Optimization. GitHub, Nov. 19, 2019.

[3] C.-C. Chang and C.-J. Lin. LIBSVM : a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1–27:27, 2011. pdf, ps.gz, ACM digital lib.