Assignment #3: Task Parallelism in the Genetic Algorithm for the Knapsack problem.

I decided to make the project in Java amongst the chosen languages since it is the one I'm more accustomed to. To make the project and follow the instruction given by the professor I started by analysing both demonstrations worked in class. I decided to use the one used in the LiveCoding folder since to me it seemed the simplest and the best to work on, I also used the default message.Class used in that project to develop the other messages classes I would use.

And with the instruction that each actor should do only one thing I created 5 actors' class that extend the Author.java:

- ActorManager: Responsible for creating the initial population like in Knapsack.java and for handling the messages, no message is transferred directly between actors, it works as a middleman, receives a message from Actor A and decides to send it to the appropriate actor, Actor B.

- FitnessActor: Responsible for calculating the fitness of the initial population, receives a PopulationMessage sent from the ActorManager with the population that it creates, like in the KnapSack project, and it sequentially calculates the fitness and then sends a message of the type FitnessPopulationMessage with the measured population to the ActorManager.

- BestFitnessActor: Will receive a FitnessPopulationMessage with the measured population from FitnessActor and will calculate the element with the best fitness and will print the best element with the corresponding fitness, it sends no message to actor Manager.

- CrossOverActor: To maintain synchronization and to avoid that generations aren't being skipped or done at the same time, only after FitnessActor is done will the crossover actor start, it will execute at the same time as the BestFitnessActor since they don't intervene with each other. It will receive the same population that FitnessActor receives with a PopulationMessage sent from the ActorManager and will determine the one with BestFitness and make the crossover just like in KnapSack, uses the tournament function for the parents and creates a new population. It will send a CrossOverPopulationMessage to the actorManager with the newPopulation Array.

- MutateActor: The last actor works just like the remaining ones; it will receive a CrossOverPopulationMessage from the CrossOverActor with the new population and will mutate some elements the same way KnapSack does and when it's done it will send a MutatedPopulationMessage to the manager.

The first generation happens when ActorManager receives the start message sent from the Main. That will initialize the FitnessActor with the population created by the manager and will work accordingly to what is specified above, however when the manager receives a MutatedPopulationMessage, which can only occur after the first iteration, actorManager will verify if the current iteration is smaller than the total number of iterations we want, if it's smaller with the message received from the MutateActor it will replace the CurrentPopulation with the population in the MutatedPopulationMessage and will send a PopulationMessage to FitnessActor with the new population which will repeat the cycle described above, finally it increments the generation counter. Eventually when the counter shows that it is the last cycle instead of replacing the CurrentPopulation with the mutated one it will instead send a stop message to all the actors shutting down the program.

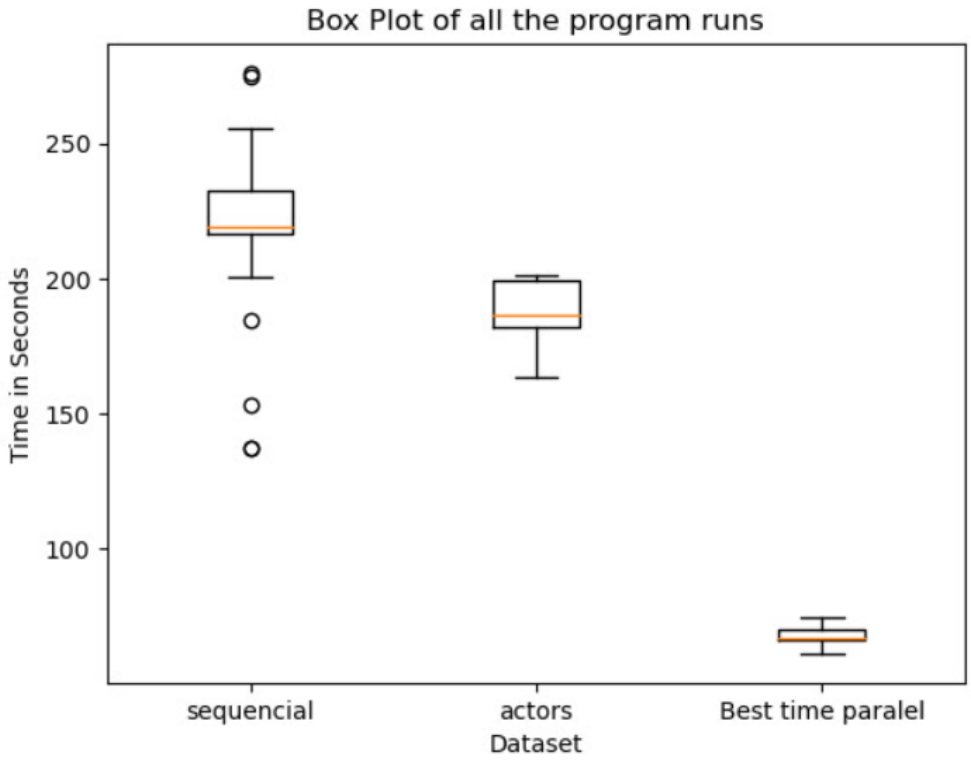
On the Main of the project to create and initialize 30 actor managers to register the execution times I created a for cycle and to prevent the code from creating all 30 at the same time I implemented a flag on ActorManager that notifies the main when it's execution is completed, and only after the execution of the current manager is completed it will create and initialize the next one.

In the following page I will show the results obtained and the compare the actors project with the original sequential project and the fastest parallel version.

As we can see with the values in page 2 even though every actor makes the necessary steps sequentially just like the original there's a performance increase of approximately 13.7% and the data is more consisted, there aren't as many outliers as there are with the original one. The reason why is slightly faster is that because of parallel processing, since different actors may be executing different tasks at same time, in this case like the calculation of the best individual and the print is happening at the same time as the Crossover and because of asynchronous execution, while one actor is idle not doing anything other actors working, which can result in a better utilization of the CPU and lead to performance gains.

Pc specs:

AMD Ryzen 7 3700Uwith Radeon Vega Mobile Gfx / 8gb RAM memory



Total Sequential time in seconds: 109.410882065

Total Actors time in Minutes: 94.42536577166668

Best Parallel total time in Minutes: 33.94290766333334

Approximate speed up between the original sequential and the actors: 13.696550115033858%