Ciências ULisboa

Assignment #3 Task Parallelism in
the Genetic Algorithm for the Knapsack problem
Programação Paralela e Concorrente (2023/2024)

Author: Daniel Lopes – Fc56357

1. The description of the overall architecture.

I implemented the KnapsackGA problem using the ActorModel in Java, as requested in the assignment. Utilizing a structure provided by the course professor, I designed an architecture where each Actor, an abstract class extending Thread, manages specific variables for the problem. I structured the KnapsackGA architecture using these Actors:

- KnapsackGAManager: Responsible for initializing the population through KnapsackGAActorInitializer or evaluating the current Population using KnapsackGAActorMeasureFitness. It controls the number of generations and terminates all active Actors upon completion.
- KnapsackGAActorInitializer: Creates a new population with random individuals, sending each to its supervisor, KnapsackGAActorMeasureFitness. After initializing all individuals, it triggers a self-termination Message.
- KnapsackGAActorMeasureFitness: Calculates the fitness of received individuals and sends the complete array to its supervisor, KnapsackGAActorBestIndividual.
- KnapsackGAActorBestIndividual: Receives the entire Population and prints the best individual found. Send it via Message and the Population array, to KnapsackGAActorCrossOver.
- KnapsackGAActorCrossOver: Identifies parent individuals from the population array and performs crossover operations. Upon completion, it sends the resulting individual to KnapsackGAActorMutate.
- KnapsackGAActorMutate: Receives individuals, performs mutation, and returns them to KnapsackGAManager to finalize the generation process.

A total of 6 Threads (Actors) + Main Thread were used.
The entire implementation architecture is explained and depicted in the diagram found in the file architecture.png located in the main directory.

2. The rationale for choosing that particular architecture.

I designed this architecture by dividing each stage into separate Actors. As a result, the workflow resembles a pipeline, with individual entities or arrays passing through dedicated Actors to execute particular tasks. These tasks are performed concurrently alongside other steps. However, Actors such as KnapsackGAActorBestIndividual and KnapsackGAActorCrossOver operate sequentially since they depend on the complete Population array. Consequently, it's crucial to await the processing of all individuals before these Actors can execute their respective tasks. Moreover, this architecture was meticulously crafted to minimize message passing frequency, aiming for straightforward message exchanges to mitigate overhead on the Actors.

3. An explanation of how the necessary synchrotion as implemented.

No synchronization was required as there is no shared memory between Actors (Threads). All data is transmitted between Actors through an asynchronous communication system using Messages. Each actor contains an Address where it receives Messages. The structure utilized is a stack that orders elements in a FIFO (First-In-First-Out) manner. This stack is a ConcurrentLinkedQueue, which inherently ensures thread safety, thereby guaranteeing synchronization when threads access it.

4. A brief benchmark of this version against the sequential and the first assignment, and its explanation.

CPU used: AMD Ryzen 7 3700X 8-Core Processor,16 Logic Processors, 3.6 GHz of Frequency.
Upon analyzing the execution times of this implementation using the ActorModel, I observed an average slowdown of 152% compared to the sequential execution of the problem. Whereas the sequential version took 86.6 seconds, the ActorModel version took 217.7 seconds. This slowdown might have occurred due to the fact that Messages are queued and processed sequentially by each Actor. This process of sending, receiving, and processing messages could introduce additional overhead compared to the sequential version or even direct synchronization between threads, especially when a large number of Individuals are sent separately. This high frequency of Messages and their processing overhead may contribute to the observed slowdown in the ActorModel implementation.
In the "data" folder, you will find boxplots comparing the first assignment with the ActorModel version (execution_time_comparation.WEBP), a graph showcasing all performance improvements in comparison to the sequential version (performance_improvement.WEBP), as well as the Python file and tables created and utilized to generate these data and graphs presented in this report.