Inês Marques 62839

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

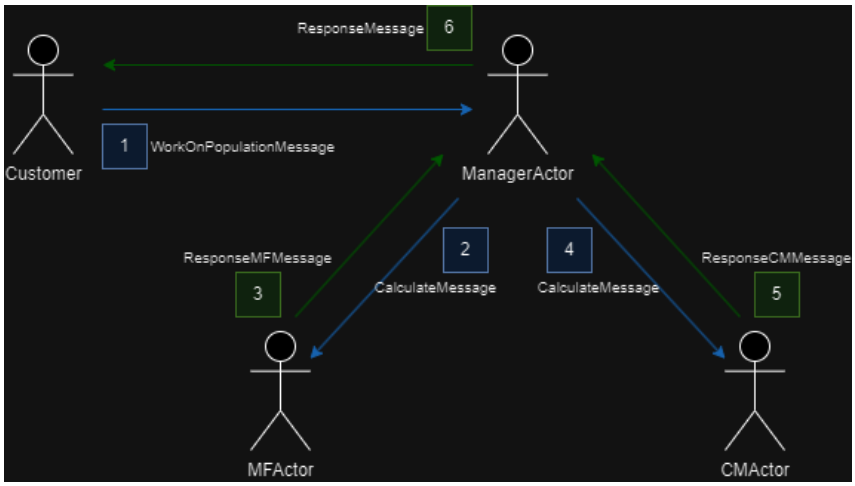
A arquitetura do sistema do Knapsack GA com Actor Model segue uma abordagem paralela, onde Actors independentes colaboram para realizar as diferentes etapas do algoritmo. A comunicação entre os Actors é baseada em mensagens, evitando a partilha direta de memória.

Número de processadores lógicos = 12

CPU Model = Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz

IDE = Eclipse

Arquitecture



1. Inicialização: Inicia-se com a criação de um ManagerActor e um Customer no Main. O Customer envia uma mensagem BootstrapMessage para iniciar o processo. O Customer gera a primeira população e envia uma mensagem WorkOnPopulationMessage para o ManagerActor.

2. Pedido do Cálculo de Fitness (MFActors): O ManagerActor cria 12 (*available processors*) MFActors para calcularem o fitness da população, dividindo-a em chunks. O ManagerActor distribui chunks da população entre os MFActors, enviando as mensagens CalculateMessage.

3. Resposta do Cálculo de Fitness: Cada MFActor calcula o fitness de cada indivíduo e envia uma mensagem ResponseMFMessage ao ManagerActor com sua parte da população. O ManagerActor quando recebe a resposta, envia uma mensagem SystemKillMessage para terminar a execução desse MFActor.

4. Pedido do Cálculo de Crossover/Mutate (CMActors): Após aguardar todas as respostas dos MFActors, o ManagerActor calcula o melhor indivíduo da geração. O ManagerActor cria 12 (*available processors*) CMActors para realizar o crossover e a mutação da população, dividindo-a em chunks. O ManagerActor distribui chunks da população entre os CMActors, enviando as mensagens CalculateMessage.

5. Resposta do Cálculo de Crossover/Mutate: Cada CMActor realiza o crossover e mutação e envia uma mensagem ResponseCMMessage ao ManagerActor com sua parte da população. O ManagerActor quando recebe a resposta, envia uma mensagem SystemKillMessage para terminar a execução desse CMActor.

6. Resposta ao Customer: Após aguardar todas as respostas dos CMActors, o ManagerActor envia uma resposta final ao Customer com a nova população.

7. Finalização: O Customer incrementa o número da geração e repete o ciclo até atingir o número desejado de gerações. Quando todas as gerações são concluídas, o Customer envia uma mensagem SystemKillMessage para o ManagerActor e para ele mesmo, para encerrar o sistema.

Justifications and Synchronization Implementation

O ManagerActor supervisiona MFActors e CMActors, responsáveis por cálculos paralelizáveis de fitness e crossover/mutação, respetivamente. O Customer assume o papel de solicitar gerações, contribuindo para a organização do sistema.

Adicionalmente, considerei desnecessário criar um Actor exclusivamente para o cálculo do melhor indivíduo da população. Optei por deixar esta responsabilidade ao ManagerActor, evitando a criação de um ator adicional e otimizando a eficiência do sistema. Considerei também a opção de os CMActors enviarem diretamente as respostas ao Customer, otimizando potencialmente o tempo de execução. No entanto, optei por manter a arquitetura atual em prol da organização e lógica do sistema.

A implementação da sincronização no sistema ocorre através de trocas de mensagens entre os Actors. O ManagerActor coordena a execução, aguardando as respostas dos Actores (MFActors e CMActors), antes de avançar para as etapas seguintes do algoritmo. Cada Actor opera em cópias independentes dos dados, evitando compartilhamento direto de memória (com o uso do *clone()*).

Benchmark

Média Sequencial: 93,4313 segs

Média Paralelo: 69,6493 segs

Média Actor Model: 277.8547 segs

A criação e término de Actors podem introduzir um aumento no tempo de execução. A troca de mensagens (envio, receção e processo) entre Actors, pode impor um acréscimo no tempo. A clonagem extensiva de dados, realizada para evitar memória compartilhada, pode gerar um overhead significativo. Além disso, a natureza do Knapsack problem, que não é altamente paralelizável devido à interdependência das soluções parciais, que limita as oportunidades de paralelismo eficiente (CM precisa esperar a conclusão do MF antes de iniciar sua execução). Estes fatores combinados contribuem para o aumento do tempo de execução na abordagem com Actors de 2.97x em relação à versão sequencial e 3.99x em relação à paralela com threads. Em suma, o modelo de Actors pode ser mais benéfico em problemas mais complexos e altamente paralelizáveis, onde as tarefas são independentes.

O uso de Co-routines e Channels poderia ter sido mais interessante no sentido em que oferecem uma abordagem mais leve e eficiente.