Genetic Algorithm for the Knapsack Problem

Introdução

Neste relatório, abordarei a necessidade de paralelizar um pequeno programa baseado em Genetic Algorithm for the Knapsack Problem. A paralelização desse programa oferece a oportunidade de acelerar significativamente o processo de busca de soluções para o Problema da Mochila e a sua eficiência.

Ao longo deste relatório, explorarei desafios envolvidos na tarefa, as técnicas e estratégias utilizadas para alcançar esse objetivo. Além disso, observamos os resultados obtidos após a paralelização, destacando as melhorias no desempenho.

Objetivos

- Discutir os desafios envolvidos na tarefa de paralelização.
- Apresentar as técnicas e estratégias utilizadas para alcançar a paralelização.
- Avaliar os resultados obtidos após a paralelização, destacando melhorias no desempenho

Desafios da paralelização

1. Dividir certos pedaços de código entre as threads de forma eficiente, equitativa e sem prejudicar a sua comunicação evitando um overhead.
2. Garantir que as tarefas tenham uma alocação adequada e equilibrada de carga de modo aumentar o potencial de aceleração.
3. Fazer uso da sincronização entre processos para garantir que a sua execução seja livre de problemas de concorrência.

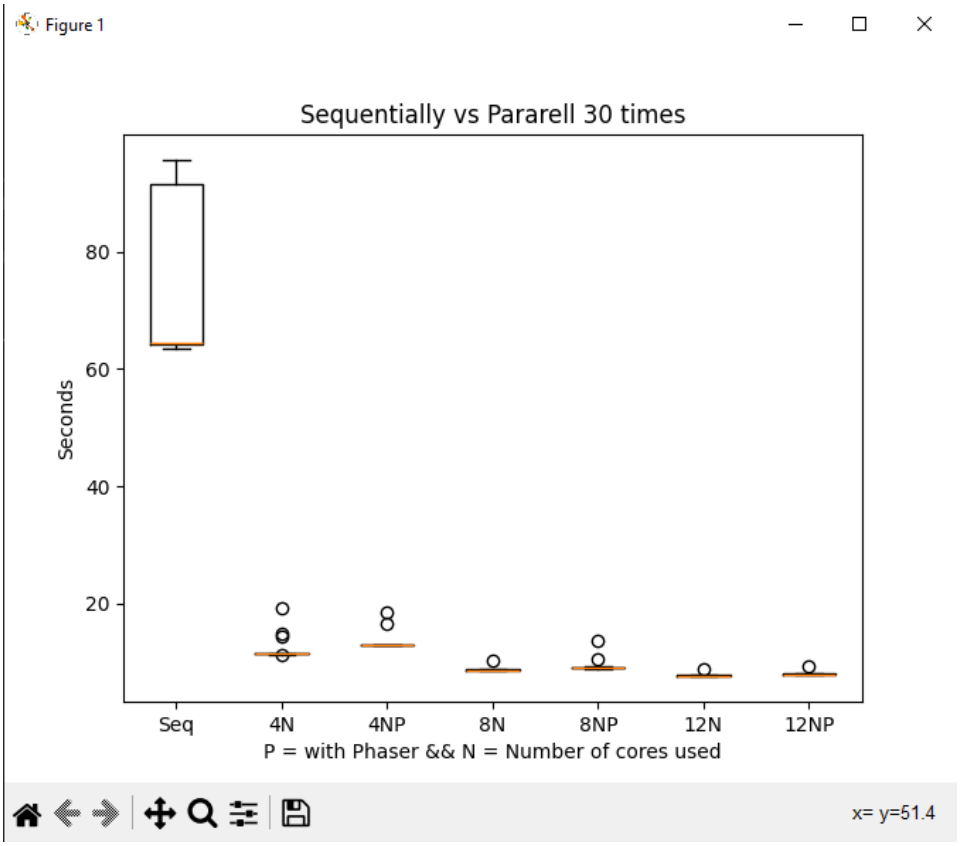
Técnicas e Estratégias de Paralelização

- Usei a técnica da paralelização com threads como aprendemos na primeira semana de aulas, onde é uma técnica poderosa, mas requer cuidados com a sincronização e gerenciamento de recursos compartilhados para evitar problemas de concorrência.
- Fiz uso do synchronized para lidar com concorrência e garantir que várias threads acessem e modifiquem recursos compartilhados de maneira segura como no caso do método `bestOfPopulation()` onde cada thread vai atualizar best que é um array de tamanho 1 (obs : usei um array porque é um objeto para ser utilizado como parâmetro do synchronized)

- Usei a classe ThreadLocalRandom visto que o objetivo era tornar o código em um ambiente multithread e o Random por si só não garante a segurança das threads e não evita problemas de concorrência, notei uma das maiores diferenças que fez na redução do tempo de execução.
- Recorri a classe do Phaser que é mais um mecanismo avançado para gerenciar a concorrência permitindo que as threads cooperem em etapas específicas de uma tarefa, aguardando umas pelas outras antes de avançar para a próxima etapa.
- No método `tournament(int tournamentSize, Random r)` não utilizei a técnica da paralelização porque o tournamentSize é muito pequeno (3) e a sobrecarga de criar e gerenciar múltiplas threads para paralelização pode superar qualquer benefício de desempenho. Nesse caso assim, a execução sequencial desse método foi mais eficiente.

Resultados e Discussão

Nesta seção, apresentarei os resultados obtidos antes e após a paralelização do programa, sendo que eu neste trabalho criei duas classes onde numa eu uso todos os métodos que referi a cima e noutra eu optei por deixar de fora a técnica do Phaser , no boxplot amostrado em baixo refiro que NP = (N quer dizer numero de cores usados e P quer dizer classe com Phaser), criei um ficheiro em python que cria um boxplot dos segundos que cada leva a correr, cada um foi corrido 30 vezes como é pedido.



Neste boxplot conseguimos observar que há enormes diferenças significativas após a paralelização , sendo o tempo médio da Sequencial de 66 segundos e sendo o tempo médio da paralelização de 4 cores de 12 segundos temos um speed up de 66/12 = 5,5 o que significa que a versão paralela do programa é 5,5 vezes mais rápida do que a versão sequencial. Isso indica uma melhoria significativa no desempenho ao usar paralelismo. De ocupação de 5,5/4 = 1,375 , o que indica que, em média, 137,5 % dos núcleos da máquina estão sendo utilizados durante a execução do programa, para 8 cores temos uma média de 9 segundos o que nos leva um speed up de 66/9 = 7.3 e de ocupação de 7.3/8 =0.916 ou seja o programa executa 7,3 vezes mais rápido e 91,6% dos núcleos da maquina estão a ser utilizados. Com 12 cores temos uma media de 7.9 segundos o que leva um speed up = 66/7.9 = 8,35 e com uma ocupação de 8.35/12 = 0.69 o que significa que 69% das threads estão a ser utilizadas e o programa corre 8.35 vezes mais rápido que a versão sequencial. Ou seja o speed up está muito positivo e a paralelização também está otimizada sendo o ideal ter uma ocupação perto de 1 .

Com isto a paralelização beneficiou muito a redução do tempo de execução, ganhou eficiência e aceleração em comparação com a versão não paralelizada.

Ao avaliar os resultados da paralelização, observamos que o desempenho do programa demonstrou uma escalabilidade ligeiramente melhorada em relação às versões paralelizadas e uma melhoria dramática em relação à versão não paralelizada. Com o aumento das threads ou de cores (núcleos) para 8 notamos um ligeiro aumento que significa que conseguiu lidar de forma mais eficiente com um maior número de threads, demonstrando uma resposta positiva à escalabilidade. No entanto, após a adição de cores para 12, a escalabilidade começou a estagnar o tempo de execução, indicando que a adição de mais threads não proporcionaria ganhos significativos.

Ainda usei um teste estatístico do Kruskal-Wallis H Test (ANOVA: for more than two independent samples) no ficheiro python e concluí que tem diferenças significativas.

Conclusão

Este relatório abordou a necessidade e a implementação da paralelização de um pequeno programa. Durante a implementação, explorei os desafios enfrentados, as técnicas e estratégias ensinadas nas aulas e avaliei os resultados obtidos.

A paralelização do programa demonstrou claramente um impacto positivo no desempenho. Os resultados revelaram uma redução significativa no tempo de execução, com uma aceleração notável em comparação com a versão não paralelizada. O uso da classe Phaser permitiu uma resposta mais eficaz à adição de threads, embora tenha atingido um ponto de saturação, indicando que, após um certo número de threads, os benefícios adicionais são mínimos.

Os desafios enfrentados na paralelização incluíram a alocação equitativa de carga de trabalho, a sincronização entre threads para evitar problemas de concorrência e o

gerenciamento eficiente de recursos compartilhados. A técnica do synchronized foi aplicada de forma eficaz para garantir a exclusividade de acesso a recursos críticos compartilhados entre as threads.

A análise dos resultados do teste de Kruskal-Wallis H Test confirmou que as diferenças observadas eram estatisticamente significativas, validando a eficácia da paralelização.

Este trabalho reforça a importância da paralelização em aplicações de otimização de alto desempenho e seu potencial para acelerar a busca de soluções em problemas complexos.

Aluno/Autor

- 56374 - Vasco Miguel Raimão Maria