Relatório: Paralelização de um Algoritmo, utilizando OpenMP, para a resolução do Problema 0-1 Knapsack

Mateus Felipe de Cássio Ferreira[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR) – Curitiba – PR – Brasil

mfcf17@inf.ufpr.br

Abstract. *Given a set of weights and profits, pick and place these items in a limited-capacity knapsack in order to get the maximum profit. This is the 0-1 Knapsack problem, a decision and resource allocation problem that falls into the class of NP-Hard problems. Since there are no known algorithms that can solve all cases in polynomial time, we intend to optimize a solution using dynamic programming concepts and code parallelization. However, despite the efforts, it was noticed that the optimization of this problem is not trivial and the application of parallelization techniques may not be the best approach for the optimization of the proposed solution.*

Resumo. *Dados um conjunto de pesos e valores, escolha e coloque esses itens em uma mochila de capacidade limitada de forma a obter o valor máximo de ganho sobre a escolha. Esse é o problema 0-1 Knapsack, um problema de decisão e alocação de recursos que encontra-se na classe dos problemas NP-Difícil. Uma vez que não se conhece algoritmos que consigam resolver todos os casos em tempo polinomial, pretende-se otimizar uma solução utilizando conceitos de programação de dinâmica e paralelização de código. No entanto, apesar dos esforços, notou-se que a otimização desse problema não é trivial e a aplicação de técnicas de paralelização podem não ser a melhor abordagem para a otimização da solução proposta.*

1. Introdução

Dados um conjunto de pesos e valores, escolha e coloque esses itens em uma mochila de capacidade limitada de forma a obter o valor máximo de ganho sobre a escolha. Esse é o problema 0-1 *Knapsack* da mochila "ilimitada". Esse é um exemplo de um problema clássico sobre alocação de recursos. É dito que é um problema 0-1 visto que os itens não podem ser quebrados ao meio, assim, eles devem pertencer ou não a mochila (problema de decisão).

Sabe-se que esse problema encontra-se na classe dos problemas NP-Difícil e, assim, não se sabe ainda da existência de um algoritmo para a resolução desse problema para todos os casos em tempo polinomial. Por outro lado, no que tange ao problema de otimização, a sua resolução é pelo menos tão difícil quanto ao problema de decisão, e não há algoritmo polinomial que possa garantir que, dada uma solução, se ela é ótima (WIKIPEDIA).

Assim, o propósito deste trabalho é o de utilizar técnicas de paralelização de código sequencial, utilizando uma interface de programação de aplicativo conhecida

como **OpenMP**, como uma tentativa de otimização de um código sequencial para a resolução do problema *Knapsack*.

1.1. Organização do Texto

A Seção 2 deste trabalho apresenta dois algoritmos que foram avaliados neste experimento, sendo um deles puramente sequencial e o outro com partes paralelas em sua construção.

Por outro lado, a Seção 3 deste trabalho apresenta os materiais e códigos que foram utilizados e desenvolvidos para a confecção do experimento.

A Seção 4, por sua vez, apresenta os resultados e algumas discussões que podem ser levantadas a respeito do experimento.

Por fim, a Seção 5 apresenta a conclusão deste trabalho. Ainda, foi disponibilizado, na Seção 6, um acesso a um diretório no GitHub contendo todo o conteúdo necessário para reprodutibilidade dos experimentos realizados.

2. Algoritmos de Resolução

A estratégia em que foi possível obter, de fato, uma otimização expressiva para o código baseia-se na resolução do problema utilizando uma estratégia de programação dinâmica. Esse tipo de paradigma de programação parte da premissa de resolver um certo problema de otimização por meio da análise de uma sequência de problemas mais simples de serem resolvidos comparando com o problema original.

No que se refere ao problema *Knapsack*, essa estratégia adotada faz o uso de dois laços de repetição, um que vai de 0 até a quantidade de itens presentes na mochila (N) e outro que vai do peso máximo da mochila (M) até o 0. Esse algoritmo consiste em, basicamente, preencher um vetor (dp) com os valores de cada item que foram sendo encontrados de modo a escolher ou não a sua presença na mochila considerando uma certa capacidade Mi. Assim, para cada posição Mi do vetor, é feita uma comparação se vale a pena manter o valor do ganho na posição atual (dp[w]) ou se é preciso escolher uma combinação de itens diferentes (dp[w - wt[i]] + val[i]) que, somado o ganho escolhendo o item que está sendo verificado no momento (val[i]), resulte em uma soma maior do que o ganho que está sendo buscado na posição atual do vetor (dp[w]). Nesse sentido, entende-se que para a capacidade máxima da mochila M, o valor do ganho máximo que podemos obter está na última posição do vetor (dp[w]).

Nesse sentido, ressalta-se que o fundamento do algoritmo permanece o mesmo não só para a execução serial, mas também para a versão paralelizada. No caso da última, existe a utilização de um vetor auxiliar que armazenará sempre os valores da iteração passada, uma vez que a utilização de várias *threads* executando em paralelo poderiam modificar o valor do vetor compartilhado (dp), o que foge do conceito do algoritmo.

Por fim, a complexidade em termos de tempo para a versão sequencial desse algoritmo é O(N*M), enquanto o espaço auxiliar de memória que é necessário concentra-se em O(M). Por outro lado, quando se trata a respeito da versão paralelizada, a complexidade de espaço auxiliar necessário para sua execução é de O(N*M), visto que

nessa última abordagem é fundamental a utilização de um vetor auxiliar, como citado acima.

3. Materiais e Métodos

Nesta seção serão apresentados os materiais utilizados ao longo do desenvolvimento deste trabalho, bem como os métodos empregados para a obtenção dos resultados dos experimentos de paralelização. É importante ressaltar que todo o trabalho foi desenvolvido em linguagem C e utilizou a API **OpenMP** dentro do código desenvolvido.

3.1. Especificações do Sistema e Ambiente de Experimento

Todos os experimentos aconteceram em um ambiente Linux, utilizando o sistema operacional Linux Mint na versão 20.2. O *kernel* do sistema está na versão 5.4.0-104-generic. O compilador utilizado para permitir a execução dos algoritmos foi o gcc na versão 9.4. As *flags* de compilação utilizadas foram: **-O3** (para permitir o máximo de otimização possível) e **-fopenmp** (para permitir a utilização da biblioteca OpenMP). A Figura 1 apresenta as principais configurações do processador utilizado, bem como a capacidade das *caches* presentes. Para a extração dessas informações, utilizou-se o *software* CPU-Z.

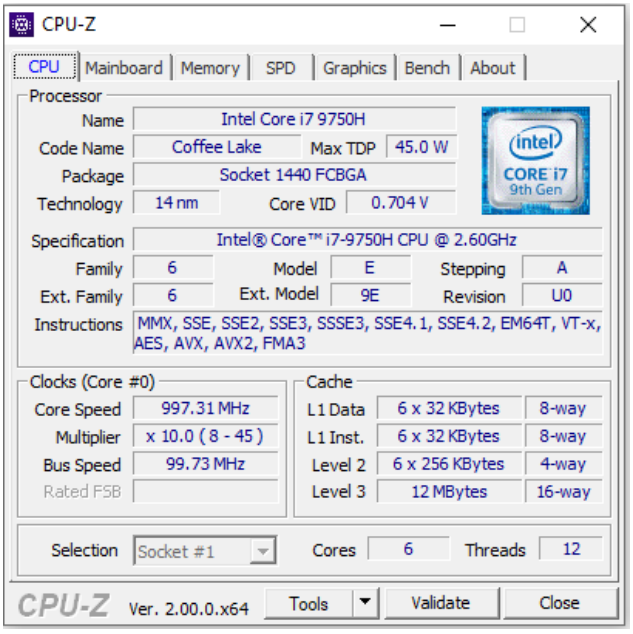

Figura 1 - Especificações Técnicas da CPU e *caches* presentes no Sistema de Experimentos.

3.2. Execução dos Experimentos

A execução do algoritmo foi feita de forma automatizada, utilizando um *script* para o *Shell* para executar todos os experimentos. Para cada um dos experimentos, o algoritmo

foi executado 20 vezes, com o objetivo de obter uma aproximação mais exata acerca do real tempo de execução.

Além disso, a quantidade de *threads* utilizadas nos experimentos foram 2, 4, 8 e 12 (visto que o sistema suporta um limite de 12 *threads* em execução simultânea). Por fim, vale ressaltar que para a implementação paralela, preocupou-se com o tamanho da cache L2 para o experimento. Nesse sentido, um dos parâmetros para o algoritmo é o tamanho da cache L2 para o cálculo do valor do *chunk* (ou passo) na execução do laço mais interno do algoritmo paralelo.

3.3. Métricas Avaliadas

As métricas avaliadas neste trabalho baseiam-se, sumariamente, no cálculo do tempo em diferentes regiões do algoritmo. A partir do cálculo desses tempos de execução é que foi possível calcular e estimar outras métricas, como o *speedup* teórico (baseado na Lei de Amdahl), *speedup* real e o cálculo da eficiência do algoritmo.

No que se refere à medida de *speedup*, na computação, essa medida busca mensurar o grau de desempenho de uma determinada versão de algoritmo. Assim, para uma mesma entrada, o *speedup* real é calculado como sendo a razão entre o tempo de execução antiga sobre o novo tempo de execução. Ainda se tratando da medida de *speedup*, uma importante medida é levar em consideração a Lei de Amdahl, que é utilizada para determinar o limite máximo de *speedup* que uma determinada aplicação poderá alcançar, independentemente do número de processadores.

Por outro lado, a medida de eficiência do algoritmo busca mensurar o grau de aproveitamento dos recursos computacionais, e é calculada como sendo a razão entre o desempenho (nesse caso, o *speedup* real) sobre os recursos computacionais disponíveis (nesse caso, o número de *threads*).

3.3. Metodologia do Cálculo do Tempo

A função para cálculo do tempo que é gasto para a resolução do problema, `timestamp()`, utiliza a função `gettimeofday()` da biblioteca sys/time.h e devolve o tempo em segundos. Assim, todos os cálculos presentes neste trabalho estão na ordem de grandeza de segundos.

No algoritmo sequencial, a parte de medição do tempo ocorreu na função `main()`, antes do algoritmo "Knapsack" ser chamado para resolver o problema. Por outro lado, no algoritmo paralelo foi utilizado algumas variáveis globais que são incrementadas conforme determinada uma região paralela e sequencial. Assim, o tempo sequencial relevante para esse algoritmo seria a troca do conteúdo de dois vetores presente dentro da função *Knapsack* da versão paralela. Essa troca, que utiliza a função `memmove()`, é fundamental para que na próxima iteração sobre a inclusão de um item novo, a comparação seja feita com o estado prévio da mochila, e não com o estado atual.

4. Resultados e Discussões

Nesta seção serão apresentados os resultados obtidos nos experimentos realizados. Esses resultados estão sumarizados, basicamente, em duas grandes tabelas. A Tabela 1 apresenta os resultados de tempo obtidos para ambas as versões avaliadas. Essa foi a

tabela base construída para o cálculo das demais métricas descritas na Seção 3.3. O valor de N refere-se ao total de itens avaliados para escolha de colocar na mochila, enquanto que o valor de W apresenta o peso máximo que a mochila pode suportar. Ainda, é apresentado a média dos valores de tempo para cada uma das 20 execuções feitas, bem como o desvio padrão dessas execuções. Por fim, na última parte da tabela é calculado o valor do tempo total médio (como sendo a soma da parte sequencial e a parte paralela) e a porcentagem em que cada parte corresponde no tempo total médio.

A análise dessa tabela, imediatamente, permite concluir que a versão implementada paralelizada tem um desempenho pior que a versão puramente sequencial. Tomando o pior caso como exemplo, nota-se que para uma entrada de 80.000 itens, o algoritmo sequencial consegue chegar na solução em apenas 1,0566 segundos, enquanto que a versão paralela, no melhor caso, consegue encontrar a solução utilizando 4,1963 segundos (considerando a utilização de 2 *threads* e levando em conta o tempo total médio). Além disso, é imediato notar que a utilização de mais *threads* para a resolução deste problema, ao contrário do que se espera, piora ainda mais o desempenho do algoritmo paralelo. Isso é possível observar nas cores presente na média da parte paralela do algoritmo, bem como no tempo total médio, uma vez que para um mesmo valor de N, à medida em que a quantidade de *threads* utilizadas vai aumentando, o tempo para encontrar o resultado da solução também aumenta.

Por outro lado, nota-se que a grande maior parte do tempo de execução da função *Knapsack* na versão paralela é consumido pela região paralelizada, uma vez que tomando a coluna da porcentagem do tempo paralelo, em nenhum experimento constatou-se uma porcentagem inferior a 67% do tempo total médio da execução de todo o algoritmo. Esse fato, aliado às demais constatações anteriores, leva à conclusão que a paralelização do código sequencial, utilizando princípios da programação dinâmica, não é trivial e pode não ser a melhor abordagem para a otimização desse problema computacional. Tendo em vista que o código precisa acessar posições de memória que estão, na maioria das vezes, distantes na memória (baixa localidade espacial), a ocorrência de *cache misses*, aliado ao fato de manter a consistência das linhas de *caches* presentes em memória, não compensa o uso de mais processadores com o intuito de resolver o problema mais rapidamente, para essa abordagem de paralelização deste algoritmo sequencial.

RESULTADOS PARA O ALGORITMO SEQUENCIAL									
N	W	THREADS	PARTE SEQUENCIAL		PARTE PARALELA		TEMPO TOTAL	% TEMPO	% TEMPO
			MÉDIA	DP	MÉDIA	DP	MÉDIO	SEQUENCIAL	PARALELO
20000	100000	1	0,2625	0,0040	-	-	0,2625	100%	-
40000	100000	1	0,5200	0,0068	-	-	0,5200	100%	-
80000	100000	1	1,0566	0,0268	-	-	1,0566	100%	-

RESULTADOS PARA O ALGORITMO PARALELO									
N	W	THREADS	PARTE SEQUENCIAL		PARTE PARALELA		TEMPO TOTAL	% TEMPO	% TEMPO
			MÉDIA	DP	MÉDIA	DP	MÉDIO	SEQUENCIAL	PARALELO
20000	100000	2	0,3420	0,0084	0,7017	0,0174	1,0437	32,8%	67,2%
		4	0,3909	0,0057	0,8344	0,0159	1,2253	31,9%	68,1%
		8	0,4163	0,0038	1,1390	0,0247	1,5553	26,8%	73,2%
		12	0,4265	0,0103	4,5002	0,6397	4,9267	8,7%	91,3%
40000	100000	2	0,6902	0,0121	1,4111	0,0312	2,1013	32,8%	67,2%
		4	0,7803	0,0079	1,6550	0,0198	2,4353	32,0%	68,0%
		8	0,8388	0,0056	2,2986	0,0657	3,1374	26,7%	73,3%
		12	0,8528	0,0144	10,2135	1,2931	11,0663	7,7%	92,3%
80000	100000	2	1,3771	0,0181	2,8192	0,0381	4,1963	32,8%	67,2%
		4	1,5700	0,0111	3,3238	0,0488	4,8939	32,1%	67,9%
		8	1,6780	0,0127	4,5908	0,0909	6,2688	26,8%	73,2%
		12	1,7045	0,0188	18,0107	0,7426	19,7153	8,6%	91,4%

Tabela 1 - Resultados obtidos dos experimentos de tempo para os algoritmos sequencial e paralelo.

Além disso, outros resultados interessantes de serem avaliados dizem respeito às métricas de *speedup* e eficiência do algoritmo paralelo. A Tabela 2 apresenta os resultados obtidos para os experimentos realizados. Nota-se, conforme esperado, que os valores de *speedup* reais para o algoritmo paralelo apresentaram valores inferiores a 1. Esse valor está em conformidade com o esperado, visto que o desempenho do algoritmo paralelo, em termos de tempo de execução, foi pior quando comparado com o tempo sequencial, conforme consta na Tabela 1. O *speedup* considerando a Lei de Amdahl, por outro lado, teve valores superiores a 1, e considerando a hipótese da existência de infinitos processadores, o valor de *speedup* para cada um dos testes realizados é apresentado na Tabela 2.

Por outro lado, no que tange à discussão acerca da escalabilidade do algoritmo, nota-se que o algoritmo paralelo não obteve uma escalabilidade forte, ou seja, tomando um determinado tamanho de itens da mochila a serem avaliados, o algoritmo não conseguiu manter a mesma eficiência (ou próxima) à medida que o número de *threads* ia aumentando. Ainda, em relação à escalabilidade fraca, o algoritmo também não se encaixou nesse conceito, uma vez que ele não conseguiu manter uma mesma eficiência à medida que o tamanho da entrada e o número de processadores aumentavam proporcionalmente. Esse fato pode ser observado nas células destacadas em cinza na Tabela 2. Assim, portanto, o algoritmo paralelo não é escalável.

SPEEDUP E EFICIÊNCIA						
	N	2	4	8	12	INFINITO
SPEEDUP TEÓRICO (baseado na Lei de Amdahl)	20000	1,5064	2,0438	2,7839	6,1467	11,5516
	40000	1,5055	2,0395	2,7860	6,4947	12,9768
	80000	1,5058	2,0383	2,7838	6,1506	11,5663
SPEEDUP REAL	20000	0,2515	0,2142	0,1688	0,0533	-
	40000	0,2475	0,2135	0,1657	0,1657	-
	80000	0,2518	0,2159	0,1685	0,0536	-
EFICIÊNCIA	20000	**0,1257**	0,0536	0,0211	0,0044	-
	40000	0,1237	**0,0534**	0,0207	0,0138	-
	80000	0,1259	0,0540	**0,0211**	0,0045	-

Tabela 2 - Resultados obtidos dos experimentos de speedup e eficiência para o algoritmo paralelo.

5. Conclusão

Nesse sentido, nota-se que o problema de otimização da solução do problema *Knapsack* não é trivial. Em função de acessos à memória que não estão, na maioria das vezes, mapeados numa região próxima, a ocorrência de *cache misses* aliado ao fato do *overhead* em manter a consistências das linhas de cache, em função de um falso compartilhamento entre elas, pode explicar o porquê do baixo desempenho na versão paralela do algoritmo, quando se esperava que fosse o contrário. Assim, para o problema do *Knapsack*, utilizando uma versão que faz uso da programação dinâmica, descobriu-se que a versão sequencial é a melhor escolha para a resolução do problema em um curto espaço de tempo.

6. Reprodutibilidade

Com o intuito de permitir a reprodutibilidade dos experimentos, todos os códigos e os dados de entrada utilizados para a construção deste trabalho estão disponíveis em (https://github.com/mateus-fecassio/_UFPR_-ParalelaKnapsack).

Referências

WIKIPEDIA, Knapsack problem. 2022. Disponível em: <https://en.wikipedia.org/wiki/Knapsack_problem>. Acesso em: 29 mar. 2022.