Relatório: Paralelização de um Algoritmo, utilizando OpenMPI, 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 values, pick and place these items in a limited-capacity knapsack in order to get the maximum amount of gain over the pick. 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, even when it involves other machines.*

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, mesmo quando envolva outras máquinas.*

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 **OpenMPI**, 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+1) e outro que vai de 0 até o peso máximo da mochila (MAXIMUM_CAPACITY+1). Esse algoritmo consiste em, basicamente, preencher uma matriz (V) 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 j. Assim, para cada posição V[i][j], é feita uma comparação se vale a pena manter o valor do ganho na posição anterior (V[i-1][j]) ou se podemos escolher uma combinação de itens diferentes (val[i-1] + value, onde value equivale a V[i-1][j - wt[i-1]]) que, somado o ganho escolhendo o item que está sendo verificado no momento, resulte em uma soma maior do que o ganho que está sendo buscado na posição atual da matriz. Nesse sentido, entende-se que para a capacidade máxima da mochila, o valor do ganho máximo que podemos obter está na última posição da matriz (V[n][MAXIMUM_CAPACITY]).

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 uma adaptação que se fez necessária para a execução paralelizável utilizado o OpenMPI. Nesse caso, a abordagem utilizada consiste em atribuir um *chunk,* ou seja, um pedaço da matriz para cada processador. Esse *chunk* é definido como sendo (MAXIMUM_CAPACITY+1)/numproc, onde numproc é o número de processadores que foram locados. Ao final da iteração mais interna, que percorre a coluna da matriz

que está sendo trabalhada, é feita uma operação de MPI_Allgather para que todas as matrizes V de cada processador tenham o conteúdo que foram calculadas individualmente por cada um dos outros processadores. A Figura 1 ilustra o funcionamento da função utilizada. Por outro lado, uma vez que a operação de (MAXIMUM_CAPACITY+1)/numproc pode não gerar um número divisível para o valor de *chunk,* foi preciso tratar o evento em que sobra um restante da matriz a ser calculada. Isso foi estipulado ser de responsabilidade do processador de *rank* igual a 0. Após terminar essa computação, o processador 0 realiza uma operação de MPI_Bcast para todos os outros processadores do restante da matriz que foi calculada.

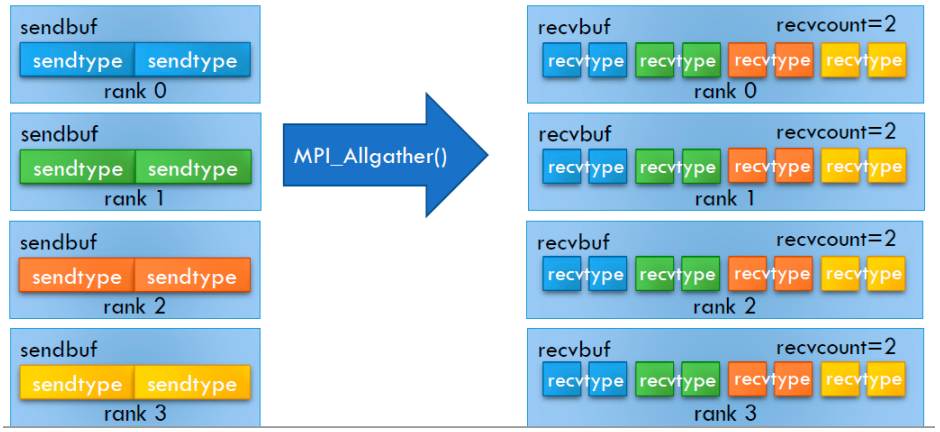

Figura 1 - Funcionamento da função MPI_Allgather.

Por fim, a complexidade em termos de tempo e espaço para a versão sequencial desse algoritmo é O(n*MAXIMUM_CAPACITY). Essas complexidades estão fortemente relacionadas com a utilização da matriz V nessa abordagem da programação dinâmica.

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 biblioteca **OpenMPI** 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. A única *flag* de compilação utilizada foi **-O3** (para permitir o máximo de otimização possível). 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.

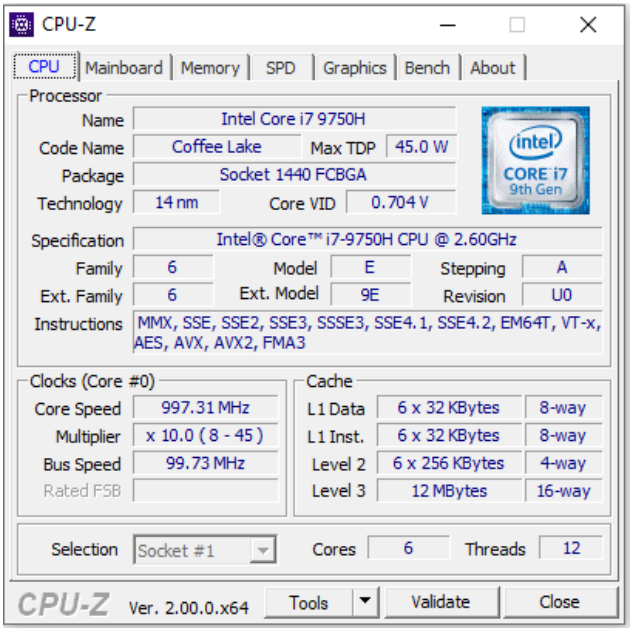

Figura 2 - Especificações Técnicas da CPU e *caches* presentes no Sistema de Experimentos.

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 processadores utilizados nos experimentos foram 1 2 3 4 5 6. Vale ressaltar que os experimentos foram realizados em uma máquina apenas, então, a troca de mensagens entre todos os processadores envolvidos no experimento ocorreu via memória.

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 ciência da 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 processadores).

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 em dois momentos: no primeiro momento, na função main(), antes da função Knapsack() ser chamada para resolver o problema. Isso nos dá, assim, o valor do tempo gasto pela função Knpasack para calcular o valor máximo da mochila. Além disso, como verificado, esse algoritmo pode ser paralelizado, então, essa medição reflete a porção paralelizável dentro do tempo total de execução de todo o algoritmo. Em um segundo momento, foi calculado o tempo total da execução do algoritmo sequencial como um todo.

Por outro lado, no algoritmo paralelo foi feito uma medição de tempo total de execução, com o objetivo de ser comparada com o tempo de execução total do algoritmo sequencial. É importante ressaltar que apenas o processador com *rank* igual a 0 que faz a contagem do tempo de execução de todo o algoritmo paralelo.

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 primeia tabela é calculado a fração (ou seja, a porcentagem) em que a parte que é possível paralelizar (chamada, nesse caso, de “TEMPO DA PARTE PARALELIZÁVEL”) representa em relação ao total do tempo gasto para o algoritmo como um todo executar. Nota-se, nesse caso, que considerando que a função Knapsack pode ser totalmente paralelizável, ela representa uma expressível fração do tempo que é possível otimizar durante a execução do algoritmo, sendo que a parte puramente sequencial está relacionada, basicamente, com a leitura da entrada.

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 12.000 itens, o algoritmo sequencial consegue chegar na solução em apenas 1,0823 segundos, enquanto que a versão paralela, no melhor caso, consegue encontrar a solução

utilizando 1,2223 segundos (considerando a utilização de 1 *processador* e levando em conta o tempo total médio). Se considerarmos 2 processadores, o tempo sobe para 1,8299 segundos. Além disso, é imediato notar que a utilização de mais processadores para a resolução deste problema, ao contrário do que se espera, piora ainda mais o desempenho do algoritmo paralelo. Isso é possível observar na tabela verde da Tabela 1, uma vez que para um mesmo valor de N, à medida em que a quantidade de processadores utilizados vai aumentando, o tempo para encontrar o resultado da solução também aumenta.

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. Uma vez que está sendo comparado o tempo de execução total do algoritmo sequencial com o tempo de execução total do algoritmo paralelo, uma hipótese que pode explicar o porquê deste último algoritmo estar demorando mais para executar está relacionada com a utilização de operações iniciais que necessitam transmitir informações coletadas pelo processador de *rank* igual a 0, o que pode estar influenciando no tempo de execução total do algoritmo e podem ser custosas de serem concluídas (uma vez que, no pior caso, precisamos transmitir dois vetores de 12.000 elementos para cada um dos 6 processadores alocados para resolução do problema). Além disso, operações que são feitas dentro da função `Knapsack()`, como o `MPI_Allgather` e `MPI_Bcas`, ainda mais dentro de laços de repetição, podem estar influenciando na piora do desempenho da abordagem paralela.

RESULTADOS PARA O ALGORITMO SEQUENCIAL									
N	W	PROCS	TEMPO DA PARTE PURAMENTE SEQUENCIAL	TEMPO DA PARTE PARALELIZÁVEL		TEMPO TOTAL MÉDIO		% TEMPO SEQUENCIAL	% TEMPO PARALELIZÁVEL
			MÉDIA	MÉDIA	DP	MÉDIA	DP		
100	12000	1	0,0002	0,0101	0,0008	0,0103	0,0008	1,8%	98,2%
200	12000	1	0,0002	0,0188	0,0004	0,0190	0,0005	1,1%	98,9%
400	12000	1	0,0003	0,0374	0,0010	0,0376	0,0011	0,7%	99,3%
800	12000	1	0,0003	0,0730	0,0009	0,0733	0,0009	0,4%	99,6%
1600	12000	1	0,0005	0,1464	0,0056	0,1469	0,0056	0,4%	99,6%
3200	12000	1	0,0010	0,2866	0,0054	0,2876	0,0054	0,3%	99,7%
6400	12000	1	0,0016	0,5711	0,0109	0,5727	0,0108	0,3%	99,7%
12000	12000	1	0,0028	1,0795	0,0242	1,0823	0,0244	0,3%	99,7%

RESULTADOS PARA O ALGORITMO PARALELO				
N	W	PROCS	TEMPO TOTAL MÉDIO	
			MÉDIA	DP
1600	12000	1	0,1630	0,0036
		2	0,1652	0,0070
		3	0,2008	0,0060
		4	0,1993	0,0080
		5	0,2665	0,0191
		6	0,2769	0,0174
3200	12000	1	0,3288	0,0163
		2	0,3470	0,0157
		3	0,4358	0,0212
		4	0,4320	0,0217
		5	0,5629	0,0294
		6	1,4287	0,0373
6400	12000	1	0,6692	0,0255
		2	0,8178	0,0212
		3	1,0575	0,0398
		4	1,2570	0,0375
		5	1,3044	0,0982
		6	1,4287	0,0373
12000	12000	1	1,2223	0,0425
		2	1,8299	0,0364
		3	2,6090	0,0432
		4	2,2607	0,0511
		5	2,7206	0,0717
		6	3,4549	0,0628

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. Portanto, o algoritmo paralelo não foi escalável.

	***SPEEDUP* E EFICIÊNCIA**							
	N	**1**	**2**	**3**	**4**	**5**	**6**	**INFINITO**
SPEEDUP TEÓRICO (baseado na Lei de Amdahl)	1600	1,0000	1,9925	2,9777	3,9556	4,9262	5,8898	267,1231
	3200	1,0000	1,9932	2,9795	3,9592	4,9322	5,8987	291,0454
	6400	1,0000	1,9945	2,9835	3,9671	4,9453	5,9182	361,5439
	12000	1,0000	1,9948	2,9844	3,9690	4,9484	5,9228	383,5202
SPEEDUP REAL	1600	0,9014	0,8894	0,7315	0,7370	0,5512	0,5306	-
	3200	0,8748	0,8290	0,6601	0,6657	0,5110	0,2013	-
	6400	0,8558	0,7003	0,5416	0,4556	0,4391	0,4009	-
	12000	0,8854	0,5914	0,4148	0,4787	0,3978	0,3133	-
EFICIÊNCIA	1600	**0,9014**	0,4447	0,2438	0,1843	0,1102	0,0884	-
	3200	0,8748	**0,4145**	0,2200	0,1664	0,1022	0,0336	
	6400	0,8558	0,3502	**0,1805**	0,1139	0,0878	0,0668	-
	12000	0,8854	0,2957	0,1383	**0,1197**	0,0796	0,0522	-

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 da utilização de operações iniciais que necessitam transmitir informações coletadas por um processador aos demais, principamente quando essas funções estão dentro de laços de repetição, essa abordagem pode estar influenciando no tempo de execução total do algoritmo por serem custosas de serem concluídas e poderia 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: 28 abr. 2022.