Trabalho MPI Programação Paralela

Pietro Polinari Cavassin

1) O algoritmo.

A parte principal do algoritmo da mochila sequencial é a mostrada na imagem 1.

```
for(i = 0; i < n; i++) {
    for(j = 1; j <= MAXIMUM_CAPACITY; j++) {
        if(wt[i] <= j) { // could put item in knapsack
            int previous_value = V[1 + i - 1][j];
            int replace_items = val[i] + V[1 + i - 1][j - wt[i]];
            V[1 + i][j]= max(previous_value, replace_items);
        }
        else {
            V[1 + i][j] = V[1 + i - 1][j];
        }
    }
}
```

Figura 1: Núcleo do algoritmo da mochila sequencial

Esse algoritmo utiliza programação dinâmica para guardar soluções intermediárias do problema e ir resolvendo-as progressivamente até chegar na solução final.

Nele, é criada uma estrutura da matriz, onde a altura é o número de itens candidatos a entrarem na mochila e a largura é a capacidade total da mochila. Se a matriz é V, um item $V_{i,j}$ é a resposta do problema da mochila para os i primeiros itens e com uma mochila de capacidade j.

Para cada $V_{i,j}$, o algoritmo testa o que resulta em um valor maior: manter os mesmos itens da solução ótima com i-1 itens ($V_{i-1,j}$) ou, se puder ser feito, adicionar o valor do item i à solução ótima para uma mochila com capacidade j – peso(i), o que é equivalente a (peso(i) + $V_{i-1,\ j-peso(i)}$).

Ao final do algoritmo, a posição correspondente a todos os itens e à capacidade máxima da mochila será o valor ótimo.

2) A paralelização

A estratégia foi paralelizar o laço interno do algoritmo. O cálculo de uma posição da matriz depende apenas dos dados da linha anterior e das colunas anteriores a ela. Dessa forma, os cálculos de uma linha são independentes entre si, podendo ser executados em paralelo.

Para fazer essa paralelização, sendo num_threads o número de threads em execução, a linha atual da matriz é quebrada em num_threads vetores (`int *array_partition`), um para cada thread. Então, cada thread calcula sua porção correspondente da linha original da matriz e armazena no vetor `array_partition`. Ao final do processamento da linha, todas as threads compartilham seus trechos do vetor com todas as outras threads através da função `MPI_Allgather()`.

Dessa forma, todas as threads recebem um buffer com todos os cálculos da linha da matriz. O buffer pode ter mais colunas que a matriz, pois nem sempre é possível dividir uma linha igualmente entre todas as threads. Esse buffer então é copiado para a matriz na linha correspondente.

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int send_count = ceil((double)MAXIMUM_CAPACITY/(double)size);
int *array_partition = malloc(send_count*sizeof(int));
int *recvbuf = malloc(send_count*size*sizeof(int));

for(i = 0; i < n; i++) {
    int V_j;
    for (j = 0, V_j = rank*send_count + 1; j < send_count && V_j <= MAXIMUM_CAPACITY; ++j, ++V_j){
        if(wt[i] <= V_j) { // could put item in knapsack
            int previous_value = V[1 + i - 1][V_j];
            int replace_items = val[i] + V[1 + i - 1][V_j - wt[i]];
            array_partition[j]= max(previous_value, replace_items);
        }
        else {
            array_partition[j] = V[1 + i - 1][V_j];
        }
    }
    MPI_Allgather(array_partition, send_count, MPI_INT, recvbuf, send_count, MPI_INT, MPI_COMM_WORLD);
    for (j = 0; j < MAXIMUM_CAPACITY; ++j) V[i + 1][j+1] = recvbuf[j];
}
```

Figura 2: Núcleo do algoritmo paralelizado

3) Metodologia dos experimentos

Para os experimentos, foram geradas três entradas do problema da mochila, com 2.5, 5 e 10 milhões de itens. Em todas as entradas, capacidade da mochila de 50 e as massas dos itens eram números inteiros aleatórios variando de 1 a 50.

O computador foi então reiniciado para garantir que todos os processos de usuário fossem encerrados e as CPUs e a RAM estivessem livres para a execução do algoritmo.

Depois, para cada uma das entradas, o algoritmo foi rodado 20 vezes e os tempos foram contabilizados com a função `time`, do UNIX.

Informações da máquina onde foram realizados os testes:

- SO: Linux Mint 20.1 (Ulyssa)
- Kernel: 5.4.0-107-generic
- Compilador: mpicc (gcc 9.4.0)
- Flags de compilação: -lm para a biblioteca matemática.
- Processador: Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 4 núcleos, 2 threads por núcleo

4) Cálculo de tempo sequencial:

Tempo sequencial: Para o cálculo de tempo sequencial do algoritmo, foi calculado o tempo para leitura e broadcast das variáveis, mais o tempo de um laço interno do for e do `mpi_allgather` multiplicados por n, pois no mínimo serão feitos n vezes. O algoritmo foi executado com 2 threads. O tempo foi posteriormente dividido pelo tempo total de execução do algoritmo. O resultado para uma entrada foi de (2.51+0.31)s/3.78s = 75%. A razão mais provável para chegarmos nesse resultado é que, para cada item, temos que fazer mandar os vetores auxiliares entre os processos usando o comando AllGather, e isso é muito custoso.

5. Lei de Amdahl

Aplicando a Lei de Amdahl com um β = 75% e considerando p como o número de processadores, temos os seguintes cálculos de speedup teórico:

$$S(p)=\frac{1}{\beta+\frac{1-\beta}{p}}=\frac{1}{0.75+\frac{0.25}{p}}$$

- Para p = 2, temos

$$S(p)=\frac{1}{0.75+\frac{0.25}{2}}=1.14$$

- Para p = 4, temos

$$S(p)=\frac{1}{0.75+\frac{0.25}{4}}=1.23$$

- Para p = 8, temos

$$S(p)=\frac{1}{0.75+\frac{0.25}{8}}=1.28$$

- Para $p\to\infty$, temos

$$\lim_{p\to\infty} S(p)=\frac{1}{0.75+0}=1.33$$

6. Speedups e eficiência

Foram calculados os tempos de execução com 1, 2 e 4 CPUs com entradas de 2.5, 5 e 10 milhões de itens na mochila 20 vezes. Os resultados de média e desvio padrão de tempo do algoritmo são mostrados na tabela abaixo.

Tabela 1: Médias e desvio padrão de execuções de acordo com tamanho da entrada e número de CPUs

	1 CPU	2 CPUs	4 CPUs
N = 2.5 Milhões	1.00 +- .01s	1.92 +- .01s	2.63 +- .04s
N = 5 Milhões	1.98 +-. 01s	3.93 +- .16s	5.24 +- .16s
N = 10 Milhões	4.08 +- .12s	8.00 +- .20s	10.57 +- .23s

O speedup do algoritmo foi o seguinte:

Tabela 2: Speedup de acordo com tamanho da entrada e número de CPUs

	1 CPU	2 CPUs	4 CPUs
N = 2.5 Milhões	1	0.52	0.38
N = 5 Milhões	1	0.50	0.38
N = 10 Milhões	1	0.51	0.39

Nota-se que o speedup para o algoritmo tornou-se menor que 1 assim que se aumentou o número de CPUs. Como discutido antes, o gargalo do algoritmo é a comunicação entre todas as threads com a função allGather. Como essa agregação é feita uma vez para cada item, isso torna o algoritmo muito lento.

No allGather, cada thread faz o envio da sua porção do vetor para todas as n-1 outras threads, totalizando n*(n-1). Isso significa que, para 2 CPUs, temos 2*1 = 2 envios a cada repetição do laço externo. Para 4 CPUs, temos 4*3 = 12 envios a cada repetição do laço externo. Ou seja, o número de envios aumenta quadraticamente com o número de threads, o que faz do algoritmo nada eficiente.

Porém, o speedup teórico ainda era maior que 1. Por que os speedups ficaram menores que 1? Minha hipótese é que não há como prever o speedup utilizando a Lei de Amdahl com o algoritmo que eu fiz.

No meu cálculo, boa parte do tempo sequencial era relacionado à comunicação entre as threads (para ser mais exato, 2.81 dos 2.82 segundos sequenciais). E, caso eu aumentasse o número de threads, esse número aumentaria ainda mais. Isso quer dizer que a porcentagem de tempo puramente sequencial não é algo constante, mas é algo que varia com a quantidade de threads. Se eu calculasse o tempo sequencial com 1 thread, eu obteria um valor absurdamente menor e speedups absurdamente maiores, o que iria condizer ainda menos com a realidade.

Tabela 3: Eficiência de acordo com tamanho da entrada e número de CPUs

	1 CPU	2 CPUs	4 CPUs
N = 2.5 Milhões	1	0.26	0.10
N = 5 Milhões	1	0.25	0.10
N = 10 Milhões	1	0.26	0.10

Nota-se que a eficiência do algoritmo também diminui com o aumento do número de threads, demonstrando que não temos escalabilidade forte. Porém, observa-se algo interessante. Com um mesmo número de CPUs, a eficiência mantém-se a mesma, mostrando que temos escalabilidade fraca no nosso algoritmo. Acredito que isso tenha ocorrido, pois para cada iteração do laço externo do for, o algoritmo com 1 núcleo executa em um tempo x e o algoritmo com mais de um núcleo executa em tempo x + ε, para algum ε real. Não importa quantas repetições um algoritmo faça a mais que o outro, sempre teremos essa proporção entre os tempos do laço.