Trabalho Prático OpenMP

Pietro Polinari Cavassin

30 de Março de 2022

1 Algoritmo knapsack ingênuo

```
int knapSack(int W, int wt[], int val[], int n)
{
  if (n == 0 || W == 0)
    return 0;

  if (wt[n - 1] > W)
    return knapSack(W, wt, val, n - 1);

  else
    return max(
      knapSack(W - wt[n - 1], wt, val, n - 1) + val[n - 1] ,
      knapSack(W, wt, val, n - 1)
    );
}
```

O algoritmo se comporta como uma árvore de chamadas recursivas. No caso geral, para cada item que pode ser inserido na mochila, são testados os os melhores casos em que ele está e que ele não está na mochila e o máximo é retornado.

Ou seja, para o primeiro item haverá duas chamadas recursivas, uma o considerando dentro da mochila e outra o considerando fora. Para cada uma dessas chamadas, serão feitas mais duas chamadas, totalizando 4 chamadas recursivas (além das duas chamadas anteriores). Para n itens, o pior caso é que hajam 2^n chamadas em um nível de recursão.

Quando um item não pode ser inserido na mochila, é apenas feita uma chamada recursiva, considerando ele fora. Se não há mais itens ou espaço na mochila, é apenas retornado o valor 0.

No formato sequencial, o algoritmo primeiro considerará o item atual dentro da mochila e testará todas as possibilidades com ele dentro para depois testar com ele fora da mochila. Isso se assemelha a uma busca em profundidade numa árvore de recursão: eu testo todas as possibilidades do filho da esquerda para depois testar o filho da direita.

2 Paralelizando a recursão

A paralelização do algoritmo foi feita utilizando *tasks*. Se o número de *threads* executando o algoritmo é t, são criadas $t - 1$ tasks. Até que o limite de *tasks*

seja atingido, sempre que um item é o primeiro a ser incluído na mochila, a thread que está executando a função cria uma *task* para resolver o caso em que o item não é incluído.

Por exemplo, suponha que existem três threads e os itens i_1, i_2 e i_3, nessa ordem. Na primeira chamada de `knapSack`, a thread principal cria uma *task* para resolver caso que exclui i_1 da mochila. Após isso, ela própria computa o caso que inclui i_1 na mochila.

A thread que receber o caso sem i_1 na mochila resolve o caso que inclui i_2 e cria uma *task* para resolver o caso que exclui i_2. Como o limite de tasks criadas $(t-1)$ foi atingido, a última task criada resolverá os casos restantes.

O algoritmo fica como segue:

```

int main(){
  ...
  // cria threads solicitadas e chama knapSack com uma delas.
  #pragma omp parallel num_threads(n_thrds)
  #pragma omp single
  {
    int free_thrds = omp_get_num_threads() - 1;
    printf("%d\n", knapSackInit(W, wt, val, n, free_thrds));
  }
  ...
}
```

Na função `main` são criadas `n_thrds` threads e uma delas é encarregada de chamar a função `knapSackInit`.

```
int knapSackInit(int W, int wt[], int val[], int n, int free_thrds)
{
  if (n == 0 || W == 0)
    return 0;

  if (wt[n - 1] > W)
    return knapSackInit(W, wt, val, n - 1, free_thrds);

  int value_1, value_2;
  if (free_thrds){
    #pragma omp task shared(value_1)
    value_1 = knapSackInit(W, wt, val, n - 1, free_thrds-1);

    value_2 = knapSack(W - wt[n - 1], wt, val, n - 1) + val[n - 1];
    #pragma omp taskwait
  }
  else {
    value_1 = knapSack(W - wt[n - 1], wt, val, n - 1) + val[n - 1];
    value_2 = knapSack(W, wt, val, n - 1);
  }
  return max(value_1, value_2);
}
```

Note que `knapSackInit` possui um parâmetro a mais que `knapSack` e trata o caso de duas chamadas recursivas de maneira diferente.

O parâmetro a mais (`n_thrds`) contabiliza o número de threads que ainda estão livres para receber uma task e garante que não será criado um número

excessivo de tasks.

Caso haja alguma thread livre (linha 10), é criada uma nova task que chama `knapSackInit` considerando uma thread a menos disponível e excluindo o item atual da mochila. Depois, (linha 14) é resolvido o problema incluindo o item atual na mochila. Como já existe um item na mochila, não serão criadas novas threads, para esse subproblema e ele é resolvido com a função `knapSack` original.

Caso não haja mais nenhuma thread livre (linha 17), o restante dos casos é resolvido sequencialmente pela função `knapSack` original.

Esse algoritmo permite um balanceamento maior entre os tempos de execução das threads, visto que praticamente todas as threads iniciam a execução com exatamente um item na mochila.

3 Recursos computacionais e metodologia

Alguns dados da máquina e da compilação:

- Sistema Operacional: Linux Mint 20.1 (Ulyssa)
- Versão do kernel: 5.4.0-100-generic.
- Processador: Intel(R) Core(TM) i7-7740X 4.30GHz.
- CPU: 4 núcleos com 2 threads cada.
- Compilador: gcc versão 9.4.0
- Flags de compilação: `-fopenmp -O3`

Para cada número de núcleos e tamanho de entrada, o algoritmo foi executado 50 vezes, os tempos foram medidos pelo próprio programa em C.

4 Região sequencial

O trechos de leitura da entrada e alocação de vetores, além algumas partes da função `knapSack` são puramente sequenciais. Além disso, um nível de recursão depende da execução dos níveis "mais profundos"de recursão.

Se $n :=$ tamanho do vetor de entrada, há no máximo n níveis de recursão. Então, o tempo puramente sequencial se dá pela soma do tempo de leitura, mais n vezes o tempo de execução da função `knapSack`.

Para medir os tempos, foi utilizada a função `gettimeofday`, da biblioteca `sys/time.h`. A função é chamada no início e no final do trecho a ser mensurado e o tempo entre as chamadas é calculado pela seguinte função:

```
1 double time_dif(struct timeval start, struct timeval end){
2   return (
3     (end.tv_sec - start.tv_sec) * 1e6 +    // Segundos
4     (double) (end.tv_usec - start.tv_usec) // Microsegundos
5   )/ 1e6 ; // Conversao final para segundos
6 }
```

Foram medidos os tempos de alocação de vetores e leitura da entrada na função `main` e o tempo de cálculos dos dois primeiros `if`s e `max` na função `knapSack`.

O tempo de alocação e leitura de entrada médio para 50 execuções foi de $53\mu s$. O tempo médio de execução sequencial da primeira chamada recursiva da função `knapSack` foi desprezível (menor que $1\mu s$) para 48 dos 50 testes. Como houve chamadas cujo tempo foi de $1\mu s$, vamos considerar que o tempo sequencial de uma chamada de `knapSack` é de $1\mu s$

Dessa forma, para uma entrada de 200 itens, temos um tempo sequencial total de

$$53\mu s + 200 * 1\mu s = 253\mu s = 0.000253s$$

Já o tempo médio para execução das 50 entradas de tamanho 200 foi de $30.940895s$

Portanto, a porcentagem de tempo puramente sequencial é de

$$\frac{0.000253}{30.940895} = 0.000818\%$$

5 Aplicando a Lei de Amdahl

Pela Lei de Amdahl, o speedup máximo teórico $s(p)$ para p processadores e β a fração da computação que é sequencial, é calculado por:

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

Considerando $\beta = 0.000818\% = 0.00000818$, temos:

$$S(p) = \frac{1}{0.00000818 + \frac{0,99999182}{p}}$$

Dessa forma,

- Para 2 núcleos:

$$S(2) = \frac{1}{0.00000818 + \frac{0,99999182}{2}} = 1,99998364 \approx 2$$

- Para 4 núcleos:

$$S(4) = \frac{1}{0.00000818 + \frac{0,99999182}{4}} = 3,99990184 \approx 4$$

- Para 8 núcleos:

$$S(8) = \frac{1}{0.00000818 + \frac{0,99999182}{8}} = 7,99954195 \approx 8$$

- Para infinitos núcleos:

$$\lim_{n\to\infty} S(n) = \lim_{n\to\infty} \frac{1}{0.00000818 + \frac{0,99999182}{n}} = \frac{1}{0.00000818} \approx 122249$$

6 Calculando speedups e eficiência

Tempos médios (segundos)

N	1 CPU	2 CPUs	4 CPUs	8 CPUs
50	6.0e−4 ± 8.0e−4	5.7e−4 ± 7.5e−4	9.4e−4 ± 9.5e−4	4.5e−3 ± 2.9e−3
100	0.030 ± 0.036	0.026 ± 0.029	0.023 ± 0.027	0.022 ± 0.018
200	30.94 ± 50.79	28.73 ± 46.41	26.87 ± 44.88	20.05 ± 31.50
201	31.26 ± 51.55	28.97 ± 46.85	27.10 ± 45.25	20.25 ± 31.82
202	31.12 ± 51.06	28.89 ± 46.65	27.00 ± 45.12	20.17 ± 31.67

Foram definidos dois grupos diferentes de casos de teste para fazer as medidas de tempo. Primeiro, foram medidos os tempos utilizando entradas de tamanho 50, 100 e 200, para sempre ir dobrando o tamanho da entrada. Porém, como o algoritmo é exponencial, quando o tamanho de uma entrada é dobrado, o tempo de execução aumenta exponencialmente. Então há uma grande diferença nas ordens dos tempos de execução para as entradas de 50 a 200.

Com isso em mente, foram realizados testes com mais dois tamanhos de vetor: 201 e 202. No pior caso, o tempo de execução dobraria com o incremento de um item na entrada. Porém, muitas chamadas recursivas duplas não são feitas, e por isso esses incrementos no tamanho da entrada não surtiram efeitos significativos no tempo de execução.

Outra observação importante sobre os tempos de execução é que seus desvios padrão estão muito grandes. Isso ocorre, pois a ordem dos pesos nos casos de teste importa. Pesos pequenos nas últimas posições do vetor acarretam em uma probabilidade maior de ocorrerem mais chamadas recursivas em um nível mais profundo da árvore, tornando o algoritmo mais lento. Pelo caminho oposto, pesos grandes no final do vetor acarretam em menos chamadas recursivas em um nível profundo, fazendo com que o algoritmo execute em um menor tempo. Dado que o algoritmo é exponencial, o tempo torna-se muito volátil dependendo da entrada.

Com as medidas de tempo, conseguimos calcular os speedups e eficiência para cada quantidade de núcleos e tamanho de entrada.

Speedup

N	1 CPU	2 CPUs	4 CPUs	8 CPUs
50	1	1.05	0.64	0.13
100	1	1.15	1.30	1.36
200	1	1.08	1.15	1.54
201	1	1.08	1.15	1.54
202	1	1.08	1.15	1.54

Pela tabela, pode-se observar que o speedup para entradas de tamanho 50 piorou com o incremento no número de threads. Uma causa provável é que o tempo para a criação e designação de uma task para um núcleo é significativamente maior que o tempo para a execução do algoritmo sequencial para

pequenas entradas.

Com o aumento do tamanho da entrada para 200, observa-se um speedup maior que 1 e que aumenta com a criação de mais núcleos, porém não acompanha os speedups teóricos, que seriam aproximadamente o número de CPUs executando o processo.

Caso a função recursiva sempre percorresse todos os 2^n nós da árvore de recursão, seria bem possível termos um speedup linear, pois poderíamos distribuir as chamadas recursivas de certo nível entre os processadores e o tempo de computação das chamadas seria distribuído igualmente. Porém, não é isso o que ocorre.

Na realidade, a árvore de recursão é bem desbalanceada. A partir do momento que uma chamada recursiva adiciona um item na mochila, a capacidade de carga diminui, e por consequência, a probabilidade de mais um item caber nela também. Porém, a chamada recursiva que não adiciona o item atual ainda tem uma alta probabilidade de poder adicionar um elemento na mochila.

Da mesma forma, os itens possuem pesos diferentes. Alguns itens, quando adicionados na mochila, não permitem que muitos outros itens sejam adicionados, diminuindo a quantidade de chamadas recursivas, enquanto outros, mais leves, permitem várias possibilidades de chamadas recursivas.

Por esse motivo, balancear as tarefas para que executem em tempos semelhantes sem deixar uma thread ociosa é um grande desafio. Minha solução envolveu executar todas as threads com apenas um ou nenhum item na mochila. Dessa forma, é dado um passo interessante em direção ao balanceamento das chamadas recursivas.

Eficiência

N	1 CPU	2 CPUs	4 CPUs	8 CPUs
50	1	0.53	0.16	0.02
100	1	0.58	0.33	0.17
200	1	0.54	0.29	0.19
201	1	0.54	0.29	0.19
202	1	0.54	0.29	0.19

Como consequência do speedup menos que linear, a eficiência do algoritmo decresce com o aumento de núcleos. Dessa forma, não há escalabilidade forte no programa.

De maneira semelhante, observando as diagonais destacadas em amarelo e azul na tabela, também não há escalabilidade fraca, nem quando a entrada dobra de tamanho(amarela), nem quando seu tamanho é incrementado unitariamente para dobrar a complexidade de tempo (azul).

Como discutido na seção do speedup, o grande vilão da escalabilidade é o desbalanceamento da árvore de recursão que permite que nós de mesmo nível possam ter tempos de execução discrepantemente diferentes.