EP 1 - MAC0219

Bruno Daiki Yamada N^o 9348780

Alan Barzilay N^o 8630515

24 de Maio de 2018

1 Objetivo

Elaborar um programa capaz de realizar multiplicação de duas matrizes de números reais em `double precision` utilizando bibliotecas de paralelização e conhecimento do uso do cache do computador.

2 Método

O algoritmo trivial de multiplicação de matriz não é *cache-friendly*. Uma prática comum ao tentar tornar algoritmos *cache-friendly*, é realizar divisão e conquista, diminuindo o problema até que as entradas sejam pequenas suficiente para que inteiramente caibam no *cache*, diminuindo o tempo de execução do programa.

2.1 Divisão e conquista

Considere as matrizes $\mathbf{A} \in \mathbb{R}^{\mathbf{n}\times\mathbf{o}}, \mathbf{B} \in \mathbb{R}^{\mathbf{o}\times\mathbf{m}}$ e $\mathbf{C} \in \mathbb{R}^{\mathbf{n}\times\mathbf{m}}$. Desejamos calcular o valor de $\mathbf{C}$:

$$\mathbf{C} = \mathbf{AB} \tag{1}$$

Para isso, o algoritmo trivial envolve a multiplicação elemento a elemento das linhas de $\mathbf{A}$ e colunas de $\mathbf{B}$. Essa multiplicação não é *cache-friendly* pois ao iterar pelos elementos em ambas as matrizes, existe probabilidade considerável de acontecer *cache-miss* se as matrizes forem de dimensão alta. Uma solução seria realizar a transposição da matriz $\mathbf{B}$, abusando da localidade dos elementos da mesma linha para diminuir a quantidade de *miss*. Uma outra solução, que foi implementada para nossos programas, é a divisão e conquista.

Cada matriz foi dividida em 4 matrizes bloco, como esquematizado a sequir:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{12}} \\ \mathbf{A_{21}} & \mathbf{A_{22}} \end{bmatrix} \tag{2}$$

$$\mathbf{B} = \begin{bmatrix} \mathbf{B_{11}} & \mathbf{B_{12}} \\ \mathbf{B_{21}} & \mathbf{B_{22}} \end{bmatrix} \tag{3}$$

$$\mathbf{C} = \begin{bmatrix} \mathbf{C_{11}} & \mathbf{C_{12}} \\ \mathbf{C_{21}} & \mathbf{C_{22}} \end{bmatrix} \tag{4}$$

Cada matriz bloco de $\mathbf{C}$ é calculada da seguinte maneira:

$$\mathbf{C_{11}} = \mathbf{A_{11}B_{11}} + \mathbf{A_{12}B_{21}} \tag{5}$$

$$\mathbf{C_{12}} = \mathbf{A_{11}B_{12}} + \mathbf{A_{12}B_{22}} \tag{6}$$

$$\mathbf{C_{21}} = \mathbf{A_{21}B_{11}} + \mathbf{A_{22}B_{21}} \tag{7}$$

$$\mathbf{C_{22}} = \mathbf{A_{21}B_{12}} + \mathbf{A_{22}B_{22}} \tag{8}$$

Cada multiplicação pode chamar recursivamente a mesma função de multiplicação, que encerra a chamada com a multiplicação trivial paralelizada quando a matriz for menor do que um certo limite.

No caso de matrizes retangulares, as considerações feitas ainda são válidas, gerando matrizes bloco de dimensões compatíveis para serem multiplicados até mesmo quando o número de linhas e colunas são ímpares (nesse caso dividindo de maneira não perfeitamente simétrica).

Note que essa abordagem não diminui a quantidade de multiplicações necessárias para realizar a multiplicação. Não alterando a complexidade do algoritmo.

2.2 Paralelização

Como criar um número muito grande de *threads* pode diminuir o desempenho do programa, escolhemos paralelizar somente a multiplicação das matrizes de dimensão *cache-friendly*.

2.2.1 OpenMP

Uma diretiva de paralelização de `for` foi adicionada para que a multiplicação de matriz fosse paralelizada. Como cada *thread* é encarregada de calcular uma iteração, não há seções críticas.

2.2.2 Pthreads

Para realizar a paralelização da multiplicação das matrizes de dimensão *cache-friendly* com a biblioteca Pthreads, nós nos utilizamos de duas funções (uma principal e uma auxiliar) e um *struct*. Uma função principal gera as *threads* e os seus respectivos *structs*, onde cada *thread* executara a função auxiliar com os argumentos recebidos de um *struct*.

Para nos aproveitarmos da localidade e minimizar o numero de *cache-misses*, fizemos com que cada *thread* realize, em um loop, a multiplicação de uma linha da matriz de cada vez com um passo de tamanho N, sendo N o numero de *threads* criadas. Dessa maneira todas as linhas são multiplicadas e as *threads* não competem entre si por qual sera a próxima linha multiplicada, portanto novamente não há seção crítica.

Realizar a multiplicação das linhas dessa maneira alternada se mostra mais eficiente do que passar um 'bloco' de linhas para cada *thread*, pois dessa maneira as linhas passadas são mais próximas e acabam cabendo melhor no *cache* e assim ele se mantém "mais quente".

2.3 Matrizes unidimensionais

Considere uma matriz $\mathbf{A} \in \mathbb{R}^{\mathbf{n}\times\mathbf{m}}$, e um vetor $\mathbf{V} \in \mathbb{R}^{\mathbf{nm}}$. Ambas estruturas têm o mesmo número de *elementos armazenados*. Uma estrutura como $\mathbf{A}$ pode ser colapsada para ser representada de maneira equivalente em $\mathbf{V}$. No nosso programa, os primeiros m elementos representam os elementos da linha 0, enquanto que os m elementos seguintes representam a linha 1, e assim por diante até a linha n, que é representada pelos últimos m elementos do vetor $\mathbf{V}$.

Testes preliminares apontaram que multiplicação de matrizes em vetores unidimensionais como V são mais rápidas, devido ao melhor aproveitamento da localidade dos dados acessados. Portanto, utilizamos essa estrutura para armazenar as matrizes de entrada e saída.