Caminho Mais Curto entre Todos os Pares de Vértices

Marco Pontes, Nuno Azevedo

up201308000@fc.up.pt, up201306310@fc.up.pt

Computação Paralela

DCC/FCUP

7 Novembro de 2016

1 Motivação

O objetivo deste trabalho consiste na implementação de um algoritmo que determine os caminhos mais curtos entre pares de nós num dado grafo dirigido. O grafo é representado numa matriz quadrática com dimensão igual ao seu número de nós, cujo valor na posição *(i, j)* da matriz é o peso da aresta que faz a conexão entre os nós i e j no grafo.

O nosso programa recebe essa matriz como entrada e usando a ideia base do algoritmo de *Fox* para multiplicação de matrizes de forma distribuída utilizando várias unidades de processamento, mas em vez de multiplicar matrizes, aplica o algoritmo de *Floyd-Warshall* para encontrar o caminho mais curto entre todos os pares de nós do grafo. A comunicação entre as várias unidades de processamento é feita através de um protocolo de transmissão de mensagens denominado *MPI (Message Passing Interface)*[1].

2 Implementação

2.1 Algoritmo de *Fox*

O algoritmo de *Fox* é usado para multiplicação de matrizes em paralelo. Supondo que temos n^2 processos, a matriz inicial $N \times N$ é dividida em sub-matrizes de dimensão $(\frac{N}{Q}) \times (\frac{N}{Q})$ e atribui cada sub-matriz a um processo.

O exemplo seguinte ilustra a aplicação do algoritmo a duas matrizes quadráticas de tamanho 2.

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} = \begin{pmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{pmatrix} \tag{1}$$

Para calcular cada sub-matriz, cada processo necessita da informação dos processos que ficaram responsáveis pelas sub-matrizes da mesma linha e coluna.

Cada sub-matriz das matrizes originais (A e B) são entregues a um processo da seguinte forma:

P_{00}	P_{01}
P_{10}	P_{11}

Após o cálculo de cada processo, volta-se a agrupar as sub-matrizes resultado de cada um destes, formando uma nova matriz C, resultante da multiplicação de A por B.

2.2 Algoritmo de *Floyd-Warshall*

O algoritmo de *Floyd-Warshall* calcula o caminho mais curto entre todos os pares de nós de um dado grafo pesado. Com uma complexidade de $O(N^3)$, para cada dois nós *(i, j)*, o algoritmo procura a possibilidade de existir um terceiro nó k cuja soma do peso das ligações entre *(i, k)* e *(k, j)* seja inferior ao custo da ligação direta entre *(i, j)*. No caso de ainda não existir ligação direta entre os nós (i, j), passa-se a ter conhecimento de que existe um caminho secundário que os une.

[1] https://www.open-mpi.org/

2.3 Estruturas de Dados Usadas

Para armazenar as várias matrizes necessárias durante o processo fizemos alocações de memória contíguas de tamanho $(n \times n \times sizeof(int))$, em que n é a dimensão da matriz e *sizeof(int)* é o tamanho em *bytes* de um número inteiro, através da função da biblioteca de C *malloc()* que nos retorna um apontador para o início da região de memória reservada. Para realizarmos o acesso à memória simulando uma matriz convertemos o apontador retornado para o tipo *(int (*)[n])*, o que nos permite depois aceder à coluna *(i, j)* da matriz através da notação *matriz[i][j]*.

Utilizamos uma *struct* para guardar todas as informações relacionadas com o *MPI*, tal como o *rank* do processo, número total de processos, posição do processo na matriz, comunicadores entre linhas e colunas e comunicador geral.

2.4 Funções Auxiliares Usadas

- ***setup_grid()***: Cria todos os comunicadores *MPI* necessários para a troca de mensagens entre os processos, e uma topologia de mapa de processos sobre a matriz de entrada para decidir qual a sub-matriz pela qual o processo ficará responsável.

- ***check_fox()***: Verifica se é possível aplicar o algoritmo de *Fox* dado uma matriz de tamanho N e P unidades de processamento.

- ***send_sub_mtrx()***: Usada pelo processo *root* para partilhar as restantes sub-matrizes com os outros processos através da função *MPI_Send()*.

- ***fix_final_mtrx()***: Ao fazer a união das sub-matrizes de todos os processos através do *MPI_Gather()* para obter a matriz final, esta matriz fica desorganizada e cada posição *(i, j)* desta não corresponde à posição *(i, j)* da matriz inicial, esta função é utilizado para voltar a organizar a matriz.

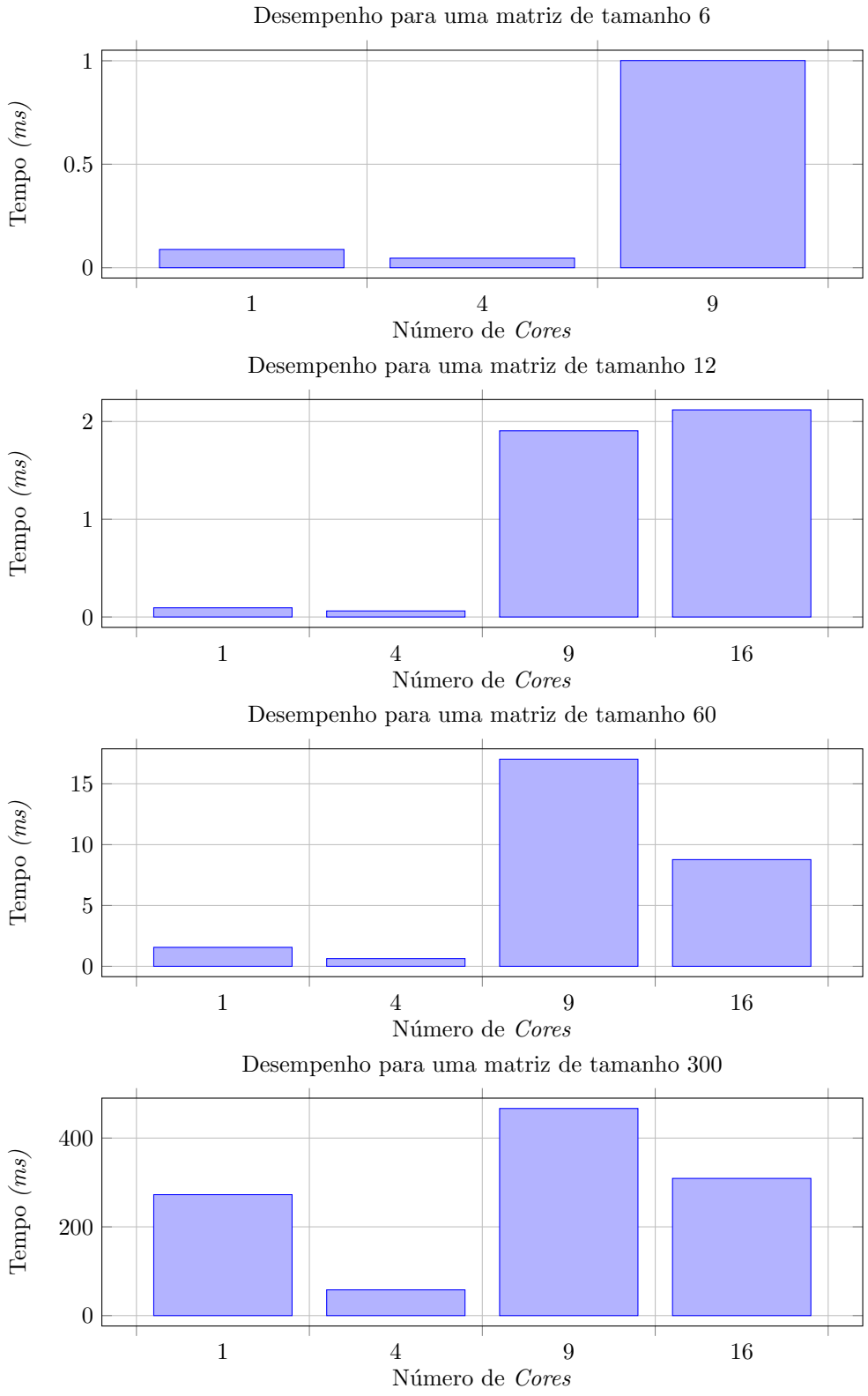
- ***print_mtrx()***: Imprime as matrizes linha por linha com as colunas separadas por espaços, para uma visualização correta.

3 Dificuldades na Implementação

A alocação de memória para as matrizes de forma eficiente foi um dos problemas que nos levou algum tempo para resolver, pois a primeira implementação a que chegamos envolvia várias chamadas à função de sistema *malloc()*, pois alocávamos um *array* com N elementos e depois em cada elemento deste voltávamos a alocar um novo *array* com N elementos, para assim representar uma matriz com $N \times N$ elementos. Devido à função *malloc()* ter um custo elevado, tentamos então encontrar uma melhor solução e após uma análise mais aprofundada verificamos que poderíamos alocar a memória para a matriz inteira de uma só vez, como se de um *array* se tratasse, e converter depois o apontador retornado pelo *malloc()* para representar um array com N elementos.

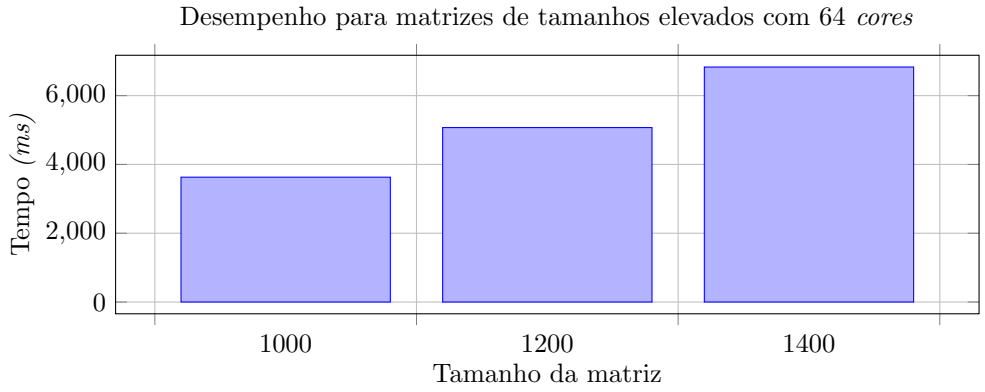
Após o cálculo de cada processo, é necessário voltar a construir uma matriz do tamanho da original através das sub-matrizes que cada um dos processos gerou. Ao realizar a chamada à função *MPI_Gather()* para unir estas sub-matrizes, temos de reestruturar a matriz gerada, para que cada par de coordenadas desta coincida com a matriz original. Tivemos de analisar como é que a concatenação das sub-matrizes estava a ser feita pelo *MPI_Gather()* para percebermos como voltar a reestruturar a matriz.

4 Avaliação de Desempenho



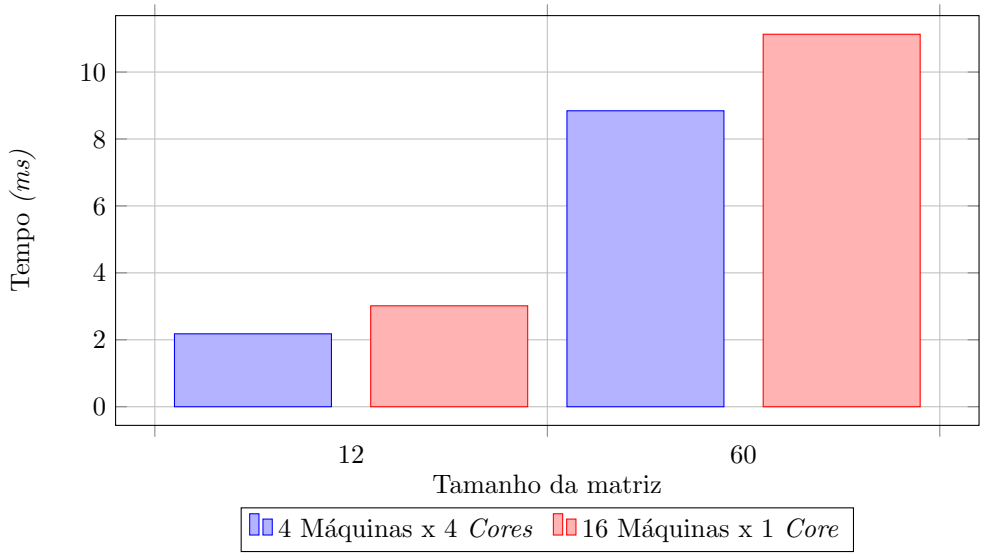
Em análise aos gráficos acima, verificamos que de uma forma geral compensa utilizar mais unidades de processamento até ao ponto em que é necessário uma nova máquina, pois aí os tempos aumentam drasticamente devido à latência da comunicação entre as máquinas. Por exemplo, vemos que em todos os gráficos acima, a diferença entre utilizar apenas 4 *cores* (1 máquina) contra utilizar 9 *cores* (3 máquinas) é elevada, neste caso os tempos de execução aumentam cerca de 10 vezes. A partir do momento em que usamos mais de uma máquina, ou seja, já introduzimos a latência de comunicação, utilizar mais máquinas tem um resultado positivo pois obtemos um maior poder de processamento, o que se pode novamente comprovar nos gráficos através da utilização de 9 *cores* (3 máquinas) contra 16 *cores* (4 máquinas).

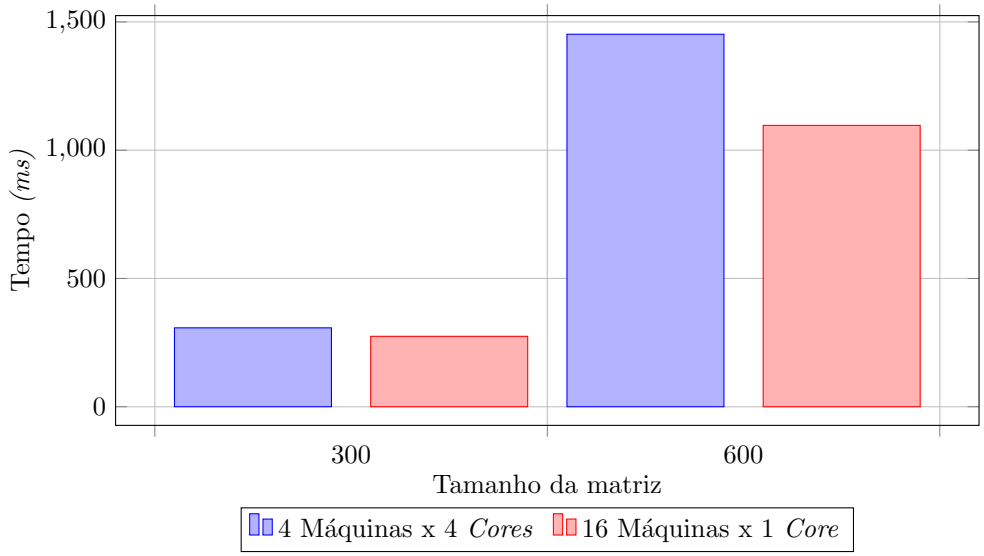
Observamos também que a latência de comunicação entre máquinas tem um impacto tão grande tal que a resolução do problema localmente com apenas 1 unidade de processamento tem um desempenho superior à da utilização de 4 máquinas constituindo um total de 16 *cores*.



Com o objetivo de conhecer melhor as capacidades do *MPI* e também para testar o nosso programa para matrizes de elevadas dimensões, geramos três matrizes de tamanhos 1000, 1200 e 1400 e aplicamos o algoritmo com 16 máquinas usando um total de 64 *cores*. Como verificamos no gráfico acima, foi possível o cálculo para todos os casos.

4.1 Avaliação de Desempenho em Diferentes Arquiteturas





Fizemos a comparação de desempenho para matrizes de tamanho 12, 60 e 300 em dois ambientes:

- 1º Ambiente: 4 máquinas em que cada uma utiliza 4 *cores*;
- 2º Ambiente: 16 máquinas em que cada uma utiliza 1 *cores*.

Verificamos que com matrizes de tamanho médio (12 e 60) o desempenho no 1º ambiente é relativamente melhor, enquanto que para matrizes de tamanho elevado (300 e 600) vemos que começa a compensar utilizar mais máquinas com menos *cores* em vez de poucas máquinas com muitos *cores*.

5 Conclusão

A realização deste trabalho permitiu o aprofundamento do nosso conhecimentos em vários temas. Aprendemos imenso sobre a gestão de memória dinamicamente em *C* tendo sempre como objetivo um acesso rápido e uso eficiente da memória.

O conhecimento adquirido sobre o *MPI* será bastante útil em aplicações futuras, devido a ser uma ferramenta de computação muito poderosa que nos permite abstrair da complexidade existente na computação paralela.

Consideramos também que o problema a resolver foi adequado para a aplicação deste método de computação e nos deu bastantes ideias para o colocar em prática numa situação real.