Comparação da Variabilidade de Tempo em Diferentes Algoritmos de Multiplicação de Matrizes utilizando Paralelização e Divisão e Conquista

Victor H. de Oliveira, Rogério Aparecido Gonçalves, João Fabrício Filho

[1]Universidade Tecnológica Federal do Paraná – Câmpus Campo Mourão
Via Rosalina M Santos, 1233 – 87301-899 – Campo Mourão – PR – Brasil

victorhugooliveira@alunos.utfpr.edu.br, {rogerioag,joaof}@utfpr.edu.br

Abstract. *Matrix multiplication is a common application in the computational field. The objective of this work is to compare distinct approaches implementing matrix multiplication in relation to time execution. Dynamic and static allocation techniques, blocking and parallelization with multiple threads are covered. The strategy involving blocking implemented together with parallelization achieved performance gains* 75.12% *greater than the sequential matrix multiplication approach and* 4.5% *faster than parallel execution without the use of blocking.*

Resumo. *A multiplicação de matrizes é uma aplicação comum no âmbito computacional. O objetivo deste trabalho é comparar abordagens distintas de implementação da multiplicação de matrizes em relação ao tempo de execução. São abordadas as técnicas de alocação dinâmica, estática, emprego de* blocking *matricial e paralelização com múltiplas* threads*. A estratégia envolvendo* blocking *implementada juntamente com paralelização alcançou ganhos de desempenho* $75,12\%$ *maiores do que a abordagem sequencial de multiplicação matricial e* $4,5\%$ *mais rápida do que a execução paralela sem o emprego de* blocking*.*

1. Introdução

As relações entre somas e multiplicações na operação de multiplicação de matrizes trazem importantes características para seu estudo. Nessa operação, o número de multiplicações é superior ao número de adições, o que exige instruções com maior tempo de resposta da *Central Processing Unit* (CPU). O maior tempo de resposta implica em maior consumo de energia e menor desempenho em execuções típicas na CPU. O objetivo deste trabalho é contornar essa condição por meio de técnicas de programação paralela [Pacheco 2011], divisão e conquista [Huss-Lederman et al. 1996] e alocação dinâmica de memória [Supriya P. Mali 2019].

Os resultados apresentam que a alocação dinâmica demanda, aproximadamente, 7% mais tempo do que a alocação estática em algoritmos de execução paralela. A execução serial apresentou maior aproveitamento da alocação dinâmica de memória, melhoria de 15% no tempo de execução, comparando com a alocação estática de memória. Nas versões paralelizadas, a estratégia incluindo a técnica de *blocking* e alocação estática é $4,5\%$ mais rápida do que a execução paralela sem o emprego do algoritmo de *blocking*.

Agradecimentos ao CNPq pelo auxílio financeiro e concessão de bolsa - Projeto 01947 SISPEQ/UTFPR.

2. Materiais e Métodos

O Código 1 apresenta a implementação-base deste trabalho em `C` da multiplicação de matrizes.

Código 1. Multiplicação de Matrizes.

```
1  void matrix_mul(int *a, int *b,int *c, int size) {
2    for (int i = 0; i < size; i++)
3      for (int j = 0; j < size; j++) {
4        int aux = 0;
5        for (int k = 0; k < size; k++)
6          aux += a[i * size + k] * b[k * size + j];
7        c[i * size + j] = aux;
8        aux = 0;
9      }
10 }
```

Para a alocação dinâmica de memória, há três ponteiros globais, que serão acessados dentro da função de multiplicação matricial. Utilizamos a alocação dinâmica quando não sabemos previamente a quantidade de memória que utilizaremos em uma aplicação [Supriya P. Mali 2019].

A implementação do algoritmo de divisão e conquista possui dois laços de repetição adicionais, que dividem a matriz em blocos menores. Seu objetivo é reduzir a quantidade de multiplicações que o processador realiza, conforme mostrado no Código 2. A biblioteca `Posix Threads` [Pacheco 2011] foi utilizada na paralelização dos algoritmos.

3. Resultados e Discussão

Como ambiente experimental utilizou-se o Sistema Operacional `GNU/Linux` com a distribuição Ubuntu $22.04.4$ LTS executando em um processador *Intel(R) Core(TM) i7-4790 CPU* @ $3.60GHz$ com 4 núcleos físicos e 32GB de memória principal. Para utilização dos núcleos físicos, na implementação foram criadas quatro *threads*.

Para a correta avaliação das cargas de trabalho nos diferentes níveis de *cache* na hierarquia de memória, foram avaliadas entradas com matrizes quadradas em potências de 2: $1024x1024$, $2048x2048$ e $4096x4096$. As matrizes quadradas de dimensões 4096, por exemplo, possuem $16.777.216$ elementos. Em uma multiplicação envolvendo duas matrizes semelhantes, teremos $68.719.476.736$ multiplicações e $68.702.699.520$ adições.

3.1. Implementação Serial

A Tabela 1 apresenta os resultados da multiplicação de matrizes serial simples, com implementação com alocação dinâmica (*malloc*) e com divisão por *blocking*.

Para tamanhos menores, a implementação da divisão e conquista apresentou maior desempenho em comparação com as outras técnicas. A técnica adiciona a complexidade de divisão, que pode não trazer benefícios se o tamanho de entrada não for grande o suficiente. Porém, ao dobrar o tamanho, a execução serial com alocação estática teve menor tempo de execução. A divisão e conquista foi a segunda mais rápida, em comparação.

Tabela 1. Variação de tempo de acordo com algoritmos de multiplicação matricial (Execução Serial).

	Tempo (Segundos)			
Tamanho	**Serial**	**S/ + Malloc + Blocking**	**S/ + Blocking**	**S/ + Malloc**
1024x1024	10,45	10,47	6,87	10,52
2048x2048	123,54	154,41	134,68	162,36
4096x4096	1729	1812	1625	1467

Entretanto, ao dobrar novamente o tamanho, a alocação dinâmica de memória acabou por ser a mais eficiente dentre as outras técnicas. Logo após ela, *blocking*. O algoritmo que envolve divisão e conquista juntamente com alocação dinâmica foi o de menor desempenho, com defasagem de aproximadamente 20% em relação ao primeiro colocado. Isso se deve às operações adicionais da divisão e conquista, que não estão sendo aproveitadas para paralelização nesta versão da implementação.

3.2. Implementação Paralela

A Tabela 2 apresenta os resultados do algoritmo paralelo de multiplicação matricial com 4 *threads*, e suas correspondentes implementações com *blocking* e alocação dinâmica (*malloc*).

Código 2. Multiplicação de matrizes implementado com *blocking*

```
1  void matrix_mul(int* a, int* b,int* c, int size){
2    int blockSize = block_size;
3    for (int i = 0; i < size; i+=blockSize)
4      for (int j = 0; j < size; j+=blockSize)
5        for(int blockRow = i; blockRow < i + blockSize; blockRow
             ++) {
6          for (int blockCol = j; blockCol < j + blockSize;
               blockCol++) {
7            int aux = 0;
8            for (int k = 0; k < size; k++) {
9              aux += a[k + blockRow * size] * b[blockCol + k *
                   size];
10           }
11           c[blockCol + blockRow * size] = aux;
12         }
13 }
```

Tabela 2. Variação de tempo de acordo com algoritmos de multiplicação matricial (Execução Paralela).

	Tempo (Segundos)			
Tamanho	**Paralelo**	**P/ + Blocking + Malloc**	**P/ + Blocking**	**P/ + Malloc**
1024x1024	2,74	2,36	2,07	2,61
2048x2048	49,24	43,4	44,87	50,44
4096x4096	450,3	484,85	430,02	480,21

Destaca-se novamente o algoritmo de divisão e conquista, que obteve bom desempenho em dimensões maiores. A estratégia de separar a matriz total em blocos menores e, a partir desses blocos, *threads* executarem em paralelo, obtém melhor desempenho em conjunto.

A técnica menos eficiente, em dimensões maiores, é representada pela terceira coluna da Tabela 2. Com desempenho 12% menor que a técnica de divisão e conquista. A diferença entre os desempenhos de alocação dinâmica e estática se devem ao tempo adicional para definição, alocação e liberação de dados não utilizados na memória.

4. Considerações Finais

Este trabalho implementou diferentes algoritmos para o problema de multiplicação de matrizes, avaliou e comparou diferentes técnicas de paralelização. Os resultados experimentais evidenciam que a abordagem de **divisão e conquista paralelizada de alocação de memória estática** obteve maior desempenho do que outras técnicas avaliadas. Implementando quatro *threads* de execução paralela, essa estratégia oferece ganhos de desempenho de até $3,84x$ no tempo de execução comparando-a com a abordagem clássica de multiplicação de matrizes.

Comparando as versões paralelizadas, a estratégia incluindo a técnica de *blocking* e alocação estática é 4,5% mais rápida do que a execução paralela sem o emprego do algoritmo de *blocking*. Para dimensões progressivamente maiores, essa diferença percentual produz benefícios relacionados ao desempenho e energia, uma vez que recursos computacionais são utilizados de maneira mais eficiente.

Sugere-se como trabalhos futuros as implementações das técnicas de divisão e conquista junto a alocação estática que possuem potencial para o alto desempenho em matrizes de grandes dimensões.

Referências

[Huss-Lederman et al. 1996] Huss-Lederman, S., Jacobson, E. M., Tsao, A., Turnbull, T., and Johnson, J. R. (1996). Implementation of strassen's algorithm for matrix multiplication. In *Supercomputing*, USA. IEEE Computer Society.

[Pacheco 2011] Pacheco, P. S. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers.

[Supriya P. Mali 2019] Supriya P. Mali, Sonali Dohe, P. R. (2019). Memory management techniques: Static and dynamic memory allocation. *International Journal of Current Engineering and Technology*, 9(1):92–94.