Vinícius Campos Tinoco Ribeiro

Programação com *Threads*

Brasil

2017, v-1.0

Vinícius Campos Tinoco Ribeiro

Programação com *Threads*

Relatório apresentado à disciplina de Programação Concorrente com o objetivo de avaliar o conhecimento de uso de threads e atuar como uma atividade avaliativa parcial da primeira unidade.

Universidade Federal do Rio Grande do Norte – UFRN

Instituto Metrópole Digital – IMD

Bacharelado em Tecnologia da Informação

Brasil

2017, v-1.0

Lista de ilustrações

Lista de tabelas

Sumário

Introdução

A utilização dos recursos computacionais de forma eficiente é um dos principais assuntos da atualidade e uma das formas de se otimizar programas de computadores é através do uso de threads. Então, este trabalho tem como objetivo analisar o tempo de processamento de multiplicações de matrizes quadradas de forma sequencial e paralela.

Este relatório está dividido da seguinte maneira: A Seção 1 descreve os algoritmos desenvolvidos, a Seção 2 descreve a metodologia utilizada nos experimentos, a Seção 3 informa os resultados obtidos e realiza comparação entre eles; e por fim a última Seção apresenta as conclusões obtidas acerca dos experimentos.

1 Implementação

O projeto foi desenvolvido na linguagem **C++** e a Figura 1 apresenta a estrutura de diretórios adotada na implementação dos algoritmos. Cada pasta tem as seguintes funções:

- **include**: Possui a definição da classe *Matrix*, assim como a interface dos seus métodos. Essa é a classe responsável por lidar com as matrizes e suas respectivas operações, principalmente com as multiplicações sequenciais e paralelas.
- **src**: Possui as implementações dos métodos da classe *Matrix* e possui os dois arquivos responsáveis por executar programas sequenciais e paralelos.
- **bin**: Possui os executáveis do projeto, além de dois scripts para executar os experimentos.
- **build**: Se trata do local onde ficam os arquivos compilados e configurações do **CMake**.
- **input**: Possui todas as matrizes utilizadas nos testes.
- **output**: Possui todas as matrizes de saida, assim como os tempos de todas as execuções realizadas.

Todo o projeto foi construido com orientação à objetos, de modo que as matrizes fossem genéricas e fosse possível alocar qualquer tipo de dado nela. Para utilizar a classe *Matrix*, basta importar o arquivo **matrix.h**, e declarar cada matriz na forma **Matrix<T> nome_matrix(dimensao, threads)**, onde **T** é o tipo dos dados alocados na matriz, **dimensao** é o tamanho da matriz quadrada e por fim, **threads** é o número máximo de threads que deseja-se utilizar, este é opcional, caso que se tem apenas multiplicações sequenciais.

Para acessar cada posição da matriz, pode-se utilizar colchetes, já que esse método foi sobrescrito. Além disso, é possivel realizar operações de soma e subtração entre matrizes através dos operadores + e **-**, respectivamente. Para realizar a multiplicação sequencial pode-se utilizar o operador ***, e caso for desejado a realização de um produto com maior performance, ou seja, em paralelo, pode-se utilizar o método **multiply(Matrix<T> matrizB)**, onde **matrizB** se trata da matriz que deseja-se realizar o produto.

Para compilar o projeto, é necessário que se tenha instalado o **CMake**[1], então o processo de compilação segue os seguintes passos:

[1] https://cmake.org/

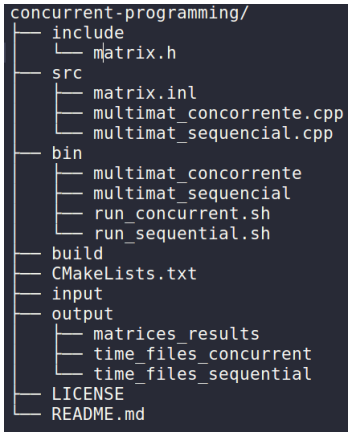

Figura 1 – Árvore de diretórios

Listing 1.1 – bash version

```
# Desloque-se a pasta raiz do projeto
$ cd build
$ cmake ..
$ make
```

Para realizar a execução, basta deslocar-se a pasta **bin**, e executar os arquivos na forma descrita nas definições do projeto.

O algoritmo de multiplicação de matrizes na forma sequêncial foi implementado como definido na descrição do projeto. Já a implementação com *threads* ocorre da seguinte maneira:

1. Divide-se a $\frac{dimensao^2}{número-de-threads}$, com isso, obtem-se o número de multiplicações que cada thread irá fazer, com exceção da última thread, que executará $dimensao^2 - \frac{dimensao^2}{número-de-threads}$ multiplicações.

2. Com isso, mapeia-se cada multiplicação em um vetor, e a partir disso, um novo método é chamado e que realizar as multiplicações sequenciais da multiplicação *init* à $final$, sendo esses a multiplicação inicial e a final, respectivamente.

3. Todas as threads são esperadas através do *join*, já que é necessário esperar as computações dessas threads para escrever em arquivo o resultado das multiplicações.

2 Metodologia

A realização dos experimentos ocorreram em um notebook *Dell Inspiron 15r 5588*, conta com processador Intel(R) Core(TM) i5-5200U *quad-core*, de frequência $2.20GHz$ e $8GB$ de memória RAM, com sistema operacional Ubuntu 16. Tentou-se executar os experimentos com o menor número de tarefas em paralelo ao programa, de modo que os resultados tivessem menor variação entre as execuções com o mesma dimensão e o mesmo número de threads.

O projeto foi desenvolvido no padrão *C++11*, utilizando *CMake*, um sistema multi-plataforma, para gerar a compilação automatizada. Para computar os tempos de processamento da aplicação, foi usada a biblioteca nativa *chrono*, através da classe *steady_clock* :: *time_point*. Além disso, utlizou-se a biblioteca também nativa do *C++*, *fstream*, para escrever em arquivo o resultado das soluções.

Para realizar a comparação entre as execuções dos algoritmos sequenciais e paralelos, foi utilizado o *Calc*, do *OpenOffice*, como ferramenta auxiliar de cálculo de média, mínimo, máximo e desvio padrão dos resultados. Já as tabelas e gráficos aqui mostrados foram criados com LaTeX.

3 Resultados

As Tabelas 1 e 2 mostram as medidas de dispersão quanto ao tempo de processamento, **em segundos**, acerca dos experimentos realizados. Nessas tabelas são mostrados os tempos obtidos com 1 *thread*, solução sequencial, e com 2, 3, 4, 5 e 10 threads. Percebe-se que a partir de soluções com mais de dez *threads*, todas as soluções obteve um crescimento temporal, mostrando o que tendia a ter um pior desempenho com mais threads. Porém, foi realizados diversos testes, com até 50 *threads*, e todos os resultados se encontram na pasta de *time_files_concurrent*.

Por esse motivo, utilizou-se poucas threads, já que a partir da décima, o rendimento em paralelo já não se tornaria uma boa alternativa.

Tabela 1 – Resultado das matrizes de ordem 4 x 4 à 128 x 128 em segundos.

Matriz	Threads	Tempo Max	Tempo Min	Tempo Médio	Desvio padrão
4 x 4	**1**	0,000647929	0,0001944	0,0002866679	0,0001087826
	2	0,00711749	0,000276515	0,0007563108	0,0015135322
	3	0,00127848	0,000303769	0,0004112402	0,000207761
	4	0,0044365	0,000300076	0,0008401055	0,0009242536
	5	0,00316746	0,000340144	0,0008968609	0,0008986241
	10	0,00351521	0,000415179	0,0008372593	0,0006940038
8 x 8	**1**	0,000684378	0,00024033	0,000312523	9,25015905032392E-05
	2	0,0012402	0,000310697	0,0004111567	0,0002262259
	3	0,00290975	0,000328082	0,0008527581	0,0008141044
	4	0,00264659	0,000344463	0,0007330349	0,0006949837
	5	0,0179139	0,000406013	0,001629548	0,0039381867
	10	0,00230592	0,00054849	0,0009383498	0,0004636399
16 x 16	**1**	0,00808809	0,00045597	0,001064697	0,0016745287
	2	0,00136388	0,000515465	0,0007083024	0,0002341625
	3	0,00282312	0,000498999	0,0008976657	0,0005821089
	4	0,00358292	0,000507408	0,00130075	0,0007463991
	5	0,0210142	0,000609303	0,0021994894	0,0044581517
	10	0,00418958	0,000715177	0,0016465847	0,0011131129
32 x 32	**1**	0,00299609	0,001555	0,001839893	0,0003534955
	2	0,00334676	0,00147958	0,0022711635	0,0005810656
	3	0,00317994	0,00141802	0,002010533	0,0005660794
	4	0,00465152	0,00138737	0,001999035	0,0008412062
	5	0,00538161	0,0017281	0,00267016	0,0008417098
	10	0,00735545	0,00170511	0,0026998185	0,0013379209
64 x 64	**1**	0,0134783	0,00796402	0,0093460805	0,00142117
	2	0,0119722	0,0051888	0,0077888205	0,0018259418
	3	0,0104559	0,00573996	0,0074386785	0,0013923501
	4	0,00926879	0,0051532	0,006807885	0,0013602399
	5	0,0189046	0,00695227	0,009427	0,0025766711
	10	0,0115208	0,00634166	0,008298839	0,0015300482

Tabela 2 – Resultado das matrizes de ordem 256 x 256 à 2048 x 2048 em segundos.

Matriz	Threads	Tempo Max	Tempo Min	Tempo Médio	Desvio padrão
128 x 128	**1**	0,0605327	0,0505568	0,05332585	0,0022602939
	2	0,0467112	0,0304086	0,03640618	0,0054967771
	3	0,0471429	0,0316122	0,03567939	0,0040120834
	4	0,0368523	0,0282179	0,03063885	0,0018248181
	5	0,0488993	0,0341006	0,03812272	0,0032521493
	10	0,0431514	0,0335688	0,038155825	0,0025684343
256 x 256	**1**	0,404946	0,366168	0,38159365	0,0121500048
	2	0,306271	0,186719	0,2349855	0,0444541262
	3	0,266476	0,208182	0,2328278	0,0121728387
	4	0,341917	0,192412	0,23621645	0,0397643445
	5	0,239516	0,218611	0,2278543	0,0066646433
	10	0,29462	0,217459	0,24090985	0,0218517738
512 x 512	**1**	3,27677	2,93019	3,088807	0,072512982
	2	2,42479	1,36854	1,809105	0,2959177621
	3	1,83617	1,42191	1,6104095	0,0888656068
	4	1,96361	1,49026	1,6145595	0,16467692
	5	2,10557	1,37253	1,572805	0,2041050336
	10	1,86206	1,59438	1,650977	0,0781441165
1024 x 1024	**1**	28,7697	24,2198	25,129085	1,3423348615
	2	16,308	12,0683	13,39036	1,2295538414
	3	14,6079	12,1118	13,403695	0,6003307742
	4	13,5685	11,8745	13,011125	0,5906258731
	5	16,0716	14,437	15,117965	0,4929820625
	10	16,1173	14,396	15,2779	0,463613664
2048 x 2048	**1**	301,17	201,687	237,71925	29,5332943461
	2	114,026	90,7886	104,04508	4,4240861848
	3	117,027	97,3774	109,51867	4,6576894169
	4	105,712	93,7954	101,5197	2,3753553099
	5	94,2264	81,5173	87,482545	3,4844384215
	10	118,823	102,52	110,176	4,7366735052

Os gráficos abaixo demostram o comportamento das execuções dos experimentos. Pode-se perceber que para matrizes a partir da dimensão 16^2, o uso de threads já possui um beneficio, onde consegue-se as maiores diferenças e melhores resultados com as matrizes 1024×1024 e 2048×2048, onde o uso de *threads* causou reduções com cerca de **10** e **120** segundos, respectivamente.

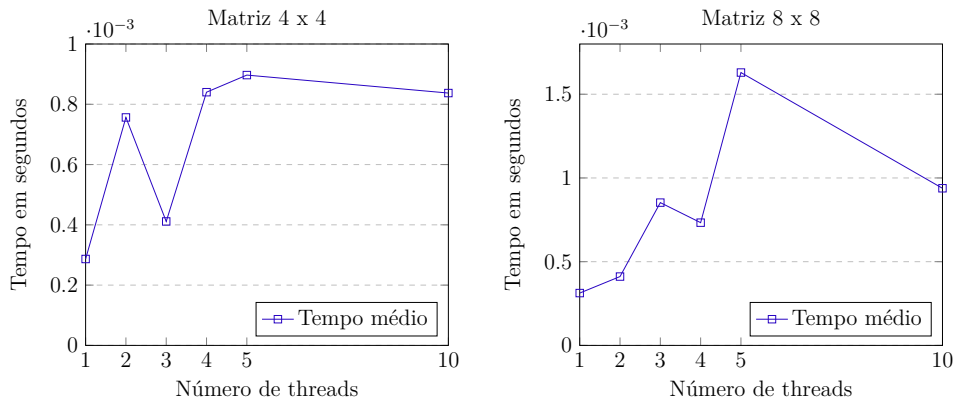

Figura 2 – Matrizes 4x4 e 8x8

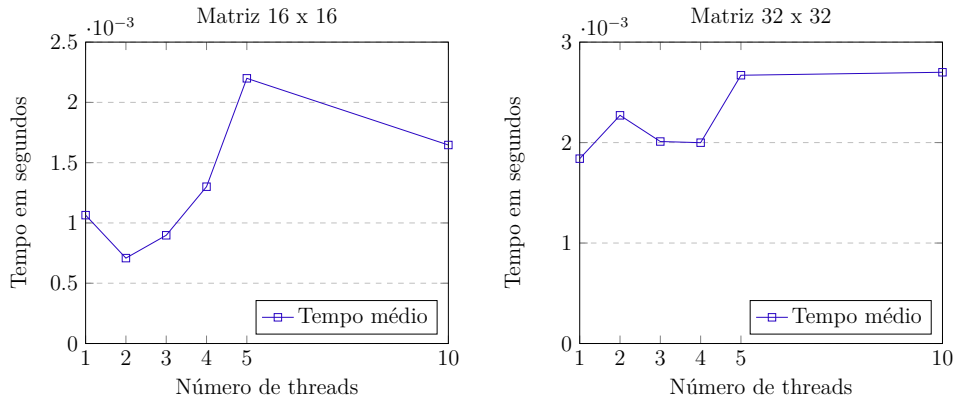

Figura 3 – Matrizes 16x16 e 32x32

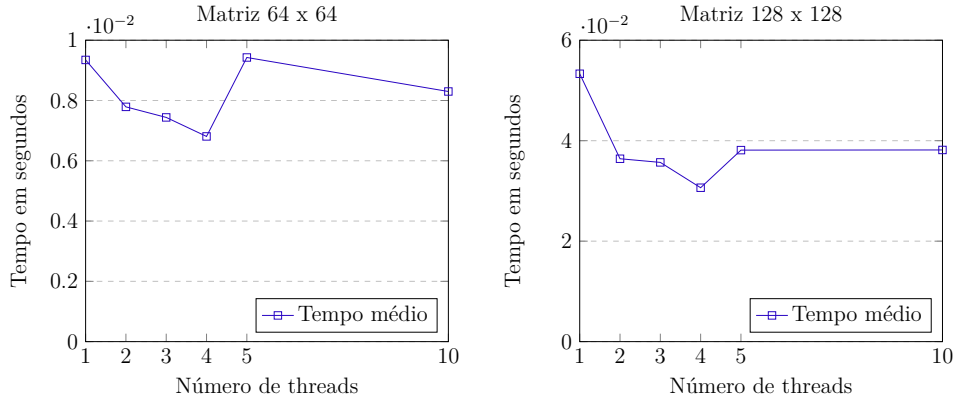

Figura 4 – Matrizes 64x64 e 128x128

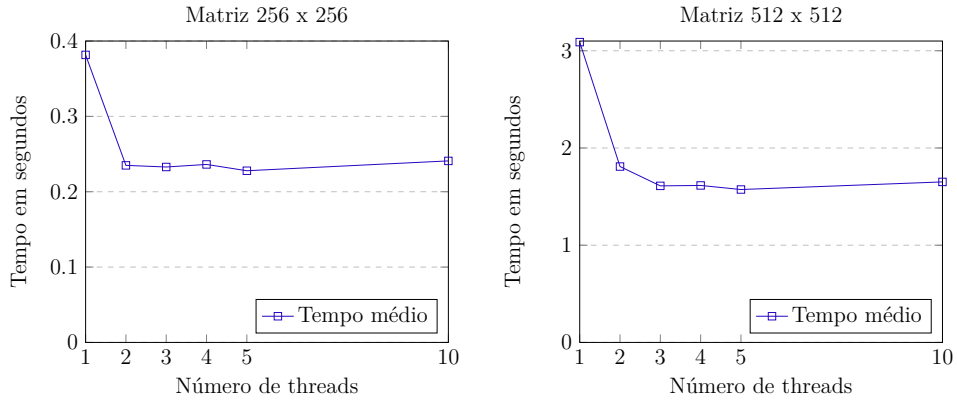

Figura 5 – Matrizes 256x256 e 512x512

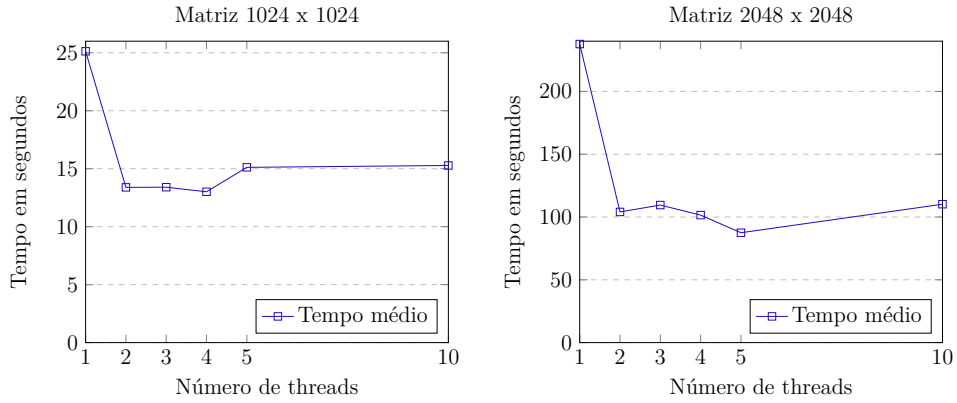

Figura 6 – Matrizes 1024x1024 e 2048x2048

Considerações finais

Conclui-se que o uso de um mecanismo de paralelismo para multiplicações de matrizes com *threads* é realmente eficaz, porém essas percepções são mais bem visiveis para matrizes de grandes dimenções, como: $\mathbf{512^2}$, $\mathbf{1024^2}$ e $\mathbf{2048^2}$. Apesar de ser perceptivel a redução de tempo no uso de *threads*, como o foco do projeto é apenas o uso de threads, a utilização de *join*, pode ser um gargalo, onde os tempos poderiam ser ainda menor.

Então, como trabalho futuro, pretende-se realizar testes desse algoritmo utilizando estratégias de exclusão mutua, como *mutex*, *semáforos*, etc.