UNIVERSIDADE PRESBITERIANA MACKENZIE

Faculdade de Computação e Informática

FCI – Faculdade de Computação e Informática

Atividade - Multiplicação de Matrizes
Computação Paralela

Turma: 05P11

Luis Felipe Basacchi Darre - 10419477

UNIVERSIDADE PRESBITERIANA MACKENZIE

Faculdade de Computação e Informática

1. **Crie dois executáveis para a parte de multiplicação de matrizes: um que percorre em ordem de linha e outro que percorre em ordem de coluna.**
   Faça agora a multiplicação de matrizes considerando:
2. A∗B=C
3. *A*∗*B*=*C*, onde:
4. Dimensões de A: M x N
5. Dimensões de B: N x X (qq dimensão razoável, mas não um vetor)
6. Consequentemente, dimensões de C: M x X

se quiser, pode utilizar matrizes quadradas.

Multiplicação por colunas:

```
#include <iostream>

#include <vector>

#include <pthread.h>



const int M = 4; // Número de linhas de A

const int N = 4; // Número de colunas de A e linhas de B

const int X = 4; // Número de colunas de B



int A[M][N], B[N][X], C[M][X];

int thread_count;



void* multiply_columns(void* rank) {

    long my_rank = (long)rank;

    int local_x = X / thread_count;
```

```
    int my_first_col = my_rank * local_x;
    int my_last_col = (my_rank + 1) * local_x - 1;

    for (int j = my_first_col; j <= my_last_col; j++) {
        for (int i = 0; i < M; i++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return NULL;
}

int main(int argc, char* argv[]) {
    pthread_t* thread_handles;
    long thread;

    // Inicializa as matrizes A e B
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = 1;

    for (int i = 0; i < N; i++)
```

```
        for (int j = 0; j < X; j++)
            B[i][j] = 2;

    thread_count = strtol(argv[1], NULL, 10);
    thread_handles = new pthread_t[thread_count];

    for (thread = 0; thread < thread_count; thread++) {
        pthread_create(&thread_handles[thread], NULL, multiply_columns,
(void*)thread);
    }

    for (thread = 0; thread < thread_count; thread++) {
        pthread_join(thread_handles[thread], NULL);
    }

    // Exibe o resultado
    std::cout << "Resultado da Multiplicação (Ordem de Coluna):\n";
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < X; j++) {
            std::cout << C[i][j] << " ";
        }
        std::cout << "\n";
    }
```

```
    delete[] thread_handles;

    return 0;

}
```

Execução com matriz 4x4: A = {1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1}, B = {2,2,2,2},{2,2,2,2},{2,2,2,2},{2,2,2,2}

```
@LuBasacchi →/workspaces/comppar-05p-threads-cpp-LuBasacchi (main) $ ./ordem_coluna 2
Resultado da Multiplicação (Ordem de Coluna):
8 8 8 8
8 8 8 8
8 8 8 8
8 8 8 8
```

Multiplicação por linhas:

```
#include <iostream>
#include <vector>
#include <pthread.h>

const int M = 4; // Número de linhas de A
const int N = 4; // Número de colunas de A e linhas de B
const int X = 4; // Número de colunas de B

int A[M][N], B[N][X], C[M][X];
int thread_count;

void* multiply_rows(void* rank) {
    long my_rank = (long)rank;
    int local_m = M / thread_count;
    int my_first_row = my_rank * local_m;
    int my_last_row = (my_rank + 1) * local_m - 1;

    for (int i = my_first_row; i <= my_last_row; i++) {
        for (int j = 0; j < X; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
```

```
    return NULL;
}

int main(int argc, char* argv[]) {
    pthread_t* thread_handles;
    long thread;

    // Inicializa as matrizes A e B
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = 1;

    for (int i = 0; i < N; i++)
        for (int j = 0; j < X; j++)
            B[i][j] = 2;

    thread_count = strtol(argv[1], NULL, 10);
    thread_handles = new pthread_t[thread_count];

    for (thread = 0; thread < thread_count; thread++) {
        pthread_create(&thread_handles[thread], NULL, multiply_rows,
(void*)thread);
    }

    for (thread = 0; thread < thread_count; thread++) {
        pthread_join(thread_handles[thread], NULL);
    }

    // Exibe o resultado
    std::cout << "Resultado da Multiplicação (Ordem de Linha):\n";
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < X; j++) {
            std::cout << C[i][j] << " ";
        }
        std::cout << "\n";
    }

    delete[] thread_handles;
    return 0;
}
```

Execução com matriz 4x4: A = {1,1,1,1},{1,1,1,1},{1,1,1,1},{1,1,1,1}, B = {2,2,2,2},{2,2,2,2},{2,2,2,2},{2,2,2,2}

```
@LuBasacchi →/workspaces/comppar-05p-threads-cpp-LuBasacchi (main) $ ./ordem_linha 2
Resultado da Multiplicação (Ordem de Linha):
8 8 8 8
8 8 8 8
8 8 8 8
8 8 8 8
```

2. **Crie um terceiro executável para utilizar corretamente o cache (hierarquia de memória)**:
Para utilizar corretamente o cache (L1, L2) utilizando a abordagem de **blocagem** apresentada no artigo da Intel.
Você deve fazer a análise da velocidade do seu algoritmo compilando o código da seguinte forma:
a. desligando todas as otimizações do compilador, como indicado acima.

b. ligando a otimização máxima.

Para cada opção, você deve medir o tempo de execução, seguindo o exemplo mostrado nos trechos de código acima.

Multiplicação em Blocos:

```cpp
#include <iostream>
#include <vector>
#include <pthread.h>
#include <chrono> // Para medir tempo

const int M = 512; // Número de linhas de A
const int N = 512; // Número de colunas de A e linhas de B
const int X = 512; // Número de colunas de B
const int BLOCK_SIZE = 64; // Tamanho do bloco
```

```
int A[M][N], B[N][X], C[M][X]; // Matrizes globais
int thread_count; // Número de threads

void* multiply_blocks(void* rank) {
    long my_rank = (long)rank;
    int rows_per_thread = M / thread_count;
    int my_start_row = my_rank * rows_per_thread;
    int my_end_row = (my_rank + 1) * rows_per_thread - 1;

    // Blocagem para otimizar o uso do cache
    for (int i = my_start_row; i <= my_end_row; i += BLOCK_SIZE) {
        for (int j = 0; j < X; j += BLOCK_SIZE) {
            for (int k = 0; k < N; k += BLOCK_SIZE) {
                for (int ii = i; ii < std::min(i + BLOCK_SIZE, M);
++ii) {
                    for (int jj = j; jj < std::min(j + BLOCK_SIZE, X);
++jj) {
                        for (int kk = k; kk < std::min(k + BLOCK_SIZE,
N); ++kk) {
                            C[ii][jj] += A[ii][kk] * B[kk][jj];
                        }
                    }
                }
            }
        }
    }
    return NULL;
}

int main(int argc, char* argv[]) {
    pthread_t* thread_handles;
    long thread;

    // Inicializando as matrizes A e B
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = 1; // Exemplo: todos elementos de A são 1

    for (int i = 0; i < N; i++)
        for (int j = 0; j < X; j++)
```

```
            B[i][j] = 2; // Exemplo: todos elementos de B são 2

    thread_count = strtol(argv[1], NULL, 10);
    thread_handles = new pthread_t[thread_count];

    auto start = std::chrono::high_resolution_clock::now(); // Começar
cronômetro

    // Criar threads
    for (thread = 0; thread < thread_count; thread++) {
        pthread_create(&thread_handles[thread], NULL, multiply_blocks,
(void*)thread);
    }

    // Esperar threads terminarem
    for (thread = 0; thread < thread_count; thread++) {
        pthread_join(thread_handles[thread], NULL);
    }

    auto end = std::chrono::high_resolution_clock::now(); // Fim do
cronômetro
    std::chrono::duration<double> elapsed = end - start;

    std::cout << "Tempo de execução: " << elapsed.count() << "
segundos\n";

    // Exibir parte do resultado (opcional, já que o output pode ser
muito grande)
    std::cout << "Resultado (amostra):\n";
    for (int i = 0; i < std::min(10, M); i++) {
        for (int j = 0; j < std::min(10, X); j++) {
            std::cout << C[i][j] << " ";
        }
        std::cout << "\n";
    }

    delete[] thread_handles;
    return 0;
}
```

Execução para matrizes 512x512 (Sem otimização):

```
● @LuBasacchi ➜/workspaces/comppar-05p-threads-cpp-LuBasacchi (main) $ ./mat_mult_blocagem 4
  Tempo de execução: 0.581643 segundos
  Resultado (amostra):
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
```

Execução para matrizes 512x512 (Com otimização):

```
● @LuBasacchi ➜/workspaces/comppar-05p-threads-cpp-LuBasacchi (main) $ ./mat_mult_blocagem_optim 4
  Tempo de execução: 0.144354 segundos
  Resultado (amostra):
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
  1024 1024 1024 1024 1024 1024 1024 1024 1024 1024
```

3. Valgrind

Faça também uma análise do padrão de acesso ao cache de todas as versões utilizando o utilitário valgrind. Veja abaixo um exemplo de saída que o valgrind dá:

UNIVERSIDADE PRESBITERIANA MACKENZIE

Faculdade de Computação e Informática

```
@LuBasacchi ➜/workspaces/comppar-05p-threads-cpp-LuBasacchi (main) $ valgrind --tool=cachegrind ./mat_mult_blocagem
==4463== Cachegrind, a cache and branch-prediction profiler
==4463== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==4463== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4463== Command: ./mat_mult_blocagem
==4463==
--4463-- warning: L3 cache found, using its data for the LL simulation.
==4463==
==4463== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==4463==  Access not within mapped region at address 0x0
==4463==    at 0x4A8EB3A: ____strtol_l_internal (strtol_l.c:292)
==4463==    by 0x1095C4: main (mat_mult_blocagem.cpp:50)
==4463==  If you believe this happened as a result of a stack
==4463==  overflow in your program's main thread (unlikely but
==4463==  possible), you can try to increase the size of the
==4463==  main thread stack using the --main-stacksize= flag.
==4463==  The main thread stack size used in this run was 8388608.
==4463==
==4463== I   refs:      9,257,235
==4463== I1  misses:        1,623
==4463== LLi misses:        1,588
==4463== I1  miss rate:      0.02%
==4463== LLi miss rate:      0.02%
==4463==
==4463== D   refs:      3,372,257  (2,653,942 rd   + 718,315 wr)
==4463== D1  misses:       49,836  (   14,526 rd   +  35,310 wr)
==4463== LLd misses:       42,876  (    8,467 rd   +  34,409 wr)
==4463== D1  miss rate:       1.5% (      0.5%     +     4.9%  )
==4463== LLd miss rate:       1.3% (      0.3%     +     4.8%  )
==4463==
==4463== LL refs:          51,459  (   16,149 rd   +  35,310 wr)
==4463== LL misses:        44,464  (   10,055 rd   +  34,409 wr)
==4463== LL miss rate:        0.4% (      0.1%     +     4.8%  )
```

```
@LuBasacchi ➜/workspaces/comppar-05p-threads-cpp-LuBasacchi (main) $ valgrind --tool=cachegrind ./mat_mult_blocagem_optim
==4616== Cachegrind, a cache and branch-prediction profiler
==4616== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==4616== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4616== Command: ./mat_mult_blocagem_optim
==4616==
--4616-- warning: L3 cache found, using its data for the LL simulation.
==4616==
==4616== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==4616==  Access not within mapped region at address 0x0
==4616==    at 0x4A8EB3A: ____strtol_l_internal (strtol_l.c:292)
==4616==    by 0x10929E: main (mat_mult_blocagem.cpp:50)
==4616==  If you believe this happened as a result of a stack
==4616==  overflow in your program's main thread (unlikely but
==4616==  possible), you can try to increase the size of the
==4616==  main thread stack using the --main-stacksize= flag.
==4616==  The main thread stack size used in this run was 8388608.
==4616==
==4616== I   refs:      2,966,358
==4616== I1  misses:        1,625
==4616== LLi misses:        1,590
==4616== I1  miss rate:      0.05%
==4616== LLi miss rate:      0.05%
==4616==
==4616== D   refs:        877,841  (553,782 rd   + 324,059 wr)
==4616== D1  misses:       49,844  ( 14,534 rd   +  35,310 wr)
==4616== LLd misses:       42,881  (  8,472 rd   +  34,409 wr)
==4616== D1  miss rate:       5.7% (    2.6%     +    10.9%  )
==4616== LLd miss rate:       4.9% (    1.5%     +    10.6%  )
==4616==
==4616== LL refs:          51,469  ( 16,159 rd   +  35,310 wr)
==4616== LL misses:        44,471  ( 10,062 rd   +  34,409 wr)
==4616== LL miss rate:        1.2% (    0.3%     +    10.6%  )
```

Análise:

Pontos de comparação

UNIVERSIDADE PRESBITERIANA MACKENZIE

Faculdade de Computação e Informática

Métrica	mat_mult_blocagem	mat_mult_blocagem_optim	Diferença
Instruções Executadas (I refs)	9,257,235	2,966,358	Redução significativa
I1 Misses (L1 Instr.)	1,623	1,625	Quase idêntico
LLi Misses (Last-Level Cache - Instruções)	1,588	1,590	Quase idêntico
D Refs (Acessos a Dados)	3,372,257	877,841	Redução significativa
D1 Misses (L1 Data Cache)	49,836	49,844	Quase igual
LLd Misses (Last-Level Cache - Dados)	42,876	42,881	Pequena diferença
LL Misses (Total)	44,464	44,471	Quase igual

A versão otimizada executou significativamente menos instruções.
Os acessos à memória (D Refs) foram muito reduzidos, indicando melhor uso do cache.