City University of Hong Kong

CS3103

Operating Systems

Semester A (2022-2023)

Group Project: Parallel Matrix Multiplication

Group: 04

Group Members:

Zhan Yipeng 56826717

Cheung Pui Kwan 56665686

CHU Lok Cheong 56866320

Outline:

- **Background Information**
- **Introduction**
- **Threads Design:**
- **Explanation**
- **Result**
- **Source Code**
- **Division of work:**

Student	**Work**
Zhan Yipeng	Matrix multiplication, Thread Design Report
Cheung Pui Kwan	Matrix multiplication, Thread Design, Report
CHU Lok Cheong	Research, Report

Background Information

Matrix multiplication is a binary operation that creates a matrix from two matrices in mathematics, especially in linear algebra. The first matrix's columns must have the same number of rows as the second matrix's rows for matrices to be multiplied. The first matrix's number of rows and the second matrix's number of columns are combined to form the final matrix, or the matrix product. The letters AB stand for the result of the matrices A and B.

If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

the *matrix product* $\mathbf{C} = \mathbf{AB}$ (denoted without multiplication signs or dots) is defined to be the $m \times p$ matrix[5][6][7][8]

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, ..., m$ and $j = 1, ..., p$.

Introduction

In this project, we are required to parallelize the matrix multiplication, determine the number of threads to use and balance the efficiency of different threads. Also,

we have to choose which algorithm to handle the multiplication with better time complexity , space complexity , correctness and performance.

There are few matrix multiplication algorithm in mathematics:

1. The Naive Matrix Multiplication Algorithm

Algorithm 1: The Naive Matrix Multiplication Algorithm

```
Data: S[A][B], P[G][H]
Result: Q[][]
if B == G then
    for m = 0; m < A; m++ do
        for r = 0; r < H; r++ do
            Q[m][r] = 0;
            for k = 0; k < G; k++ do
                Q[m][r] += S[m][k] * P[k][r];
            end
        end
    end
end
```

Three nested loops make up the simple matrix multiplication algorithm. The total number of runs in the inner loops would be equal to the length of the matrix for each iteration of the outer loop. The time complexity is O(n^3)

2. The Solvay Strassen Algorithm

   The Stassen algorithm uses a divide-and-conquer strategy and a clever math trick to quickly solve the matrix multiplication problem, unlike the

naive method, which employs an exhaustive approach.The time complexity is O(n^2.8074).

In short, we use the The Naive Matrix Multiplication Algorithm which is a brute force manner.

Threads Design:

Four threads are sufficient to complete the task in the allotted time, since larger threads do not perform better. Create new threads and Using pthread join, the main thread then waits for the created thread to terminate. The * multiply() function, which computes the matrix multiplication, is then changed to fit the anticipated prototype of the pthread create call. The range of k in * multiply() that

each thread operates on is distributed equally among the threads because we used 4threads.

We initialized thread args for each thread in accordance on the main thread. Once all threads had been created, we called pthread join in a separate loop, so calling pthread join immediately after pthread create will not make the main thread wait before creating the next thread. Every thread is adding to every location in matrixC. Utilizing synchronization primitives like lock, mutex, and semaphore will help keep the correctness.

Finally, a global lock variable's overhead for locking and unlocking is also added. Each thread should carry out parallel local addition before synchronizing the last updates.

Explanation

There are 4 functions in our program.

1. Init()

```
2.  void init()
3.  {
4.    scanf("%d%d%d", &n, &p, &m);
5.
6.    for (size_t i = 0; i < n; ++i)
7.    {
8.      for (size_t j = 0; j < p; ++j)
9.      {
10.       scanf("%i", &matrixA[i][j]);
11.     }
12.   }
13.
14.   for (size_t i = 0; i < p; ++i)
15.   {
16.     for (size_t j = 0; j < m; j++)
17.     {
18.       scanf("%i", &matrixB[i][j]);
19.     }
20.   }
21. }
```

This function is used to input the number from test case in to matrixA and martixB. The time complexity is O(n^2)

2. multiply()

```
void *multiply(void *arg)
{
  struct thread_args *parsed_args = (struct thread_args *)arg;
  for (size_t i = 0; i < n; ++i)
  {
    for (size_t j = 0; j < p; ++j)
    {
      for (size_t k = parsed_args->from; k < parsed_args->to; ++k)
      {
        matrixC[i][k] += matrixA[i][j] * matrixB[j][k];
      }
    }
```

```
 }
}
```

This function is used to perform the matrix multiplication with the Naive Matrix Multiplication Algorithm in the thread manner. We basically assigned the start number and end number for each thread so they can have their own job to work. The time complexity is O(n^3)

3. printResult()

```
void printResult()
{
  for (size_t i = 0; i < n; ++i)
  {
    for (size_t j = 0; j < m; ++j)
    {
      printf("%i ", matrixC[i][j]);
    }
  }
}
```

This function is simply print the result to the output terminal. The time complexity is O(n^2)

Result

```
pkcheung44@ubt20a:~/cs3103-pmm-2022$ make test
TEST 0 - clean build (program should compile without errors or warnings)
Test finished in 0.149 seconds
RESULT passed

TEST 1 - toy matrix multiplication (smaller than 10 * 10, 1 sec timeout)
Test finished in 0.011 seconds
RESULT passed

TEST 2 - small matrix multiplication 1 (smaller than 200 * 200, 1 sec timeout)
Test finished in 0.021 seconds
RESULT passed

TEST 3 - small matrix multiplication 2 (smaller than 400 * 400, 1 sec timeout)
Test finished in 0.050 seconds
RESULT passed

TEST 4 - small matrix multiplication 3 (smaller than 600 * 600, 1 sec timeout)
Test finished in 0.128 seconds
RESULT passed

TEST 5 - small matrix multiplication 4 (smaller than 800 * 800, 1 sec timeout)
Test finished in 0.304 seconds
RESULT passed

TEST 6 - large matrix multiplication 1 (smaller than 1000 * 1000, 5 sec timeout)
Test finished in 0.468 seconds
RESULT passed

TEST 7 - large matrix multiplication 2 (smaller than 1200 * 1200, 15 sec timeout)
Test finished in 0.799 seconds
RESULT passed

TEST 8 - large matrix multiplication 3 (smaller than 1400 * 1400, 25 sec timeout)
Test finished in 1.149 seconds
RESULT passed

TEST 9 - large matrix multiplication 4 (smaller than 1600 * 1600, 40 sec timeout)
Test finished in 1.712 seconds
RESULT passed

TEST 10 - large matrix multiplication 5 (smaller than 2000 * 2000, 80 sec timeout)
Test finished in 2.992 seconds
RESULT passed
```

Source Code

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

```c
#include <stdio.h>

#define MATRIX_SIZE 2000
#define THREADS_NUM 4

int matrixA[MATRIX_SIZE][MATRIX_SIZE];
int matrixB[MATRIX_SIZE][MATRIX_SIZE];
int matrixC[MATRIX_SIZE][MATRIX_SIZE];
int n, p, m;

struct thread_args
{
  int from;
  int to;
};

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void init()
{
  scanf("%d%d%d", &n, &p, &m);

  for (size_t i = 0; i < n; ++i)
  {
    for (size_t j = 0; j < p; ++j)
    {
      scanf("%i", &matrixA[i][j]);
    }
  }

  for (size_t i = 0; i < p; ++i)
  {
    for (size_t j = 0; j < m; j++)
    {
      scanf("%i", &matrixB[i][j]);
    }
  }
}

void *multiply(void *arg)
```

```
{
  struct thread_args *parsed_args = (struct thread_args *)arg;
  for (size_t i = 0; i < n; ++i)
  {
    for (size_t j = 0; j < p; ++j)
    {
      for (size_t k = parsed_args->from; k < parsed_args->to; ++k)
      {
        matrixC[i][k] += matrixA[i][j] * matrixB[j][k];
      }
    }
  }
}

void printResult()
{
  for (size_t i = 0; i < n; ++i)
  {
    for (size_t j = 0; j < m; ++j)
    {
      printf("%i ", matrixC[i][j]);
    }
  }
}

int main(int argc, char *argv[])
{
  init();

  pthread_t threads[THREADS_NUM];
  struct thread_args args[THREADS_NUM];
  int current_from = 0, chunk_size = m / THREADS_NUM;

  for (size_t i = 0; i < THREADS_NUM; ++i)
  {
    args[i].from = current_from;
    args[i].to = current_from + chunk_size;
    current_from += chunk_size;
  }
  args[THREADS_NUM - 1].to = m;
```

```
  for (size_t i = 0; i < THREADS_NUM; ++i)
  {
    pthread_create(&threads[i], NULL, multiply, &args[i]);
  }
  for (size_t i = 0; i < THREADS_NUM; ++i)
  {
    pthread_join(threads[i], NULL);
  }

  printResult();

  return 0;
}
```