**x`МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №6**

**по дисциплине «Параллельные алгоритмы»**

**Тема**: **Умножение матриц**

|  |  |  |
| --- | --- | --- |
| Студент гр. 1384 |  | Усачева Д. В. |
| Преподаватель |  | Татаринов Ю. С. |

Санкт-Петербург

2023

**Цель**

Цель работы заключается в изучении работы с виртуальными топологиями, их управлением и функциями в библиотеке MPI.

## Задание

Вариант 4. Блочный алгоритм Фокса.

Выполнить задачу умножения двух квадратных матриц A и B размера m × m, результат записать в матрицу C. Реализовать последовательный и параллельный алгоритм, одним из перечисленных ниже способов и провести анализ полученных результатов. Выбор параллельного алгоритма определяется индивидуальным номером задания. Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам.

Непараллельный алгоритм умножения матриц (должен быть реализован во всех вариантах на одном процессе).

# Выполнение работы

Алгоритм Фокса.

Предполагается, что все матрицы являются квадратными размера m×m, количество блоков по горизонтали и вертикали являются одинаковым и равным q (т.е. размер всех блоков равен k×k, k=n/q).

При блочном разбиении данных для определения базовых подзадач естественным представляется взять за основу вычисления, выполняемые над матричными блоками. С учетом сказанного определим базовую подзадачу как процедуру вычисления всех элементов одного из блоков матрицы С.

Итак, за основу параллельных вычислений для матричного умножения при блочном разделении данных принят подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы C и при этом в подзадачах на каждой итерации расчетов располагается только по одному блоку исходных матриц A и B. Для нумерации подзадач будем использовать индексы размещаемых в подзадачах блоков матрицы C, т.е. подзадача (i,j) отвечает за вычисление блока Cij – тем самым, набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы C.

В соответствии с алгоритмом Фокса в ходе вычислений на каждом процессе с условными координатами в квадратной сетке (i, j) располагается четыре матричных блока:

− блок Cij матрицы C, вычисляемый подзадачей;

− блок Aij матрицы A, размещаемый в подзадаче перед началом вычислений;

− блоки A\_ij, B\_ij матриц A и B, получаемые подзадачей в ходе выполнения вычислений.

Выполнение параллельного метода включает:

• этап инициализации, на котором каждом процессе (i, j) передаются блоки Aij, Bij и обнуляются блоки Cij;

• этап вычислений, в рамках которого на каждой итерации l, 0≤l <q, осуществляются следующие операции:

− для каждой строки i, 0≤ i <q, блок Aij процесса (i, j) пересылается на все процессы той же строки i решетки;

− полученные в результаты пересылок блоки A\_ij, B\_ij каждого процесса (i, j) перемножаются и прибавляются к блоку Cij.

− блоки B'ij каждого процесса (i, j) пересылаются подзадачам, являющимися соседями сверху в столбцах решетки процессов (блоки процессов из первой строки решетки пересылаются процессам последней строки решетки).

Для выполнения поставленной задачи написано две программы на языке C, код блочный алгоритм Фокса представлен ниже в листинге 1, а код непараллельный алгоритма умножения матриц в листинге 2.

Для реализации непараллельного алгоритма реализованы функции ввода (предусмотрен ввод матрицы через консоль, по умолчанию матрица заполняется 1), вывода и умножения матриц. В теле главной функции происходит выделение памяти для массивов заданного пользователем размера, вызываются необходимые функции. В конце программы происходит освобождение выделенной памяти.

Для реализации блочного алгоритма Фокса так же были реализованы функции ввода, вывода, умножения матриц и функция взятия числа по модулю (использовалась для отрицательных чисел). Функция ввода сохраняет полученную матрицу во временную переменную и преобразует матрицу в формат удобный для пересылки (блочный) между процессами. Для наглядности преобразования приведем пример:

0 1 2 3 4 5 0 1 2 | 3 4 5

6 7 8 9 10 11 6 7 8 | 9 10 11

12 13 14 15 16 17 (кол-во процессов = 9)12 13 14| 15 16 17

18 19 20 21 22 23 🡺 18 19 20| 21 22 23

24 25 26 27 28 29 24 25 26| 27 28 29

30 31 32 33 34 35 30 31 32| 33 34 35

🡺 0 1 2 4 7 8 12 13 14 3 4 5 9 10 11 15 16 17 18 19 20 24 25 26 30 31 32 21 22 23 27 28 29 33 34 35. Далее в таком виде будут храниться матрицы A, B, и C.

Функция вывода реализована для вывода матриц блочного формата.

Функция умножения представляет собой функцию для перемножения блоков (в отличие от обычной функции перемножения матриц, эта функция суммирует результат ранее перемноженных блоков с нынешними).

Алгоритм реализован без использования виртуальных топологий, но в отчете представлены моменты программы, в которых мы могли бы их использовать.

Для наилучшей работы алгоритма, считаем, что к­оличество процессов –­­ полный квадрат. Размерность матрицы должна нацело делиться на корень из количества процессов. Сначала происходит этап инициализации – отправка необходимых блоков матрицы каждому процессу (похоже на топологию типа звезда). Далее идет нулевой шаг алгоритма, он находится вне цикла, так как нет необходимости в пересылке матрицы B\_ij, а также в нем определяются номера процессов, содержащих диагональные блоки матрицы. Отправка блоков матрицы А в каждой строке идет слева направо, начиная с диагонали. Процесс рассылает свой блок Aij всем остальным процессам строки (похоже на топологию типа звезда, но есть необходимость на каждом шаге алгоритма менять центральный элемент). После в цикле реализованы остальные шаги алгоритма. Сначала происходит пересылка матрицы B\_ij, на каждом шаге процесс пересылает своему соседу сверху (похоже на топологию типа кольцо). Далее пересылка Aij и их перемножение.

После выполнения алгоритма при помощи коллективной операции MPI\_Gather нулевой процесс собирает все блоки матрицы С и выводит результат. В конце программы происходит освобождение выделенной памяти.

Ниже представлена сеть Петри основной части алгоритма (см. рис 1).

Рисунок 1 — Сеть Петри основной части алгоритма

Листинг 1 — Код программы lab6\_1.c

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <mpi.h>**

**#include <math.h>**

**void printRes(int\* matrix, int m, int k, int q) {**

**printf("\nРезультат умножения матриц:\n");**

**for (int i = 0; i < q ; i ++) { //строка по блокам**

**for (int l = 0; l < k; l++) { // строка по элементам блока**

**for (int j = 0; j < q; j++) { // столбец по блокам**

**for (int n = 0; n < k; n++) { // столбец по элементам блока**

**printf("%d ",matrix[n + l \* k + j \* k \* k + i \* k \* k \* q]);**

**}**

**}**

**printf("\n");**

**}**

**}**

**}**

**void inputMatrix(int\* matrix, int m, int procNum, int k){**

**int\*\* data = (int\*\*)malloc(m \* sizeof(int\*));**

**for (int i = 0; i < m; i++) {**

**data[i] = (int\*)malloc(m \* sizeof(int\*));**

**}**

**if (m <= 10) printf("Введите элементы матрицы:\n");**

**for (int i = 0; i < m; i++) {**

**for (int j = 0; j < m; j++) {**

**if (m <= 10)**

**scanf("%d", &data[i][j]);**

**else**

**data[i][j] = i \* m + j;**

**}**

**}**

**int count = 0;**

**int column = 0;**

**int row = 0;**

**for (int num = 0; num < procNum; num++){**

**for (int i = 0; i < k; i++){**

**for (int j = 0; j < k; j++){**

**matrix[count] = data[i+row][j+column];**

**count++;**

**}**

**}**

**column += k;**

**if (column == m){**

**column = 0;**

**row += k;**

**}**

**}**

**for (int i = 0; i < m; i++) {**

**free(data[i]);**

**}**

**free(data);**

**}**

**void multiply(int\* matrix1, int\* matrix2, int\* result, int m) {**

**int row;**

**int column;**

**for (int i = 0; i < m \* m; i++) {**

**column = i % m;**

**for (int j = 0; j < m; j++) {**

**if (i % m == 0 ) row = i / m;**

**result[i]+= matrix1[j + m \* row] \* matrix2[column + j \*m];**

**}**

**}**

**}**

**int mod(int a, int mod) {**

**if (a < 0){**

**a += mod;**

**}**

**return a % mod;**

**}**

**int main(int argc, char\*\* argv) {**

**int m = 0;**

**int procRank, procNum;**

**int q, k;**

**int\* A;**

**int\* B;**

**int\* C;**

**double start;**

**MPI\_Init(&argc, &argv);**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &procRank);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &procNum);**

**q = sqrt(procNum);**

**if (q \* q != procNum) {**

**printf("Количество процессов должно быть полным квадратом.\n");**

**MPI\_Finalize();**

**return 0;**

**}**

**if (procRank == 0) {**

**printf("Размерность матриц должна нацело делиться на корень из количества процессов.\n");**

**while(m % q != 0 || m == 0) {**

**printf("Введите размерность матриц:\n");**

**scanf("%d", &m);**

**}**

**k = m / q ;**

**A = (int\*)malloc(k \* k \* procNum \* sizeof(int));**

**B = (int\*)malloc(k \* k \* procNum \* sizeof(int));**

**C = (int\*)malloc(k \* k \* procNum \* sizeof(int));**

**inputMatrix(A, m, procNum, k);**

**inputMatrix(B, m, procNum, k);**

**start = MPI\_Wtime();**

**for (int i = 1; i < procNum; i++) {**

**MPI\_Send(&m, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD);**

**MPI\_Send(&A[i \* k \* k], k \* k, MPI\_INT, i, 0, MPI\_COMM\_WORLD);**

**MPI\_Send(&B[i \* k \* k], k \* k, MPI\_INT, i, 0, MPI\_COMM\_WORLD);**

**}**

**}**

**else {**

**MPI\_Recv(&m, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**k = m / q ;**

**}**

**int\* Aij;**

**int\* A\_ij;**

**int\* B\_ij;**

**int\* Cij;**

**Aij = (int\*)malloc(k \* k \* sizeof(int));**

**A\_ij = (int\*)malloc(k \* k \* sizeof(int));**

**B\_ij = (int\*)malloc(k \* k \* sizeof(int));**

**Cij = (int\*)malloc(k \* k \* sizeof(int));**

**for (int i = 0; i < k \* k; i++) {**

**Cij[i] = 0;**

**}**

**if (procRank == 0) {**

**for (int i = 0; i < k \* k; i++) {**

**Aij[i] = A[i];**

**B\_ij[i] = B[i];**

**}**

**}**

**else{**

**MPI\_Recv(&Aij[0], k \* k, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**MPI\_Recv(&B\_ij[0], k \* k, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**}**

**int row = (procRank / q);**

**int column = (procRank % q);**

**int sendRankA;**

**int sendRankB = mod(procRank - q, procNum);**

**int recvRankB = (procRank + q) % procNum;**

**//0**

**if (row == column){**

**for (int i = 0; i < q ; i++) {**

**if (i != row){**

**MPI\_Send(&Aij[0], k \* k, MPI\_INT, row \* q + i, 0, MPI\_COMM\_WORLD);**

**}**

**}**

**multiply(Aij, B\_ij, Cij, k);**

**sendRankA = (procRank + 1) % q + row \*q;**

**}**

**else{**

**sendRankA = row \* q + row;**

**MPI\_Recv(&A\_ij[0], k \* k, MPI\_INT, sendRankA, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**multiply(A\_ij, B\_ij, Cij, k);**

**sendRankA = (sendRankA + 1) % q + row \*q;**

**}**

**for (int step = 1; step < q; step++) {//количество итераций**

**MPI\_Send(&B\_ij[0], k \* k, MPI\_INT, sendRankB, 0, MPI\_COMM\_WORLD);**

**MPI\_Recv(&B\_ij[0], k \* k, MPI\_INT, recvRankB, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**if (procRank == sendRankA){**

**for (int i = 0; i < q; i++) {**

**if (row \* q + i != procRank){**

**MPI\_Send(&Aij[0], k \* k, MPI\_INT, row \* q + i, 0, MPI\_COMM\_WORLD);**

**}**

**}**

**multiply(Aij, B\_ij, Cij, k);**

**sendRankA = (sendRankA + 1) % q + row \*q;**

**}**

**else{**

**MPI\_Recv(&A\_ij[0], k \* k, MPI\_INT, sendRankA, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);**

**multiply(A\_ij, B\_ij, Cij, k);**

**sendRankA = (sendRankA + 1) % q + row \*q;**

**}**

**}**

**MPI\_Gather(&Cij[0], k \* k, MPI\_INT, C, k \* k, MPI\_INT, 0, MPI\_COMM\_WORLD);**

**if ( procRank == 0) {**

**// printRes(C , m, k, q);**

**printf("Время работы программы: %f\n",MPI\_Wtime() - start);**

**free(A);**

**free(B);**

**free(C);**

**}**

**free(Aij);**

**free(A\_ij);**

**free(B\_ij);**

**free(Cij);**

**MPI\_Finalize();**

**return 0;**

**}**

**Ниже представлен вывод программы lab5.с**

Листинг 2 — Код программы lab6\_2.c

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <time.h>**

**void inputMatrix(int\*\* matrix, int m){**

**if (m <= 10) printf("Введите элементы матрицы:\n");**

**for (int i = 0; i < m; i++) {**

**for (int j = 0; j < m; j++) {**

**if (m <= 10) scanf("%d", &matrix[i][j]);**

**else matrix[i][j] = 1;**

**}**

**}**

**}**

**void multiply(int\*\* matrix1, int\*\* matrix2, int\*\* result, int m) {**

**for (int i = 0; i < m; i++) {**

**for (int j = 0; j < m; j++) {**

**result[i][j] = 0;**

**for (int k = 0; k < m; k++) {**

**result[i][j] += matrix1[i][k] \* matrix2[k][j];**

**}**

**}**

**}**

**}**

**void printMatrix(int\*\* matrix, int m) {**

**printf("Результат перемножения матриц:\n");**

**for (int i = 0; i < m; i++) {**

**for (int j = 0; j < m; j++) {**

**printf("%d ", matrix[i][j]);**

**}**

**printf("\n");**

**}**

**}**

**int main() {**

**clock\_t start;**

**int m;**

**printf("Введите размерность матриц:\n");**

**scanf("%d", &m);**

**int\*\* A = (int\*\*)malloc(m \* sizeof(int\*));**

**int\*\* B = (int\*\*)malloc(m \* sizeof(int\*));**

**int\*\* C = (int\*\*)malloc(m \* sizeof(int\*));**

**for (int i = 0; i < m; i++){**

**A[i] = (int\*)malloc(m \* sizeof(int));**

**B[i] = (int\*)malloc(m \* sizeof(int));**

**C[i] = (int\*)malloc(m \* sizeof(int));**

**}**

**inputMatrix(A, m);**

**inputMatrix(B, m);**

**start = clock();**

**multiply(A, B, C, m);**

**// printMatrix(C, m);**

**printf("Время работы программы: %f\n", (double)(clock() - start) / CLOCKS\_PER\_SEC);**

**for (int i = 0; i < m; i++) {**

**free(A[i]);**

**free(B[i]);**

**free(C[i]);**

**}**

**free(A);**

**free(B);**

**free(C);**

**return 0;**

**}**

**Листинг 3 — Вывод программы lab6\_1.c для 4, 16 и 64 процессов для матрицы 16\*16**

**Размерность матриц должна нацело делиться на корень из количества процессов.**

**Введите размерность матриц:**

**16**

**Время работы программы: 0.000853**

**Размерность матриц должна нацело делиться на корень из количества процессов.**

**Введите размерность матриц:**

**16**

**Время работы программы: 0.003475**

**Размерность матриц должна нацело делиться на корень из количества процессов.**

**Введите размерность матриц:**

**16**

**Время работы программы: 0.013235**

**Листинг 4 — Вывод программы lab6\_2.c для матрицы 16\*16**

**Введите размерность матриц:**

**16**

**Время работы программы: 0.000024**

Так как мы всегда передаем число, нет необходимости прослеживать зависимость времени работы от объема данных. Рассмотрим зависимость времени работы программы от количества процессов.

**Таблица 1 — Среднее время выполнения.**

|  |  |
| --- | --- |
| **Количество процессов** | **Среднее время на выполнение(мс)** |
| **5** | 2.002 |
| **7** | 2.152 |
| **9** | 3.186 |

**Ниже указаны графики зависимостей времени выполнения и ускорения (см. рисунки 2-3).**

**Рисунок 2 — График зависимости времени выполнения от числа процессов**

**Ускорение времени работы программы можно вычислить по формуле:**

Sp (𝑛) = 𝑇1(𝑛)/𝑇p(𝑛)

**Рисунок 3 — График зависимости ускорения от числа процессов**

**Выводы**

В ходе выполнения лабораторной работы были изучены и использованы функция для создания топологии графа MPI\_Graph\_create и функции для получения информации о вершинах соседях MPI\_Graph\_neighbors\_count (количество соседей) и MPI\_Graph\_neighbors (список соседей). Так как в интервал, заданный в условии, попадает всего три нечетных числа, полученные графики не позволяют сделать точный вывод о зависимости времени работы программы от количества процессов. Однако по графикам прослеживается увеличение времени работы программы, это связано с тем, что при большем количестве процессов увеличивается количество соседей нулевого процесса, с которыми он должен обменятся числами. Так же скорость работы программы замедляется из-за увеличения времени заполнения массивов index и edges.