Федеральное государственное автономное образовательное учреждение высшего образования Санкт-Петербургский политехнический университет Петра Великого

Физико-механический институт

Высшая школа прикладной математики и вычислительной физики

Лабораторная работа №4

**Параллелизация алгоритмов умножения матрицы на вектор и**

**перемножения матриц**

Выполнил:

Кучиев Д.Ю.

Преподаватель:

Абрамов А.Г.

Санкт-Петербург

2025

**Цель работы**

Освоить и с помощью технологии MPI реализовать алгоритмы распараллеливания

операции умножения матрицы на вектор и перемножения матриц большой размерности общего вида (плотных матриц) при использовании различных способов декомпозиции и обмена данными. Программные коды представлены в репозитории, а также в Приложении: https://github.com/MrDionisio/MPI\_CC

**Характеристики компьютера**

Количество ядер: 8 AMD Ryzen 6000 Series.

Объем ОЗУ: 16 Гб.

**Часть 1: Параллелизация операции умножения матрицы на вектор**

Разработать программу, параллелизующую средствами MPI операции умножения матрицы на вектор. Параллелизация в рамках MPI достигается путем разбиения исходной матрицы по строкам и разделением всех строк между процессами, которые в дальнейшем будут находить сумму покомпонентных произведений элементов строки на элементы вектора. Затем значения будут собираться в единый вектор в качестве итогового результата работы программы.

Результаты работы программы (Приложение А) представлены на рисунке 1, где видна корректность выполнения исходного кода. Затем проводились аналогичные вычисления, но на матрице размерами 20 000 x 20 000. Время выполнения программы для различного количества процессов приведено в таблице 1, как и расчет эффективности и ускорения.

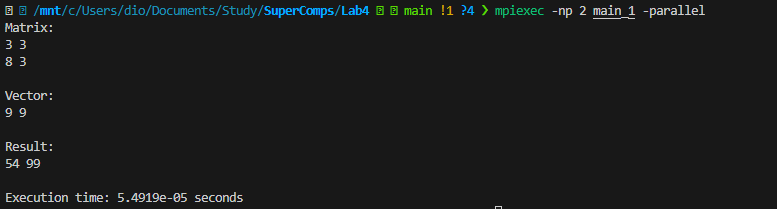


Рис. 1 Пример выполнения программы

Таблица 1. Расчет ускорения и эффективности работы программы.

|  |  |  |  |
| --- | --- | --- | --- |
| p | T | S | E |
| 1 | 1.98 | 1.00 | 1.00 |
| 2 | 1.08 | 1.83 | 0.92 |
| 3 | 0.66 | 3.00 | 1.00 |
| 4 | 0.53 | 3.74 | 0.93 |
| 5 | 0.47 | 4.21 | 0.84 |
| 6 | 0.369 | 5.37 | 0.89 |
| 7 | 0.33 | 6.00 | 0.86 |
| 8 | 0.3 | 6.60 | 0.83 |

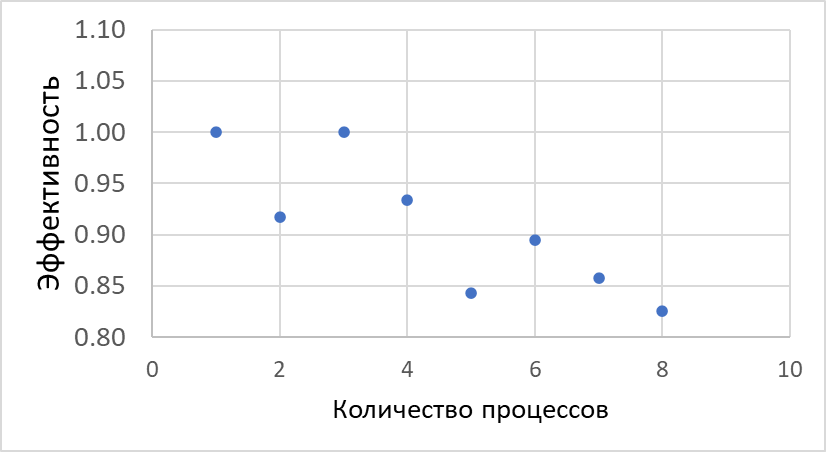
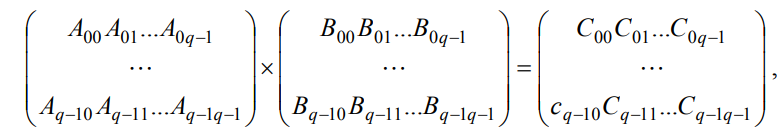


Рис. 2. Зависимость эффективности от количества процессов

Из рисунка 2 видно, что при увеличении процессов эффективность работы программы уменьшается. При увеличении количества процессов до 8, эффективность упала на 17%.

**Часть 2: Параллелизация операции перемножения матриц**

Используя параллельный алгоритм Фокса (Fox's algorithm) при блочной схеме разбиения матриц и циклическом распределении блоков между процессами, реализуем программный код на языке C++, который будет реализовывать операцию перемножения матриц. В алгоритме Фокса данные разделяются по блокам, размер которого изначально выбирается согласно максимальному числу процессов. То есть если имеется матрица *N* x *N* и P процессов, то каждому процессу будет передана матрица размерами x (*q* x *q*). Тогда операция умножения матриц в блочном виде:



Примерный вид выполнения программы (Приложение Б), а также иллюстрация корректности выполнения программы приведены на рисунке 3.

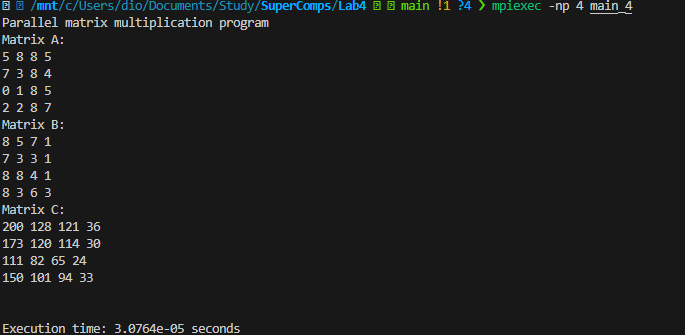


Рисунок 3. Пример выполнения программы операции умножения матриц методом Фокса

В данной работе использовалась матрица размера 2 000 х 2 000, так как было необходимо иметь такой размер матрицы, чтобы она ровно делилась по блокам в количестве q, а также количество процессов принимали значения P=1,4 так как необходимо, чтобы корень из количества нитей было целым числом. В таблице 2 приведено время работы, а также посчитанная эффективность программы. Видно, что эффективность при увеличении ядер до 4 уменьшилась всего на 8%.

Таблица 2. Расчет ускорения и эффективности работы программы операции умножения.

|  |  |  |  |
| --- | --- | --- | --- |
| p | T | S | E |
| 1 | 28.1 | 1.00 | 1.00 |
| 4 | 7.6 | 3.68 | 0.92 |

**Вывод:**

В ходе работы были реализованы параллельные алгоритмы позволяющие проводить различные операции с матрицами в рамках MPI подхода.

Полученные результаты подтверждают, что использование MPI технологии подходит для распараллеливаия вычислений, связанных с операциями над матрицами. На примере было продемонстрировано ускорение работы программы за счет использования распределения данных между процессами и проведения там локальных вычислений, которые в дальнейшем собирались воедино для дальнейшего использования.

Однако, из-за недостатка имеющихся рабочих ядер не было возможности провести исследования для операций умножения матрицы на матрицы, ввиду ограничения на то, что корень квадратный из числа процессов должен быть целым число, что сильно влияет на используемую архитектуру.

**Приложения**

**Приложение А. Операция умножения матрицы на вектор**

#include <mpi.h>

#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

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

    MPI\_Init(&argc, &argv);

    int rank, size;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

    const int N = 20000;  // Размер матрицы N x N и вектора N

    std::vector<int> matrix;

    std::vector<int> vector(N);

    std::vector<int> result;

    // Генерация данных в корневом процессе

    if (rank == 0) {

        matrix.resize(N \* N);

        std::srand(std::time(nullptr));

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

            matrix[i] = std::rand() % 9 + 1;

        for (int i = 0; i < N; ++i)

            vector[i] = std::rand() % 9 + 1;

    }

    // Замер времени выполнения (включая коммуникации)

    // Распространение вектора

    MPI\_Bcast(vector.data(), N, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // Распределение матрицы

    int rows\_per\_proc = N / size;

    int remainder = N % size;

    int local\_rows = rows\_per\_proc + (rank < remainder ? 1 : 0);

    int send\_counts[size], displs[size];

    for (int r = 0; r < size; ++r) {

        send\_counts[r] = (rows\_per\_proc + (r < remainder ? 1 : 0)) \* N;

        displs[r] = (r > 0) ? displs[r-1] + send\_counts[r-1] : 0;

    }

    std::vector<int> local\_matrix(local\_rows \* N);

    MPI\_Scatterv(matrix.data(), send\_counts, displs, MPI\_INT,

                 local\_matrix.data(), send\_counts[rank], MPI\_INT,

                 0, MPI\_COMM\_WORLD);

    double start\_time = MPI\_Wtime();

    // Локальное умножение

    std::vector<int> local\_result(local\_rows);

    for (int i = 0; i < local\_rows; ++i) {

        local\_result[i] = 0;

        for (int j = 0; j < N; ++j)

            local\_result[i] += local\_matrix[i \* N + j] \* vector[j];

    }

    // Сбор результатов

    int recv\_counts[size], offsets[size];

    for (int r = 0; r < size; ++r) {

        recv\_counts[r] = rows\_per\_proc + (r < remainder ? 1 : 0);

        offsets[r] = (r > 0) ? offsets[r-1] + recv\_counts[r-1] : 0;

    }

    double end\_time = MPI\_Wtime();

    if (rank == 0) result.resize(N);

    MPI\_Gatherv(local\_result.data(), local\_rows, MPI\_INT,

                result.data(), recv\_counts, offsets, MPI\_INT,

                0, MPI\_COMM\_WORLD);

    // Расчет и вывод времени

    double duration = end\_time - start\_time;

    double max\_duration;

    MPI\_Reduce(&duration, &max\_duration, 1, MPI\_DOUBLE, MPI\_MAX, 0, MPI\_COMM\_WORLD);

    // Вывод результатов

    if (rank == 0) {

        /\*

        std::cout << "Matrix:" << std::endl;

        for (int i = 0; i < N; ++i) {

            for (int j = 0; j < N; ++j)

                std::cout << matrix[i \* N + j] << " ";

            std::cout << std::endl;

        }

        std::cout << "\nVector:\n";

        for (int x : vector) std::cout << x << " ";

        std::cout << "\n\nResult:\n";

        for (int x : result) std::cout << x << " ";

        \*/

        std::cout << "\n\nExecution time: " << max\_duration << " seconds" << std::endl;

    }

    MPI\_Finalize();

    return 0;

}

**Приложение Б. Операция умножения матрицы на матрицу**

//Перемножеие матриц

// Алгоритм Фокса умножения матриц– блочное представление данных

//  Условия выполнения программы: все матрицы квадратные, размер блоков и их

//  количество по горизонтали и вертикали одинаковово, процессоры образуют

//  квадратную решетку

#include <mpi.h>

#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <cmath>

int ProcNum = 0;

int ProcRank = 0;

int GridSize; // Size of virtual processor grid

int GridCoords[2]; // координыты процессора в решетке

MPI\_Comm GridComm; // коммуникатор сетки

MPI\_Comm ColComm; // коммуникатор столбцов

MPI\_Comm RowComm; // коммуникатор строк

// функция заполнения матриц

void RandomDataInitialization (double\* pAMatrix, double\* pBMatrix, int Size) {

int i, j; // Loop variables

srand(unsigned(10));

for (i=0; i<Size; i++)

for (j=0; j<Size; j++) {

pAMatrix[i\*Size+j] = rand()%10;

pBMatrix[i\*Size+j] = rand()%10;

}

}

// собираем блоки в результат

void ResultCollection (double\* pCMatrix, double\* pCblock, int Size, int BlockSize) {

double \* pResultRow = new double [Size\*BlockSize];

for (int i=0; i<BlockSize; i++) {

/\*Функция MPI\_Gather производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root

&pCblock[i\*BlockSize] - адрес начала размещения посылаемых данных

BlockSize - число посылаемых элементов

&pResultRow[i\*Size] - адрес начала буфера приема

BlockSize - число элементов, получаемых от каждого процесса

0 - номер процессора получатедля\*/

        MPI\_Gather( &pCblock[i\*BlockSize], BlockSize, MPI\_DOUBLE, &pResultRow[i\*Size], BlockSize, MPI\_DOUBLE, 0, RowComm);

}

if (GridCoords[1] == 0) {

/\*Функция MPI\_Gather производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root\*/

    MPI\_Gather(pResultRow, BlockSize\*Size, MPI\_DOUBLE, pCMatrix, BlockSize\*Size, MPI\_DOUBLE, 0, ColComm);

}

delete [] pResultRow;

}

// Умножение матричных блоков

void BlockMultiplication(double \*pAblock, double \*pBblock, double \*pCblock, int BlockSize) {

// вычисление произведения матричных блоков

for (int i=0; i<BlockSize; i++) {

for (int j=0; j<BlockSize; j++) {

double temp = 0;

for (int k=0; k<BlockSize; k++ )

temp += pAblock [i\*BlockSize + k] \* pBblock [k\*BlockSize + j];

pCblock [i\*BlockSize + j] += temp;

}

}

}

/\*в начале каждой итерации iter алгоритма для каждой строки процессной решетки выбирается процесс, который будет рассылать свой блок матрицы А\*/

void ABlockCommunication(int iter, double \*pAblock, double\* pMatrixAblock, int BlockSize) {

// определяем ведущий процессор

int Pivot = (GridCoords[0] + iter) % GridSize;

// копируем передаваемый блок в буфер

if (GridCoords[1] == Pivot) {

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

pAblock[i] = pMatrixAblock[i];

}

// раскидываем блок

MPI\_Bcast(pAblock, BlockSize\*BlockSize, MPI\_DOUBLE, Pivot, RowComm);

}

// циклический сдвиг

void BblockCommunication(double \*pBblock, int BlockSize) {

MPI\_Status Status;

int NextProc = GridCoords[0] + 1;

if ( GridCoords[0] == GridSize-1 ) NextProc = 0;

int PrevProc = GridCoords[0] - 1;

if ( GridCoords[0] == 0 )

PrevProc = GridSize-1;

/\*обмен данными одного типа с замещением посылаемых данных на принимаемые

pBblock - адрес начала расположения посылаемого и принимаемого сообщени

BlockSize\*BlockSize - число передаваемых элементов

NextProc - номер процесса-получателя

0 - идентефикатор посылаемого сообщения

PrevProc - номер процессора отправителя

0 - идентефикатор принимаемого сообщения

\*/

MPI\_Sendrecv\_replace( pBblock, BlockSize\*BlockSize, MPI\_DOUBLE, NextProc, 0, PrevProc, 0, ColComm, &Status);

}

void ParallelResultCalculation(double\* pAblock, double\* pMatrixAblock, double\* pBblock, double\* pCblock, int BlockSize) {

for (int iter = 0; iter < GridSize; iter ++) {

/\*рассылка блока матрицы А по строке процессорной решетки\*/

ABlockCommunication (iter, pAblock, pMatrixAblock, BlockSize);

// умножение

BlockMultiplication(pAblock, pBblock, pCblock, BlockSize);

/\*циклический сдвиг блоков матрицы В вдоль столбца процессорной решетки\*/

BblockCommunication(pBblock, BlockSize);

}

}

// функция разложения матрицы на сетку

void CheckerboardMatrixScatter(double\* pMatrix, double\* pMatrixBlock, int Size, int BlockSize) {

double \* pMatrixRow = new double [BlockSize\*Size];

/\*делим матрицу горизонтальными полосами между процессорами нулевого столбца решетки\*/

if (GridCoords[1] == 0) {

/\*pMatrix - адрес начала размещения блоков распределяемых данных

BlockSize\*Size - число элементов, посылаемых каждому процессу

pMatrixRow - адрес приема

BlockSize\*Size - число получаемых элементов

0 - номер процессора отправителя

colComm - коммуникатор\*/

MPI\_Scatter(pMatrix, BlockSize\*Size, MPI\_DOUBLE, pMatrixRow, BlockSize\*Size, MPI\_DOUBLE, 0, ColComm);

}

/\*распределение каждой строки горизонтальной полосы матрицы вдоль строк процессорной решетки\*/

for (int i=0; i<BlockSize; i++) {

MPI\_Scatter(&pMatrixRow[i\*Size], BlockSize, MPI\_DOUBLE, &(pMatrixBlock[i\*BlockSize]), BlockSize, MPI\_DOUBLE, 0, RowComm);

}

delete [] pMatrixRow;

}

// функция распределения данных между процессорами

void DataDistribution(double\* pAMatrix, double\* pBMatrix, double\* pMatrixAblock, double\* pBblock, int Size, int BlockSize) {

CheckerboardMatrixScatter(pAMatrix, pMatrixAblock, Size, BlockSize);

CheckerboardMatrixScatter(pBMatrix, pBblock, Size, BlockSize);

}

// функция инициализации памяти и данных

void ProcessInitialization(double\* &pAMatrix, double\* &pBMatrix, double\* &pCMatrix, double\* &pAblock, double\* &pBblock, double\* &pCblock, double\* &pTemporaryAblock, int &Size, int &BlockSize ){

MPI\_Bcast(&Size, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

BlockSize = Size/GridSize;

pAblock = new double [BlockSize\*BlockSize];

pBblock = new double [BlockSize\*BlockSize];

pCblock = new double [BlockSize\*BlockSize];

pTemporaryAblock = new double [BlockSize\*BlockSize];

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

pCblock[i] = 0;

}

if (ProcRank == 0) {

pAMatrix = new double [Size\*Size];

pBMatrix = new double [Size\*Size];

pCMatrix = new double [Size\*Size];

RandomDataInitialization(pAMatrix, pBMatrix, Size);

}

}

/\*функция создает коммуникатор в виде двумерной квадратной решетки, определяет координаты каждого процесса в этой решетке\*/

void CreateGridCommunicators() {

int DimSize[2]; // число процессоров в каждом измерении сетки

int Periodic[2]; // =1, если размерность должна быть динамична

int Subdims[2]; // =1, если размерность должна быть фиксированна

DimSize[0] = GridSize;

DimSize[1] = GridSize;

Periodic[0] = 0;

Periodic[1] = 0;

// создание коммуникатора

/\*2 - число измерений

DimSize - -  массив, в котором задается число процессов вдоль каждого измерения

Periodic - логический массив размера ndims для задания граничных условий

1 - логическая переменная, указывает, производить перенумерацию процессов (true) или нет (false)

GridComm - новый коммуникатор\*/

MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, DimSize, Periodic, 1, &GridComm);

//определение координат каждого процессора в решетке

MPI\_Cart\_coords(GridComm, ProcRank, 2, GridCoords);

// создание коммуникатора для строк

Subdims[0] = 0; // фиксация размерности

Subdims[1] = 1; // наличие данной размерности в подсетке

MPI\_Cart\_sub(GridComm, Subdims, &RowComm);

// создание коммуникатора для столбцов

Subdims[0] = 1;

Subdims[1] = 0;

// создание множества коммуникаторов для каждой строки и каждого столбца решетки в отдельности

MPI\_Cart\_sub(GridComm, Subdims, &ColComm);

}

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

double\* pAMatrix; // первая матрица

double\* pBMatrix; // вторая матрица

double\* pCMatrix; // результат

int Size = 2000; // размер матрицы

int BlockSize; // размер матричного блока на процессоре

double \*pAblock; // начальный блок матрицы А

double \*pBblock; // начальный блок матрицы В

double \*pCblock; // результирующий блок

double \*pMatrixAblock;

double Start, Finish, Duration;

/\*Функция setvbuf() позволяет программисту задать буфер, его размер и режим работы с указанным потоком\*/

setvbuf(stdout, 0, \_IONBF, 0);

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &ProcNum);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &ProcRank);

GridSize = sqrt((double)ProcNum);

if (ProcNum != GridSize\*GridSize) {

if (ProcRank == 0) {

printf ("Number of processes must be a perfect square \n");

}

}

else {

if (ProcRank == 0)

printf("Parallel matrix multiplication program\n");

// создание процессорной сетки

CreateGridCommunicators();

// выделение памяти и инициализация матриц

ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock, pCblock, pMatrixAblock, Size, BlockSize );

DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock, Size, BlockSize);

double start\_time = MPI\_Wtime();

ParallelResultCalculation(pAblock, pMatrixAblock, pBblock, pCblock, BlockSize);

double end\_time = MPI\_Wtime();

ResultCollection(pCMatrix, pCblock, Size, BlockSize);

if (ProcRank == 0) {

    /\*

    std::cout << "Matrix A:" << std::endl;

    for (int i = 0; i < Size; ++i) {

        for (int j = 0; j < Size; ++j)

            std::cout << pAMatrix[i \* Size + j] << " ";

        std::cout << std::endl;

    }

    std::cout << "Matrix B:" << std::endl;

    for (int i = 0; i < Size; ++i) {

        for (int j = 0; j < Size; ++j)

            std::cout << pBMatrix[i \* Size + j] << " ";

        std::cout << std::endl;

    }

    std::cout << "Matrix C:" << std::endl;

    for (int i = 0; i < Size; ++i) {

        for (int j = 0; j < Size; ++j)

            std::cout << pCMatrix[i \* Size + j] << " ";

        std::cout << std::endl;

    }

    \*/

    std::cout << "\n\nExecution time: " << end\_time- start\_time << " seconds" << std::endl;

}

MPI\_Finalize();

}

return 0;

}