**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра вычислительной техники**

**ОТЧЕТ**

**по лабораторной работе №1**

**по дисциплине «Архитектура параллельных вычислительных систем»**

**Тема: «Умножение матрицы на вектор и матрицы на матрицу на системах с общей памятью»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 6307 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Киварин Д.М. |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Новиков Б.М. |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Ходос А.А. |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Костичев С.В. |

Санкт-Петербург

2020

**Оглавление**

[Цель 3](#_Toc53409878)

[Задание 3](#_Toc53409879)

[Пример запуска программы 4](#_Toc53409880)

[Экспериментальная проверка эффективности алгоритмов 5](#_Toc53409881)

[Блок-схема параллельного алгоритма 11](#_Toc53409882)

[Заключение 13](#_Toc53409883)

[Текст программы 14](#_Toc53409884)

# Цель

Получить знания о конструировании параллельных алгоритмов матричных умножений на системах с общей памятью; получить общее представление о масштабируемости задач; практическое освоение основных директив OpenMP и mpi; способах распределения вычислений между потоками; способах распределения вычислений итерационных циклов между потоками

# Задание

|  |  |  |
| --- | --- | --- |
| **Вариант** | **Задание** | **Библиотека** |
| 4 | Умножение матрицы на матрицу | MPI |

1. Написать и отладить программу на языке С++, реализующую разработанный алгоритм последовательных и параллельных вычислений с использованием библиотеки MPI.

2. Оценить размерность матрицы, при котором эффективнее использовать алгоритмы последовательного и параллельного вычисления.

# Пример запуска программы

На рисунке 1 изображен скриншот консоли с результатом запусков программы для 2-ух и 4-ех процессов. В результате работы программа выводит в консоль размерность матрицы и время выполнения операции перемножения

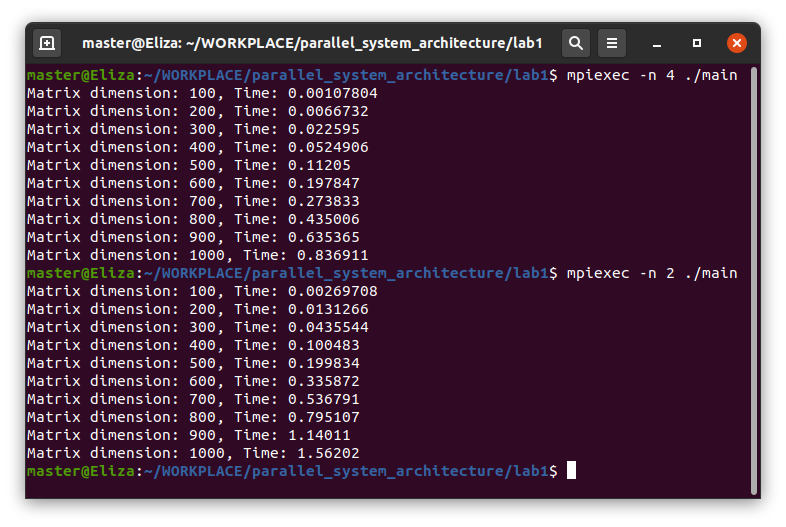


Рисунок 1. Запуск программы для 4-ех и 2-ух процессов

# Экспериментальная проверка эффективности алгоритмов

Аппаратное и программное окружение, в котором производился эксперимент, указано в таблице 1.

|  |  |  |
| --- | --- | --- |
| **ОС** | **IDE** | **Процессор** |
| Ubuntu 20.04 | CLion 2020 Pro | Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz  Ядра: 4  Логических процессоров: 8 |

Таблица 1. Аппаратное и программное окружение

Параметры поставленного эксперимента указаны в таблице 2.

|  |  |  |  |
| --- | --- | --- | --- |
| **Количество процессов** | **Алгоритм** | **Размерность матрицы** | **Количество замеров** |
| 1 | Последовательный | 2х2 – 10х10,  10х10 – 100х100 с шагом 10,  100х100 -1000х1000 с шагом 100 | 5 |
| 2 | Параллельный | 2х2 – 1000х1000,  10х10 – 100х100 с шагом 10,  100х100 -1000х1000 с шагом 100  (КРАТНЫЕ 2) |
| 4 | 4х4 – 1000х1000,  20х20 – 100х100 с шагом 10,  100х100 -1000х1000 с шагом 100  (КРАТНЫЕ 4) |

Таблица 2. Параметры эксперимента

В результате замеров времени исполнения алгоритмов для разного количества процессов было получено по 5 результатов для каждой размерности, было найдено среднее значение для этих результатов и сведено в таблицу 3.

|  |  |  |  |
| --- | --- | --- | --- |
| **Размерность** | **Среднее время (1 процесс)** | **Среднее время (2 процесса)** | **Среднее время (4 процесса)** |
| 2 | 0,000008784 | 0,000015314 | - |
| 3 | 9,856E-07 | - | - |
| 4 | 0,000001462 | 0,000004224 | 3,35416E-05 |
| 5 | 0,000002274 | - | - |
| 6 | 0,000003362 | 0,000003148 | - |
| 7 | 0,000004938 | - | - |
| 8 | 0,00000698 | 0,00000381 | 0,000010558 |
| 9 | 0,000009658 | - | - |
| 10 | 0,00001316 | 0,000008648 | - |
| 20 | 0,0000983 | 0,00004128 | 6,96224E-05 |
| 30 | 0,000340778 | 0,000099191 | - |
| 40 | 0,000671763 | 0,000251538 | 0,00016198 |
| 50 | 0,000670356 | 0,000365118 | - |
| 60 | 0,001140652 | 0,000468039 | 0,000483472 |
| 70 | 0,001823806 | 0,000623188 | - |
| 80 | 0,002844232 | 0,000927636 | 0,000941422 |
| 90 | 0,003900922 | 0,00188993 | - |
| 100 | 0,005361454 | 0,003048736 | 0,001661054 |
| 200 | 0,04283648 | 0,0133242 | 0,01376926 |
| 300 | 0,1463164 | 0,04513692 | 0,03535416 |
| 400 | 0,3367376 | 0,10201 | 0,0676496 |
| 500 | 0,6557924 | 0,1989192 | 0,1123006 |
| 600 | 1,131256 | 0,3431612 | 0,2092738 |
| 700 | 1,78854 | 0,5438752 | 0,3186692 |
| 800 | 2,675556 | 0,8111106 | 0,4675226 |
| 900 | 3,803196 | 1,15819 | 0,6590836 |
| 1000 | 5,227176 | 1,61476 | 0,849266 |

Таблица 3. Среднее время исполнения алгоритмов

На рисунке 2 изображен график зависимости времени исполнения алгоритма от размерности матрицы для размерностей от 2х2 до 10х10.

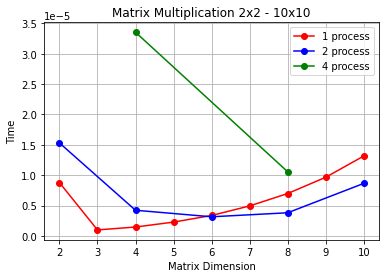


Рисунок 2. График зависимости времени исполнения матричного перемножения от размерности матрицы для размерностей от 2х2 до 10х10

Как видно по графику, для данного диапазона размерностей матрицы оптимальным выбором для использования будет являться вариант с исполнением последовательного алгоритма одним процессом. Ситуация начинает меняться после увеличения размерности больше 6-ти, но не критическим образом. Вариант с двумя процессами показывает близкий результат, но все-таки хуже на матрицах с размерностями меньше 6-ти. 4-ех процессное исполнение параллельного алгоритма для этого диапазона оказывается медленнее других.

На рисунке 3 изображен график зависимости времени исполнения алгоритма от размерности матрицы для размерностей от 2х2 до 50х50.

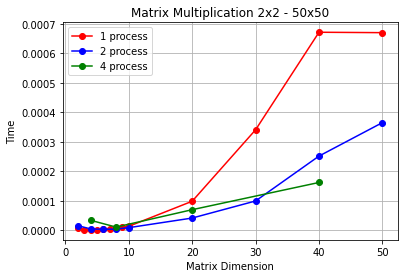


Рисунок 3. График зависимости времени исполнения матричного перемножения от размерности матрицы для размерностей от 2х2 до 50х50

Как видно по графику, для матриц с размерностью от 10 до 30 оптимальным вариантом является использование параллельного алгоритма для 2-ух потоков. От 40-ой размерности первенство по скорости за 4-ех поточным исполнением. Однако, скорость их исполнения не сильно отличается. Однопоточное исполнение последовательного алгоритма с ростом размерности матрицы показывает на данном диапазоне результаты, которые сильно проигрывают другим вариантам.

На рисунке 4 изображен график зависимости времени исполнения алгоритма от размерности матрицы для размерностей от 2х2 до 100х100.

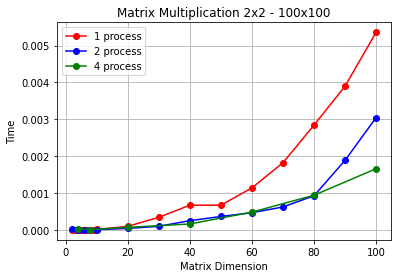


Рисунок 4. График зависимости времени исполнения матричного перемножения от размерности матрицы для размерностей от 2х2 до 100х100

На данном диапазоне более наглядно прослеживается отставание одно-процессного исполнения, а также примерная схожесть результатов для 2-ух и 4-ех процессного исполнения.

На рисунке 5 изображен график зависимости времени исполнения алгоритма от размерности матрицы для размерностей от 2х2 до 1000х1000.

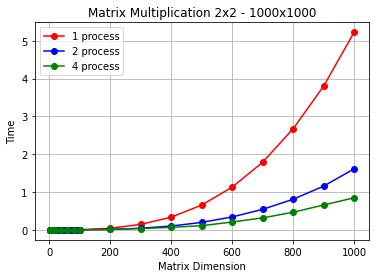


Рисунок 5. График зависимости времени исполнения матричного перемножения от размерности матрицы для размерностей от 2х2 до 1000х1000

На больших размерностях матрицы наглядно видно преимущество исполнения алгоритма 4-мя процессами: оно оказывается быстрее почти в 5 раз одно-процессного исполнения, быстрее 2-ух процессного почти в два раза.

# Блок-схема параллельного алгоритма

На рисунке 7 изображена блок-схема алгоритма параллельного матричного перемножения.

Алгоритм представляет собой итерационную процедуру, количество итераций которой совпадает с числом подзадач.

* На каждой итерации алгоритма каждая подзадача содержит по одинаковому количеству строк матрицы А и матрицы В.
* При выполнении итерации проводится скалярное умножение содержащихся в подзадачах строк, что приводит к получению соответствующих элементов результирующей матрицы С.
* По завершении вычислений в конце каждой итерации столбцы матрицы В должны быть переданы между подзадачами с тем, чтобы в каждой подзадаче оказались новые столбцы матрицы В и могли быть вычислены новые элементы матрицы C.
* По завершении итераций строки собираются в единую матрицу C.

На рисунке 6 представлена демонстрация работы параллельного алгоритма.

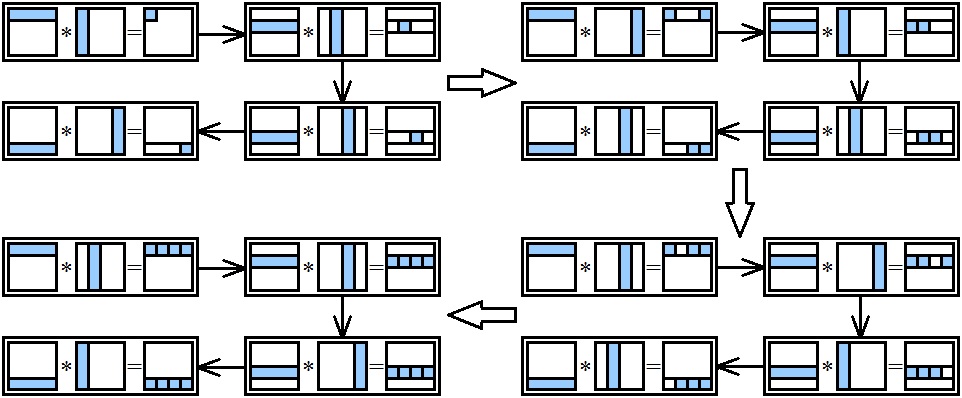


Рисунок 6. Демонстрация алгоритма параллельного матричного перемножения

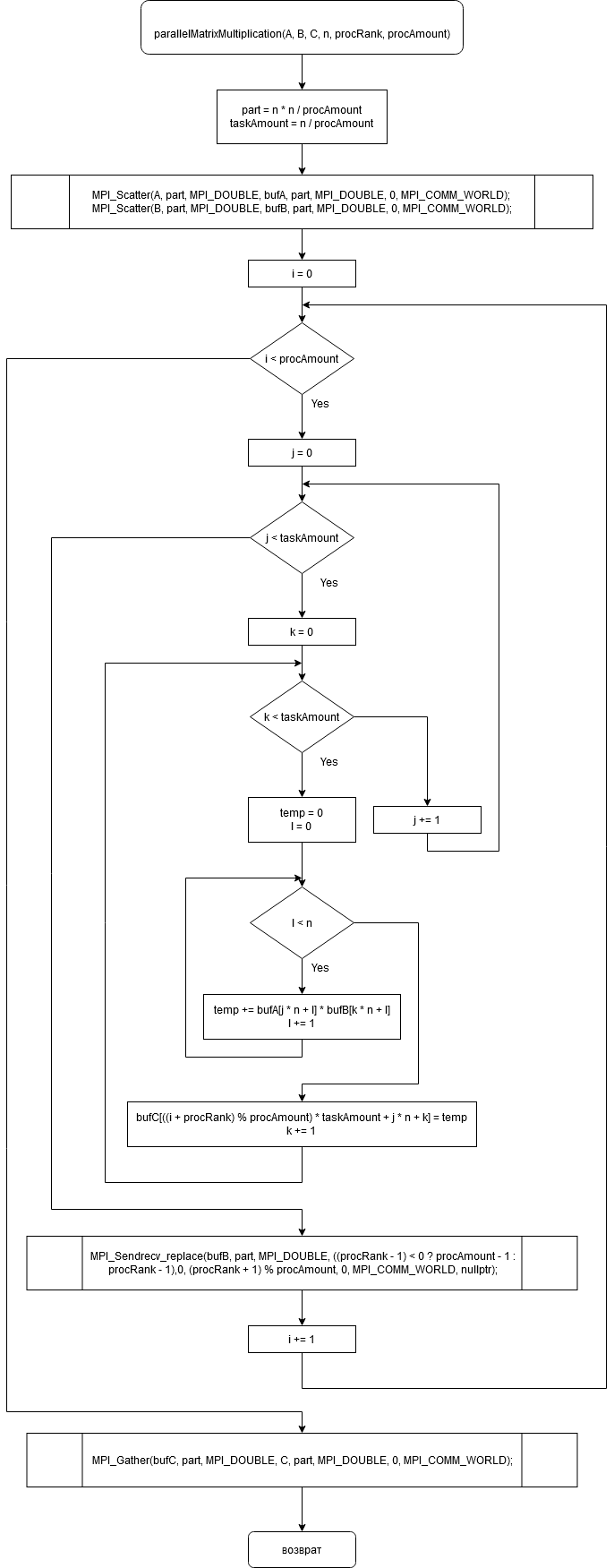


Рисунок 7. Блок-схема параллельного алгоритма матричного перемножения.

# Заключение

Поставленный эксперимент показал следующие результаты:

1. Последовательный алгоритм, исполняемый одним процессом, эффективно работает для размерностей матрицы до 10х10.
2. Параллельный алгоритм, исполняемый двумя процессами, эффективно работает для размерностей матрицы от 10х10 до 80х80.
3. Параллельный алгоритм, исполняемый четырьмя процессами, эффективно работает для размерностей матрицы от 80х80 до 1000х1000.

Из этого следует, что применение параллельных алгоритмов целесообразно для матриц большого размера.

Также, с увеличением размерностей матриц увеличение числа процессов дает хороший результат.

Стоит отметить, что использование многопроцессных вычислений накладывает на данные весьма неудобные ограничения: размерность матрицы должна быть кратна количеству процессов.

# Текст программы

*matrix\_multiplication.h*  - библиотека с реализацией алгоритмов матричного переменожения.

#pragma once  
  
#include <mpi/mpi.h>  
  
void linearMatrixMultiplication(double \*&A, double \*&B, double \*&C, const int &n)  
{  
 for (int i = 0; i < n; ++i)  
 for (int j = 0; j < n; ++j)  
 for (int k = 0; k < n; ++k)  
 C[i \* n + j] += A[i \* n + k] \* B[j \* n + k];  
}  
  
void parallelMatrixMultiplication(double \*&A, double \*&B, double \*&C, const int &n, const int &procRank, const int &procAmount)  
{  
 int part = n \* n / procAmount;  
 int taskAmount = n / procAmount;  
  
 auto \*bufA = new double[part];  
 auto \*bufB = new double[part];  
 auto \*bufC = new double[part];  
  
 MPI\_Scatter(A, part, MPI\_DOUBLE, bufA, part, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  
 MPI\_Scatter(B, part, MPI\_DOUBLE, bufB, part, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  
  
 for (int i = 0; i < procAmount; ++i) {  
 for (int j = 0; j < taskAmount; ++j)  
 for (int k = 0; k < taskAmount; ++k) {  
 double temp = 0;  
 for (int l = 0; l < n; ++l)  
 temp += bufA[j \* n + l] \* bufB[k \* n + l];  
 bufC[((i + procRank) % procAmount) \* taskAmount + j \* n + k] = temp;  
 }  
  
 MPI\_Sendrecv\_replace(bufB, part, MPI\_DOUBLE, ((procRank - 1) < 0 ? procAmount - 1 : procRank - 1),  
 0, (procRank + 1) % procAmount, 0, MPI\_COMM\_WORLD, nullptr);  
 }  
  
 MPI\_Gather(bufC, part, MPI\_DOUBLE, C, part, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);  
  
 delete [] bufA;  
 delete [] bufB;  
 delete [] bufC;  
}

*lab1.cpp* – исходный код программы с экспериментальной оценкой последовательного и матричного перемножения

#include <iostream>  
#include <cstdlib>  
#include "timer.h"  
  
#include "matrix\_multiplication.h"  
  
using namespace std;  
  
// MPI  
void setup(int &argc, char \*\*argv, int &size, int &rank);  
void endup();  
  
// Matrix  
void generate2DMatrix(double \*&M, const int &n);  
void randFill2DMatrix(double \*&M, const int &n);  
void free2DMatrix(double \*&M);  
void show2DMatrix(double \*&M, const int &n);  
  
int main(int argc, char \*\*argv) {  
 //MPI\_INIT  
 int size, rank;  
 setup(argc, argv, size, rank);  
  
 Timer timer;  
 for (int i = 0; i < 10; ++i)  
 {  
 int N = 100 + i \* 100;  
  
 double \*A = nullptr;  
 double \*B = nullptr;  
 double \*C = nullptr;  
  
 generate2DMatrix(A, N);  
 generate2DMatrix(B, N);  
 generate2DMatrix(C, N);  
  
 randFill2DMatrix(A, N);  
 randFill2DMatrix(B, N);  
  
 timer.reset();  
 //linearMatrixMultiplication(A, B, C, N);  
 parallelMatrixMultiplication(A, B, C, N, rank, size);  
 if (rank==0) cout << "Matrix dimension: " << N << ", Time: " << timer.elapsed() << endl;  
  
 free2DMatrix(A);  
 free2DMatrix(B);  
 free2DMatrix(C);  
 }  
  
 /\*if (rank==0) {  
 //show2DMatrix(A, N);  
 //show2DMatrix(B, N);  
 //show2DMatrix(C, N);  
 }\*/  
  
 //MPI\_FINALIZE  
 endup();  
  
 return 0;  
}  
  
void setup(int &argc, char \*\*argv, int &size, int &rank)  
{  
 int res;  
  
 res = MPI\_Init(&argc, &argv);  
 if (res != MPI\_SUCCESS) {  
 fprintf(stderr, "MPI\_Init failed\n");  
 exit(0);  
 }  
  
 res = MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  
 if (res != MPI\_SUCCESS) {  
 fprintf(stderr, "MPI\_Comm\_size failed\n");  
 exit(0);  
 }  
  
 res = MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  
 if (res != MPI\_SUCCESS) {  
 fprintf(stderr, "MPI\_Comm\_rank failed\n");  
 exit(0);  
 }  
}  
  
void endup() {  
 int res = MPI\_Finalize();  
 if (res != MPI\_SUCCESS) {  
 fprintf(stderr, "MPI\_Finalize failed\n");  
 exit(0);  
 }  
}  
  
void generate2DMatrix(double \*&M, const int &n)  
{  
 M = new double[n\*n];  
 for (int i = 0; i < n \* n; ++i) {  
 M[i] = 0;  
 }  
}  
  
void randFill2DMatrix(double \*&M, const int &n)  
{  
 for (int i = 0; i < n \* n; ++i)  
 M[i] = rand() % 10;  
}  
  
void free2DMatrix(double \*&M)  
{  
 delete [] M;  
 M = nullptr;  
}  
  
void show2DMatrix(double \*&M, const int &n)  
{  
 for (int i = 0; i < n; ++i)  
 {  
 for (int j = 0; j < n; ++j)  
 cout << M[i \* n + j] << " ";  
 cout << endl;  
 }  
 cout << endl;  
}