# **ЗАДАНИЕ №1**

Установка и настройка интегрированной среды параллельного программирования C++ и OpenMP

# **Постановка задачи**

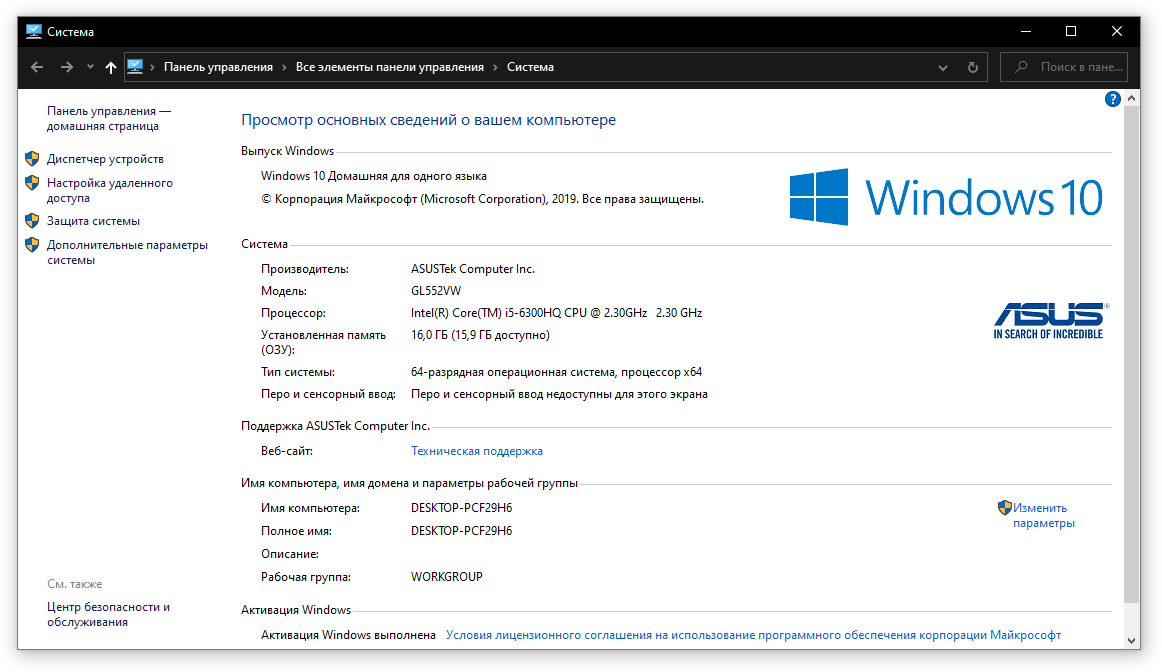
1. Провести обоснованный выбор версии (издания) IDE MVS на основе сопоставления технических характеристик компьютера и версии используемой операционной системы Windows с требованиями, предъявляемой IDE MVS.
2. Скопировать (загрузить) с официального сайта производителя интегрированной среды разработки MVS бесплатно распространяемую версию дистрибутива IDE MVS, например, [2] и произвести ее установку на свой компьютер. Ссылку на официальный сайт в обязательном порядке привести в списке используемых информационных источников.
3. Инсталлировать IDE MVS и настроить интегрированную среду параллельного программирования C++ и OpenMP. В случае, если IDE MVS уже установлена на компьютере, необходимо только расширить ее настройку подключением OpenMP.
4. Создать тестовый проект параллельного приложения для работы с библиотекой OpenMP, который выводит на экран монитора сообщение «Hello world! \n».
5. Модифицировать текст исходного кода теста, установив заданное количество параллельных потоков тремя разными способами с помощью:

* Переменной окружения OMP\_NUM\_THREADS,
* Опции num\_threads() директивы #pragma omp parallel,
* Функции set\_num\_threads()

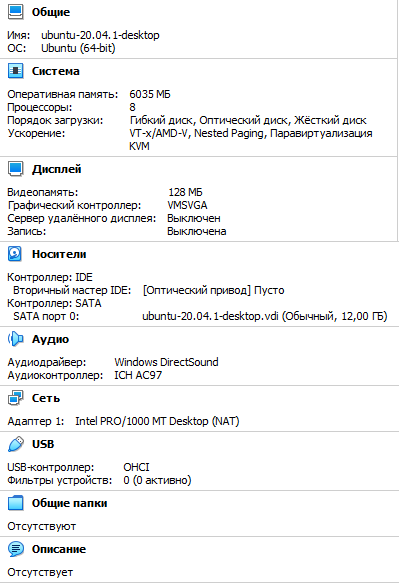
1. Определить приоритет каждого из перечисленных средств.
2. Определить максимальное количество параллельных потоков, которое может быть установлено на компьютере.
3. Сделать выводы по полученным результатам и оформить отчет по выполненной работе.
4. Привести характеристики используемого компьютера и его процессора (в том числе, в обязательном порядке указать производителя, марку, тактовую частоту, количество ядер, наличие технологии hyper treading).

# **Обосновать выбор**

Практическая работа выполняется на виртуальной машине в среде ОС Linux с дистрибутивом Ubuntu 20.04. Технические характеристики компьютера основываются на технических характеристик виртуальной машины, куда установлена ОС (рисунок 1).

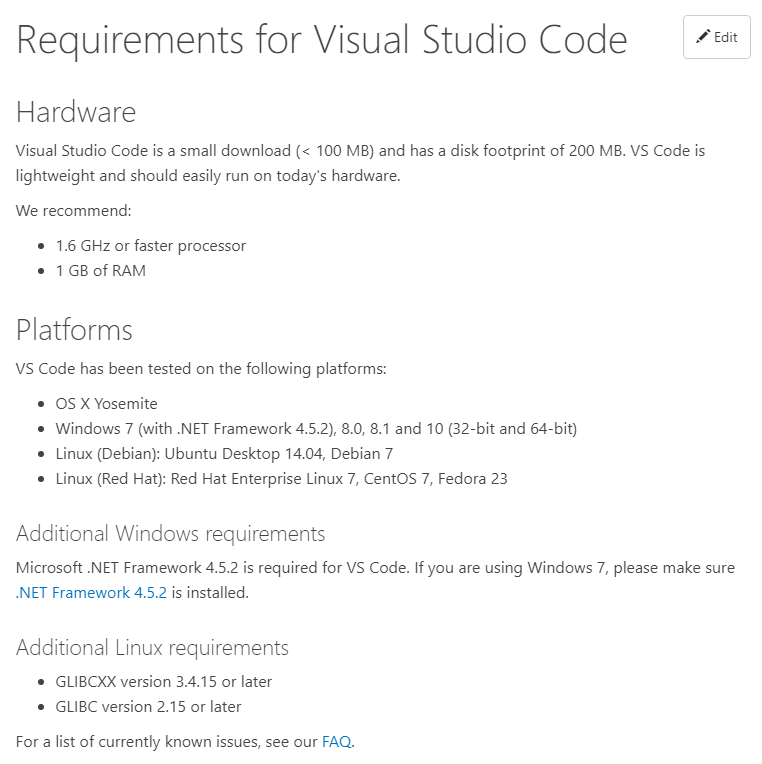


*Рисунок 1. Технические характеристики ПК, на котором установлена ОС.*



*Рис 2. Характеристики ПК (виртуальной машины), используемой при выполнении данной практической работы.*

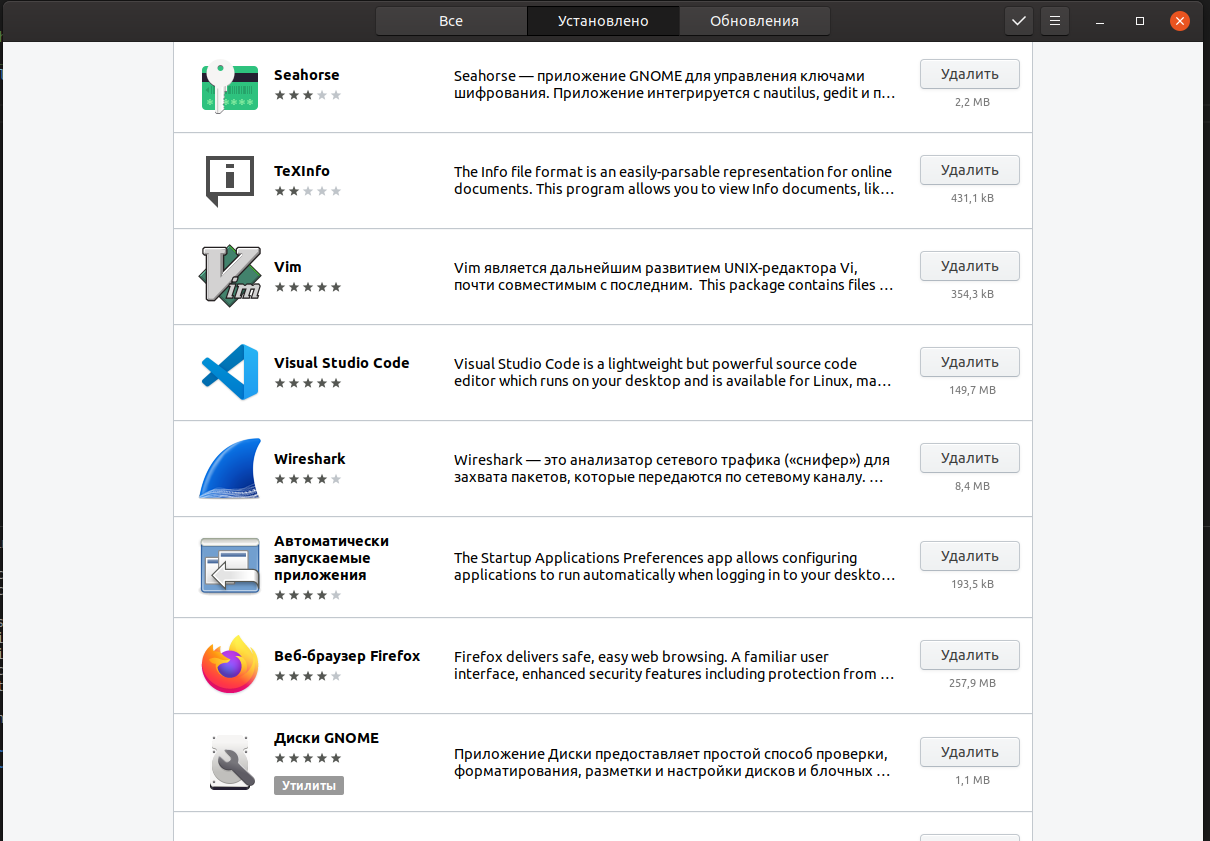
В качестве IDE для разработки будет использоваться Microsoft Visual Studio Code ver 1.50 с поддержкой интерпретатора кода C++ и установленным OpenMP. Системные требования IDE MVS Code представлены на рисунке 2.



*Рисунок 3. Системные требования MVS Code последней версии 1.50.*

# **Установка**

Так как используется ОС Linux с дистрибутивом Ubuntu 20.04, установка MVS Code производилась при помощи внутреннего магазина приложений (рисунок 3).



*Рис 4. Магазин приложений OC Linux Ubuntu 20.04.*

Так же можно установить с помощью команды представленной на официальном сайте VS Code

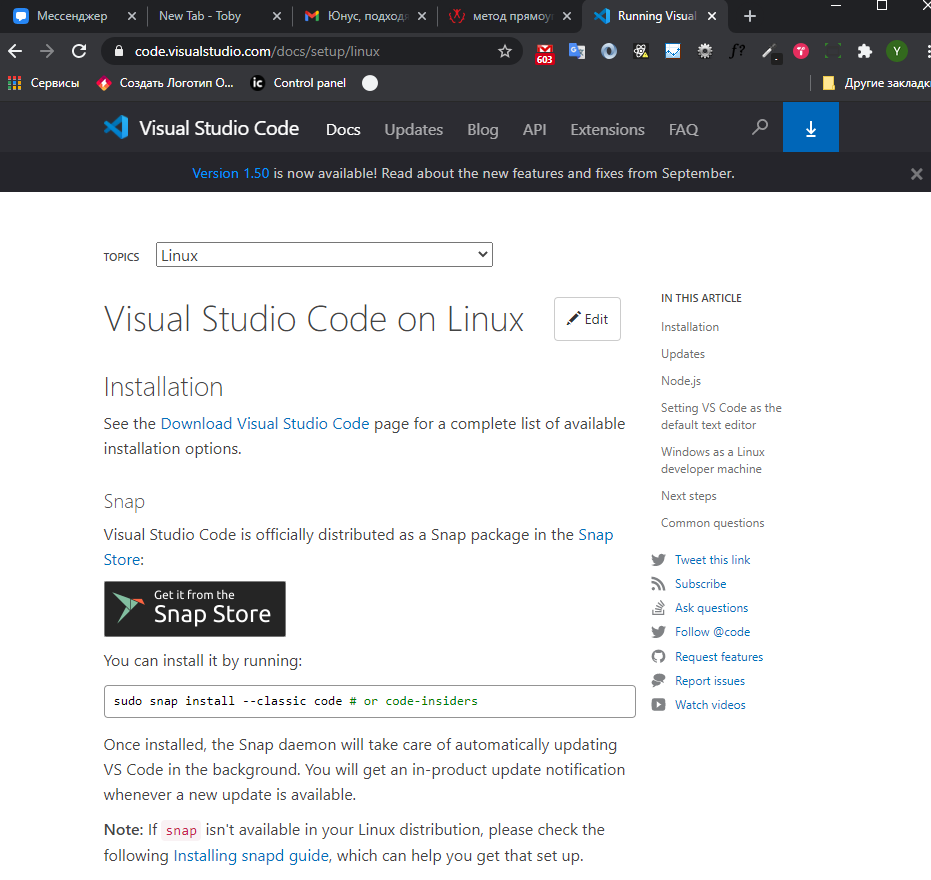
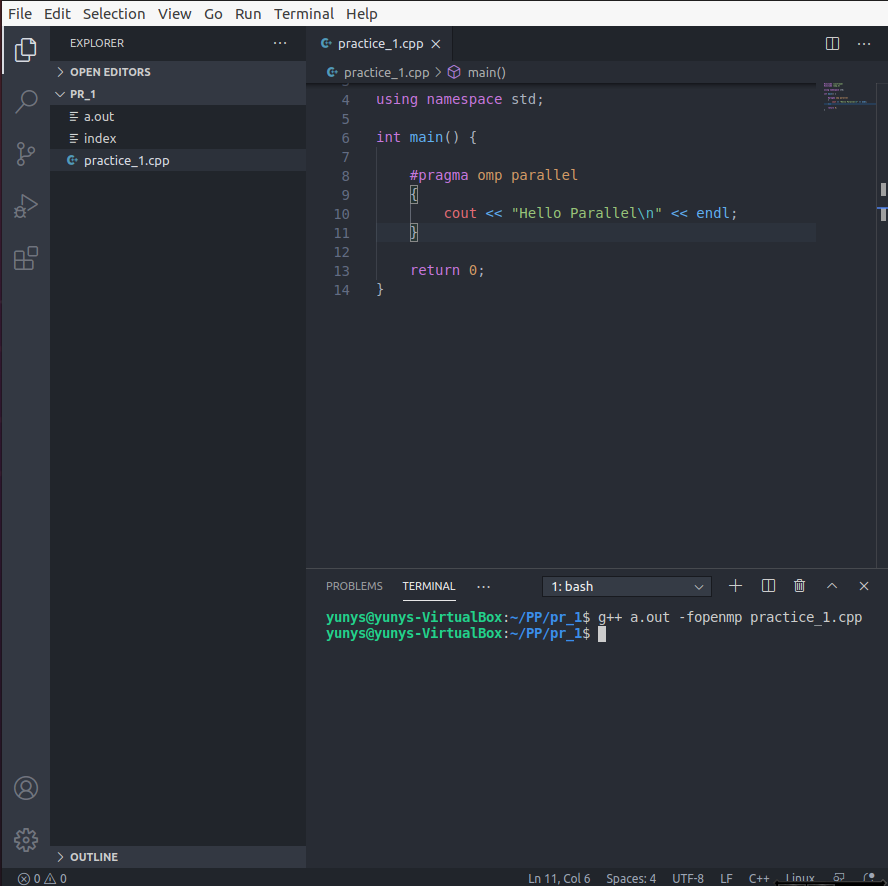


Рис 5 Установка с помощью команды через терминал OC Ubuntu

# **Подключение**

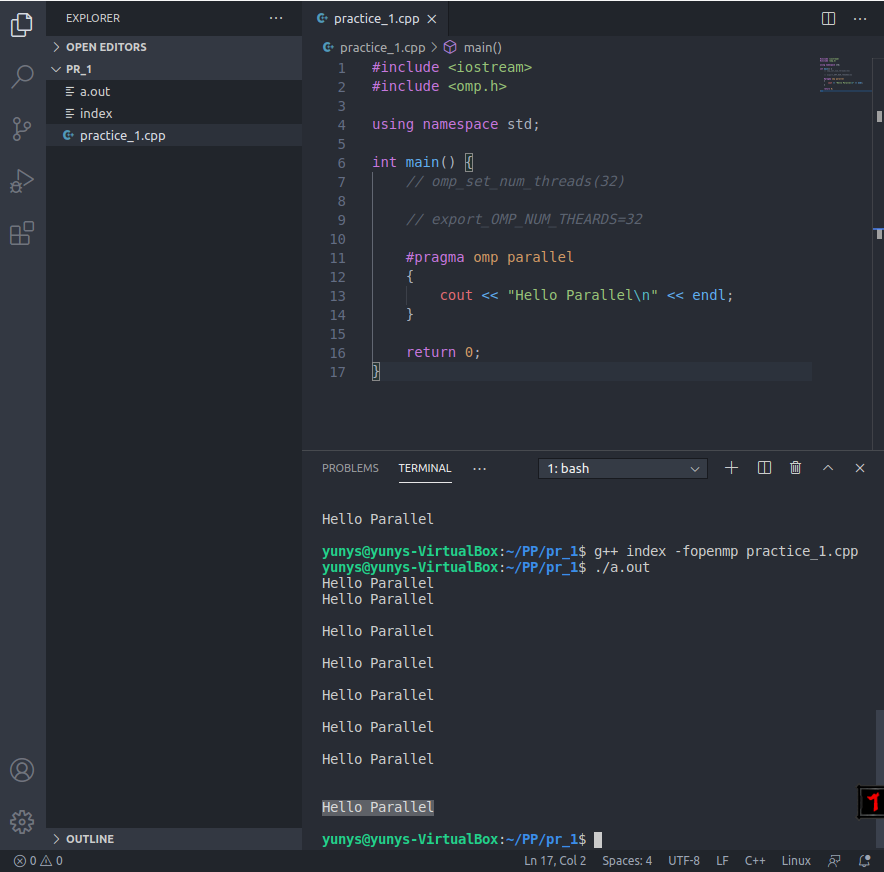
После установки MVS Code ver 1.50 при помощи встроенного терминала была создана директория выполнения для первой практической работы, а также главный файл practice.cpp (рисунок 4).



*Рис 6. Скриншот предварительная настройка MVS Code для выполнения последующих задач практической работы.*

# **Тестирование установки**

В файле practice.cpp необходимо написать код программы, а также скомпилировать исходный файл при помощи команды *g++ -o Имя выходного файла -fopenmp Имя компилируемого файла.* Результат проделанных действий предоставлен на рисунке 5.

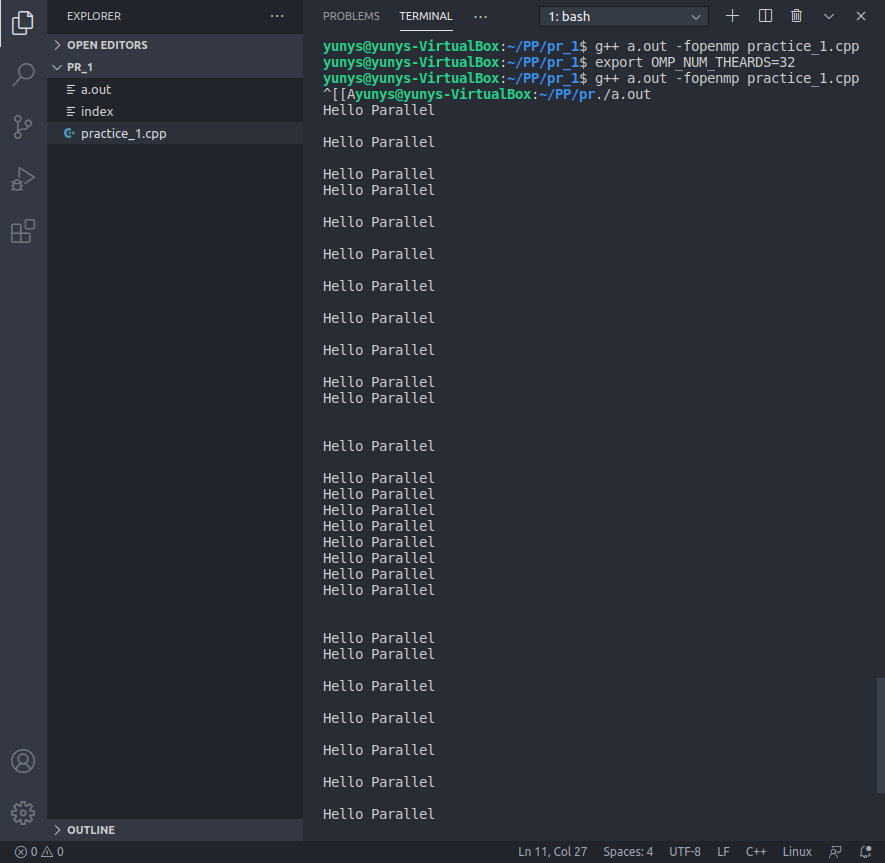


*Рис 7. Скриншот результат работы программы по пункту 4.*

# **Управление количество потоков**

## **Использование переменной окружения**

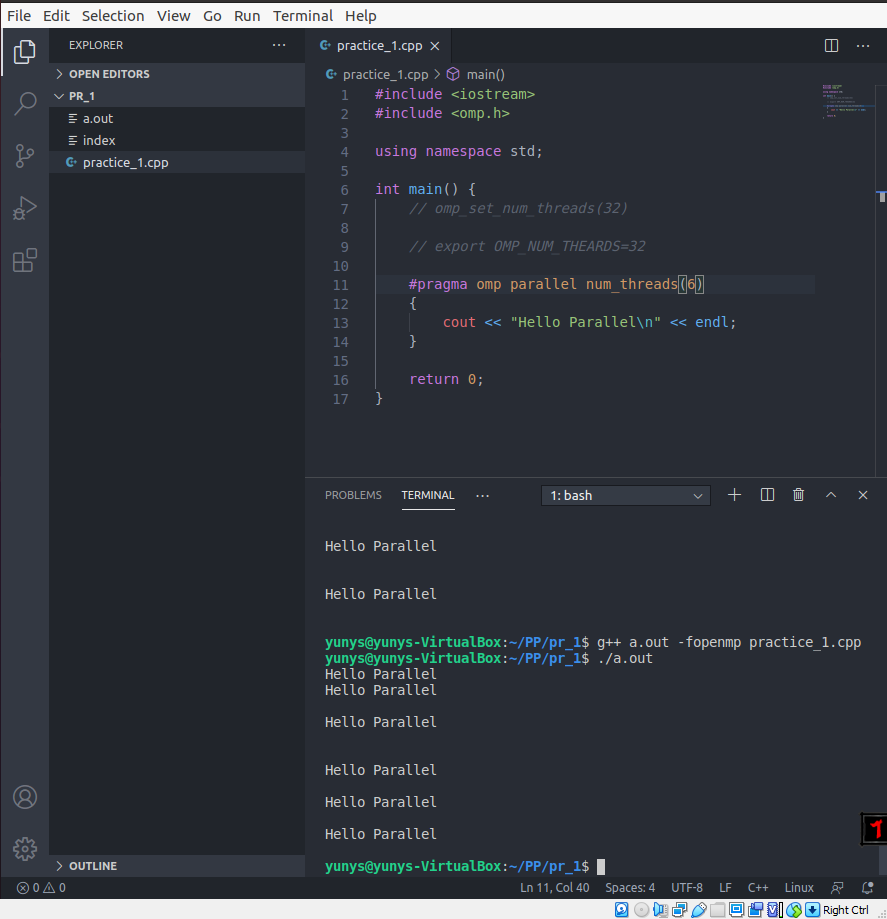
В Unix системах для определения переменной окружения OMP\_NUM\_THREADS достаточно написать export OMP\_NUM\_THREADS = кол-во ядер (в данном случае необходимо написать 32). После выполнения этих действий снопа компилируем программу и смотрим результат (рисунок 6).



*Рис 8. Скриншот результат работы программы по пункту 5.1*

## **Использовании опции**

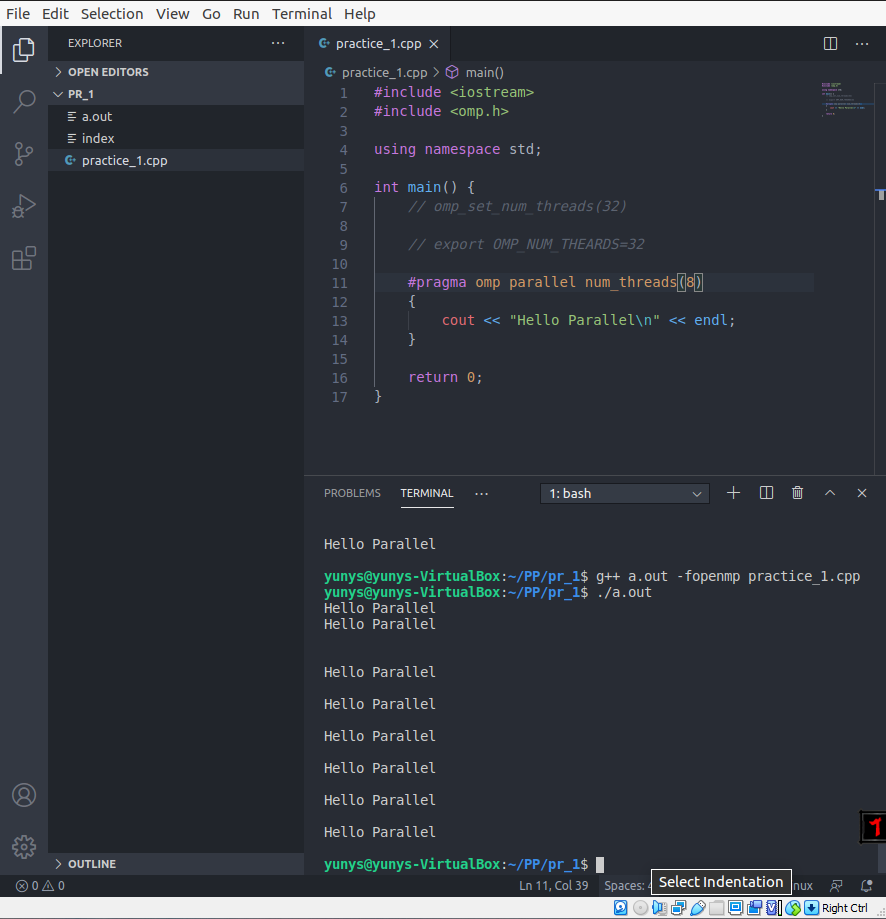
Для выполнения данного пункта задачи, достаточно после #pragma omp parallel написать опция num\_threads (число потоков). В данном примере я выставил 6 потоков. Результат выполнения программы представлен на рисунке 6.



*Рис 9. Скриншот результат работы программы по пункту 5.2*

## **Использование функции**

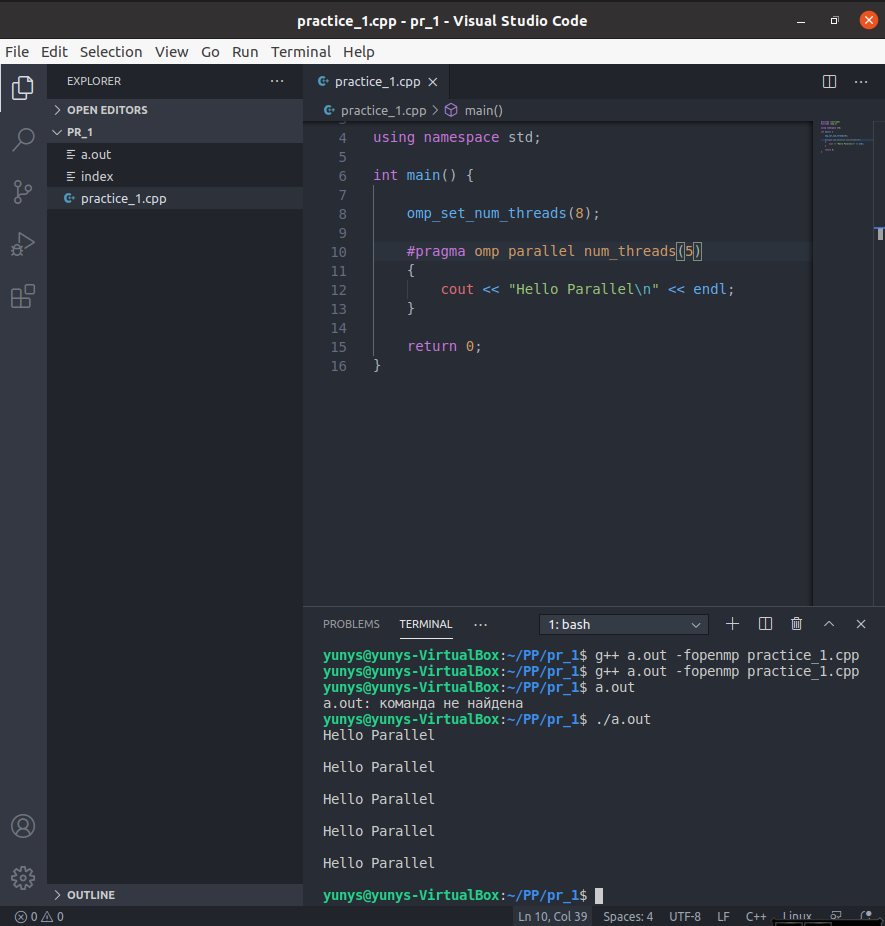
Для выполнения данного пункту задачи 5 достаточно перед строкой #pragma omp parallel написать функцию omp\_set\_num\_threads(количество потоков). В данном примере количество равно 8. Результат работы представлен на рисунке 7.



*Рис 10. Скриншот результат работы программы по пункту 5.3*

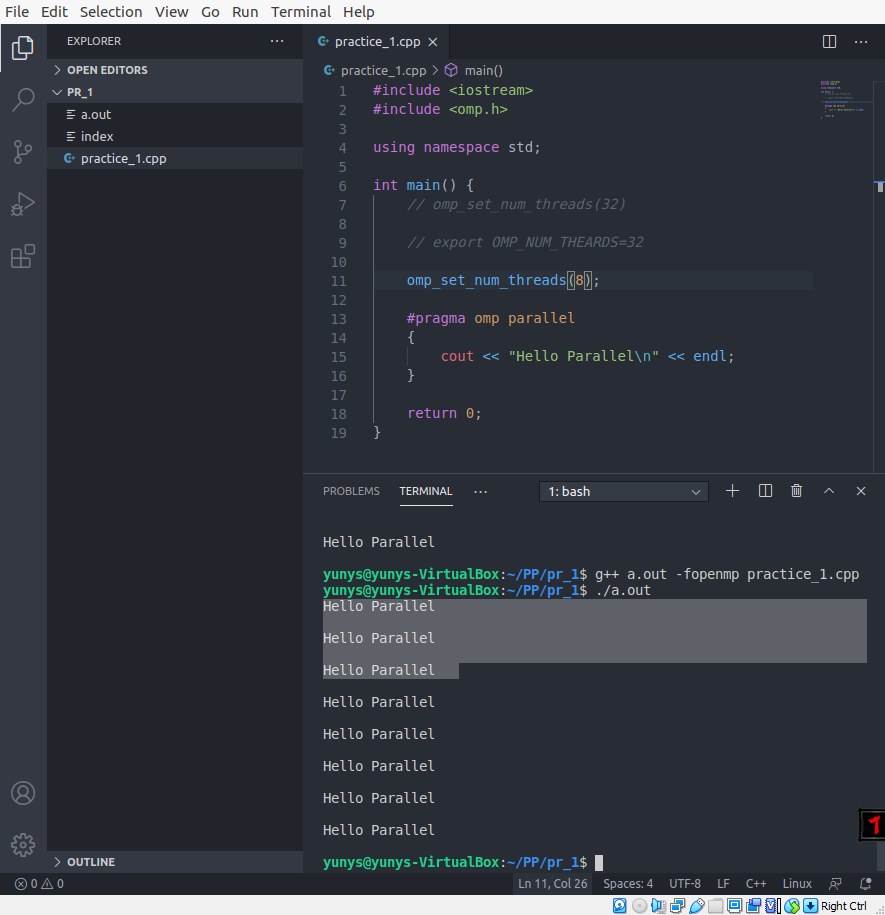
# **Выявить приоритет**

Для проверки приоритета проведем следующие действия: объявим изначально переменную окружения OMP\_NUM\_THREADS = 10, а в коде программы выставим функцию omp\_set\_num\_threads(8) и опцию в директиве num\_threads(5). Результат представлен на рисунке 8.



*Рисунок 11. Скриншот результат работы программы по пункту 6(1).*

Исходя из представленного рисунка, можем сделать вывод, что по приоритету выполнения вначале будет стоять опция директивы. Далее, уберем эту опцию и повторим процедуру для окончательного формирования вывода по приоритету каждого из перечисленных средств изменения номеров потоков. Результат представлен на рисунке 9.

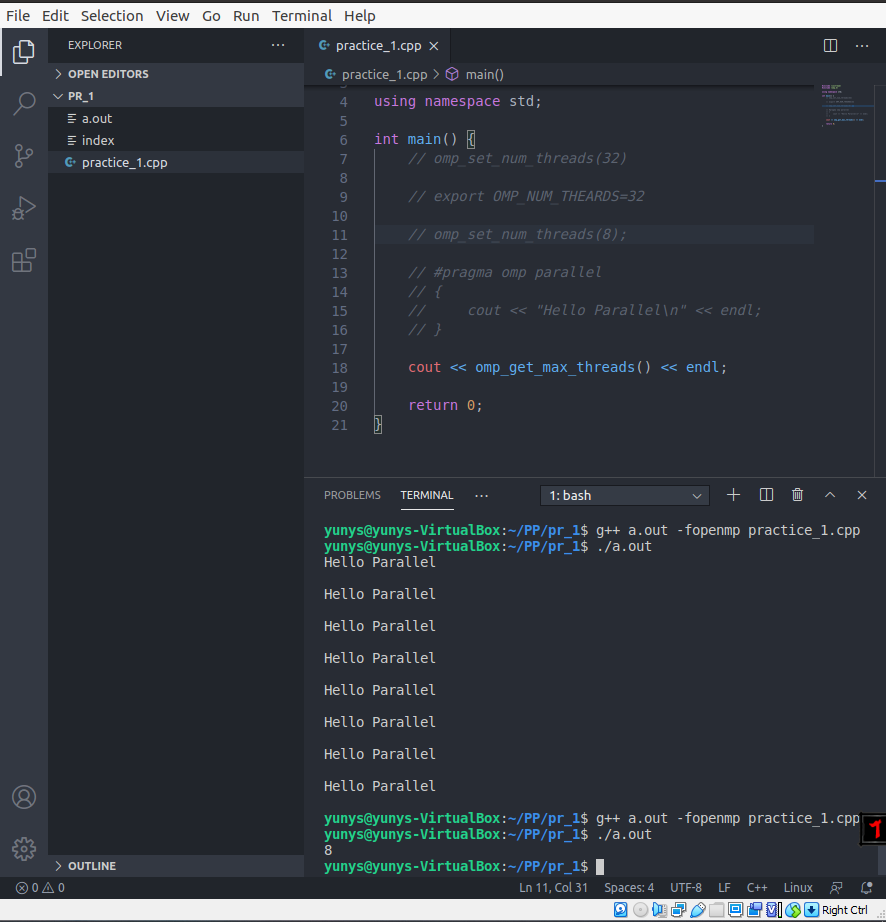


*Рисунок 12. Скриншот результат работы программы по пункту 6(2).*

Исходя из этого результата, делаем вывод что приоритет способа потоков следующий: 1 – опция директивы, 2 – функция omp\_set\_num\_threads(), 3 – переменная окружения OMP\_NUM\_THREADS

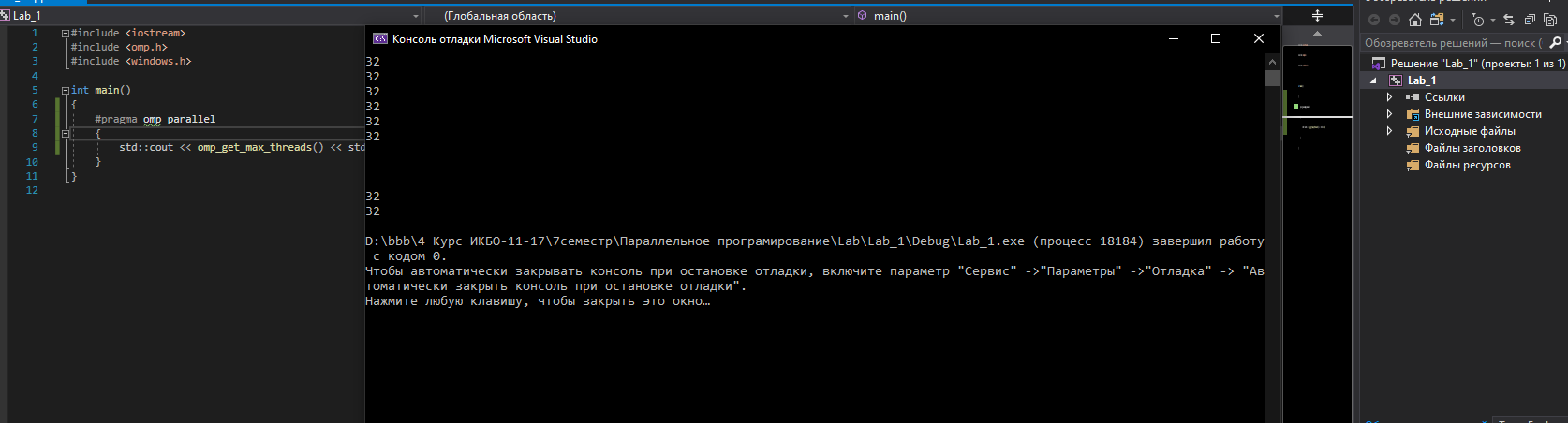
# **Вывести количество потоков**

Для выполнения данного пункта задачи необходимо вывести в коде значение функции omp\_get\_max\_threads(). Исходя из результатов рисунка 10 убеждаемся, что количество потоков равно количеству ядер используемого процессора. В моем случае – это число 8. Число 8 потому что виртуальная машина настроена на 8 потоков.



*Рисунок 13. Скриншот результат работы программы задачи 7.*

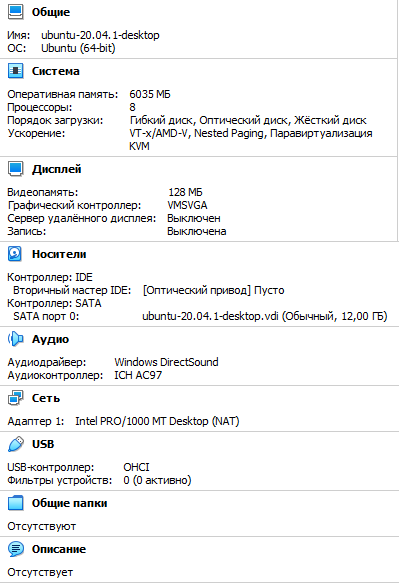
На физической машине максимальное количество потоков = 32, это можно определить по рисунку 14.



*Рисунок 14. Скриншот результат работы программы задачи 8.*

# **Ход выполнения задачи 9**

Характеристики ПК, используемого в практической работе представлены на рисунке 11. Полное наименование процессора, на котором запущена виртуальная машина – Intel Core i5-6300HQ 2.30MHz. Технология hyper threading отсутствует.



*Рис 13. Характеристики ПК (виртуальной машины), используемой при выполнении данной практической работы.*

# **Выводы**

В ходе выполнения данной практической работы была установлена и настроена IDE MVS Code ver 1.50. Так же рассмотрена работа с пакетом для языка C++ OpenMp, выполнены задачи настоящей практической работы, по каждой задаче проведены соответствующие выводы.Параллельное программирование с использованием основ технологии OpenMP.

# **ЗАДАНИЕ №2**

Параллельное программирование с использованием основ технологии OpenMP

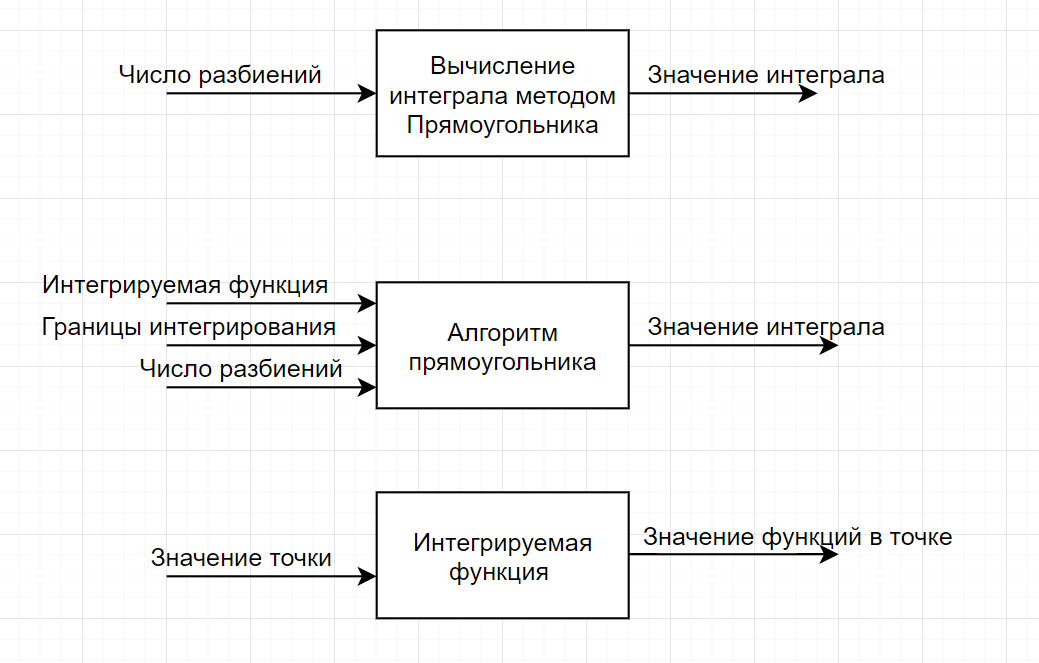
# **Постановка задачи**

Составить программу последовательного и параллельного вычисления определённого интеграла методом прямоугольников. Провести контрольные прогоны программы для числа разбиений отрезка интегрирования n = 104, 105, 106, 107, 108 и установленных для параллельного варианта количествах потоков p = 4, 8, 16, 32 и 64 с вычислением времени выполнения и ускорения. Полученные результаты свести в сводную таблицу.

Построить графики изменения ускорения при последовательном и параллельных вычислениях в зависимости от числа разбиений отрезка интегрирования. Построить графики изменения ускорения при параллельных вычислениях в зависимости от количества используемых потоков. Вычислить показатели эффективности и стоимости параллельной реализации программы. Провести анализ полученных результатов. Сделать выводы о проделанной работе, основанные на полученных данных.

# **Описание алгоритмов, используемых для решения задачи**

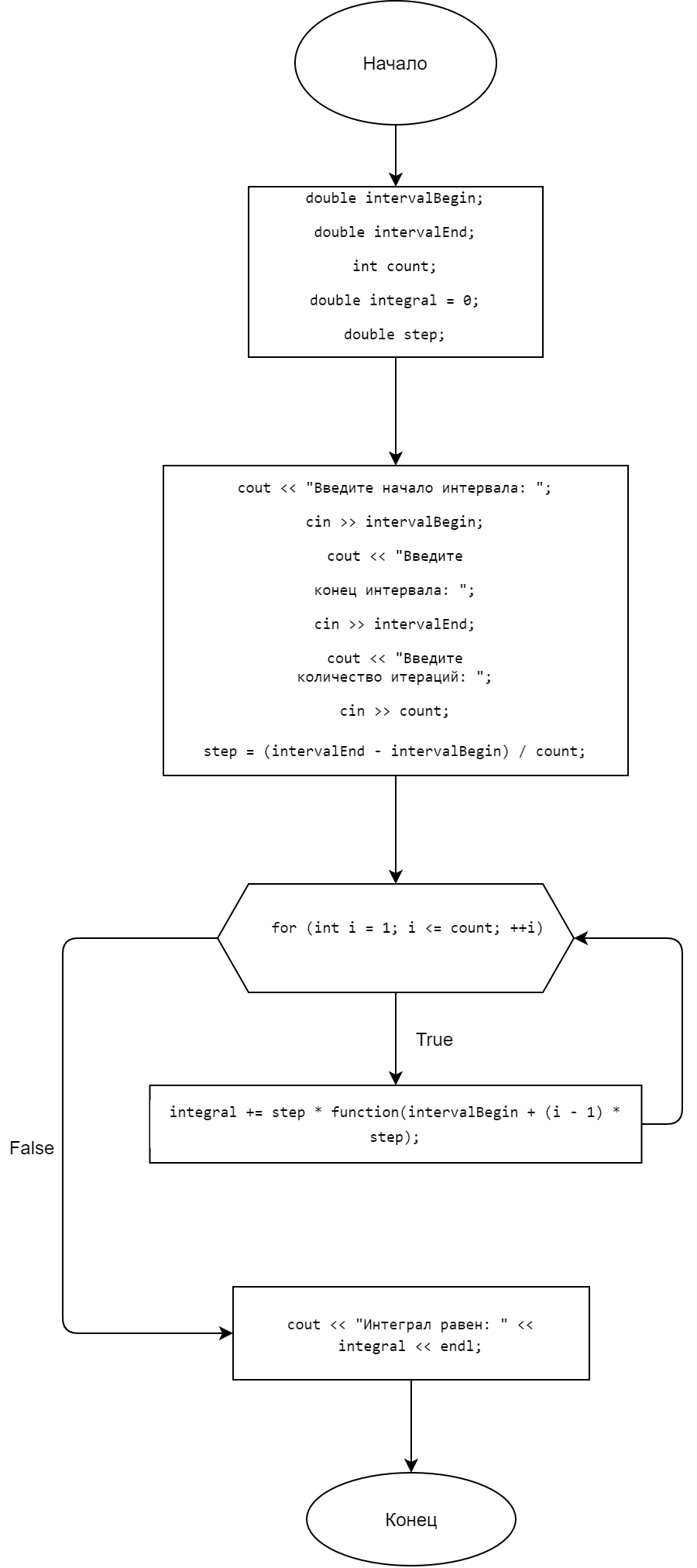
Для выполнения данной практической работы было решено провести декомпозицию общего процесса на более мелкие составляющие (рисунок 1).



*Рисунок 1. Декомпозиция общей задачи на подзадачи*

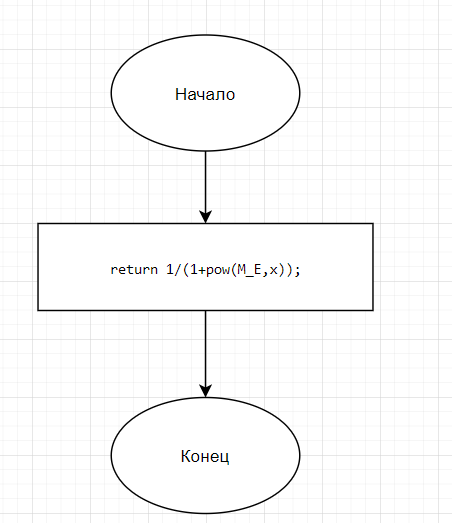
После проведения декомпозиции опишем использование всех алгоритмов, используемых в решении общей задачи.

Во-первых, ниже продемонстрирован рисунок общего алгоритма работы программы (рисунок 2).



*Рисунок 2. Схема алгоритма работы программы*

Во-вторых, продемонстрирован рисунок алгоритма вычисления значения функции в точке (рисунок 4).



*Рисунок 4. Схема алгоритма вывода элементов массива на экран*

# **Текст исходного кода программы**

|  |
| --- |
| #include <iostream>  #include <math.h>  #include <omp.h>  using namespace std;  double function(double x);  int main() {  double intervalBegin;  double intervalEnd;  int count;  double integral = 0;  double step;  cout << "Введите начало интервала: ";  cin >> intervalBegin;  cout << "Введите конец интервала: ";  cin >> intervalEnd;  cout << "Введите количество итераций: ";  cin >> count;  step = (intervalEnd - intervalBegin) / count;  #pragma omp parallel for  for (int i = 1; i <= count; ++i) {  integral += step \* function(intervalBegin + (i - 1) \* step);  }  cout << "Интеграл равен: " << integral << endl;  int a;    cin >> a;  return 0;  }  double function(double x) {  return 1/(1+pow(M\_E,x));  } |
|  |

Листинг файла practice.cpp. Исходный код программы.

# **Тестирование**

На основе выполненных тестов были составлены сводные таблицы, на основе которых составлены графики, указанные в основной задаче.

*Таблица 1. Результаты работы сортировки без использования методов распараллеливания*

|  |  |  |
| --- | --- | --- |
| № тестирования | Количество разбиений отрезков, n | Время выполнения работы программы, сек. |
| 1 | 10000 | 0.005 |
| 2 | 100000 | 0.009 |
| 3 | 1000000 | 0.020 |
| 4 | 10000000 | 0.097 |
| 5 | 100000000 | 3.462 |

*Таблица 2. Результаты работы сортировки с использованием методов распараллеливания при количестве потоков p = 4*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № тестирования | Количество элементов массива, n | Время выполнения программы, сек | Ускорение (Sn) | Эффективность | Стоимость |
| 1 | 10000 | 0.033 | 0.09 | 0.008 | 0.132 |
| 2 | 100000 | 0.014 | 0.71 | 0.004 | 0.056 |
| 3 | 1000000 | 0.022 | 3.81 | 0.006 | 0.088 |
| 4 | 10000000 | 0.245 | 7.61 | 0.061 | 0.98 |
| 5 | 100000000 | 2.317 | 3.59 | 0.579 | 9.268 |

*Таблица 3. Результаты работы сортировки с использованием методов распараллеливания при количестве потоков p = 8*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № тестирования | Количество элементов массива, n | Время выполнения программы, сек | Ускорение (Sn) | Эффективность | Стоимость |
| 1 | 10000 | 0.020 | 0.15 | 0.0025 | 0.16 |
| 2 | 100000 | 0.053 | 0.19 | 0.006625 | 0.424 |
| 3 | 1000000 | 0.06 | 1.4 | 0.0075 | 0.48 |
| 4 | 10000000 | 0.240 | 7.78 | 0.03 | 1.92 |
| 5 | 100000000 | 2.068 | 4.03 | 0.2585 | 16.544 |

*Таблица 4. Результаты работы сортировки с использованием методов распараллеливания при количестве потоков p = 16*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № тестирования | Количество элементов массива, n | Время выполнения программы, сек | Ускорение (Sn) | Эффективность | Стоимость |
| 1 | 10000 | 0.002 | 1.5 | 0.000125 | 0.032 |
| 2 | 100000 | 0.056 | 0.18 | 0.0035 | 0.896 |
| 3 | 1000000 | 0.052 | 1.62 | 0.00325 | 0.832 |
| 4 | 10000000 | 0.176 | 10.59 | 0.011 | 2.816 |
| 5 | 100000000 | 2.097 | 3.98 | 0.1310625 | 33.552 |

*Таблица 5. Результаты работы сортировки с использованием методов распараллеливания при количестве потоков p = 32*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № тестирования | Количество элементов массива, n | Время выполнения программы, сек | Ускорение (Sn) | Эффективность | Стоимость |
| 1 | 10000 | 0.025 | 0.12 | 0.00078125 | 0.8 |
| 2 | 100000 | 0.004 | 2.5 | 0.000125 | 0.128 |
| 3 | 1000000 | 0.021 | 4.0 | 0.00065625 | 0.672 |
| 4 | 10000000 | 0.274 | 6.81 | 0.0085625 | 8.768 |
| 5 | 100000000 | 2.127 | 3.92 | 0.06646875 | 68.064 |

*Таблица 6. Результаты работы сортировки с использованием методов распараллеливания при количестве потоков p = 64*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № тестирования | Количество элементов массива, n | Время выполнения программы, сек | Ускорение (Sn) | Эффективность | Стоимость |
| 1 | 10000 | 0,045 | 0.067 | 0.0007 | 2.88 |
| 2 | 100000 | 0,029 | 0.345 | 0.0004 | 1.856 |
| 3 | 1000000 | 0,054 | 1.56 | 0.0008 | 3.456 |
| 4 | 10000000 | 0,167 | 11.16 | 0.0026 | 10.688 |
| 5 | 100000000 | 1,719 | 4.85 | 0.0269 | 110.016 |

*Рисунок 7. Зависимости ускорения при различных потоках от количества элементов массива*

На основе данного рисунка можно сделать вывод о том, что ускорение происходит на больших величинах (в районе 10000000 разбиений) больше всего при выполнении алгоритма на 64 и 16 потоков. Самое же низкое ускорение достигается при использовании 32 потоков.

*Рисунок 8. Графики зависимости времени выполнения при различных потоках от количества элементов массива*

На основе данного рисунка можно сделать вывод о том, что самое долгое затраченное время на вычисление интеграла при разбиении отрезка от 10000 до 1000000000 выполнение происходит у не распараллеленного алгоритма. При использовании потоков время выполнения намного меньше и все примерно равны, где наименьшее время выполнения у 64 потоков.

*Рисунок 9. Графики зависимости эффективности выполнения при различных потоках от количества элементов массива*

На основе данного рисунка можно сделать вывод о том, самая высокая эффективность достигается при 4 поточном вычислений значений. Усредненная эффективность лучше всего достигается при выполнении кода при 16 потоках. Худший вариант – использование 64 потоков.

*Рисунок 10. Графики зависимости стоимости выполнения при различных потоках от количества элементов массива*

На основе данного рисунка можно сделать вывод о том, самая высокая стоимость выполнения достигается при использовании 64 потоков, самая маленькая – при использовании 4 потоков.

# **Выводы**

В соответствии с полученными данными построены различные графики зависимости, к которым сделаны соответствующие логические выводы.

1. Наилучшая стоимость при 64 потоках
2. Наилучшая эффективность при 64 потоках
3. Наилучшее время при 4 потоках
4. Наилучшее ускорение при 64 потоках
5. Наихудшая стоимость при 4 потоках
6. Наихудшее эффективность при 64 потоках
7. Наихудшее время при 64 потоках
8. Наихудшее ускорение при 4 потоках

# **ЗАДАНИЕ №2**

Параллельное программирование с использованием расширенных средств технологии OpenMP (редукции, атомарных операций, критических секций, замков).

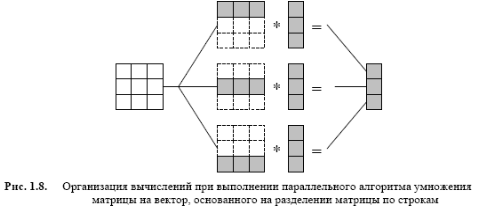
# **Постановка задачи**

Составить программу последовательного и параллельного умножения двух матриц A[n,n] и В[n,n], используя горизонтальный (rowwize) алгоритм разбивки матрицы А по столбцам. Провести тестирование программы (в последовательном и параллельном исполнениях) на массивах размером n = 3, сформированных вводом с клавиатуры. Рабочие матрицы А и В, сформировать используя генератор псевдослучайных чисел. Провести контрольные прогоны программы для размеров n = 500, 600, 700, 800, 900, 1000 в последовательном и параллельном исполнении с количеством потоков p = 2, 4, 8, 16, 32. Полученные результаты свести в сводную таблицу. Построить графики изменения ускорения умножения в зависимости от размеров матриц. Построить графики изменения ускорения параллельных вычислений с разным количеством потоков. Вычислить показатели эффективности и стоимости параллельной реализации программы. Провести анализ полученных результатов. Сделать выводы о проделанной работе, основанные на эмпирических данных.

# **Описание алгоритмов, используемых для решения задачи**

Последовательный алгоритм представляется тремя вложенными циклами и ориентирован на последовательное вычисление столбцов результирующей матрицы С, здесь можно выполнить циклы параллельно, так как результаты не зависят друг от друга.

Пусть A и B – матрицы, которые необходимо перемножить, C – матрица, содержащая результат умножения. Начальные значения всех элементов матрицы C равны нулю. Алгоритм представляет собой итерационную процедуру. На каждой итерации алгоритма один столбец матрицы A умножается на один столбец матрицы B. При выполнении итерации проводится поэлементное умножение столбцов, что приводит к получению столбца частичных результатов для матрицы С. Пример работы алгоритма представлен на рисунке 1.



*Рисунок 1 Организация вычислений при выполнении параллельного алгоритма на примере умножения матрицы на вектор*

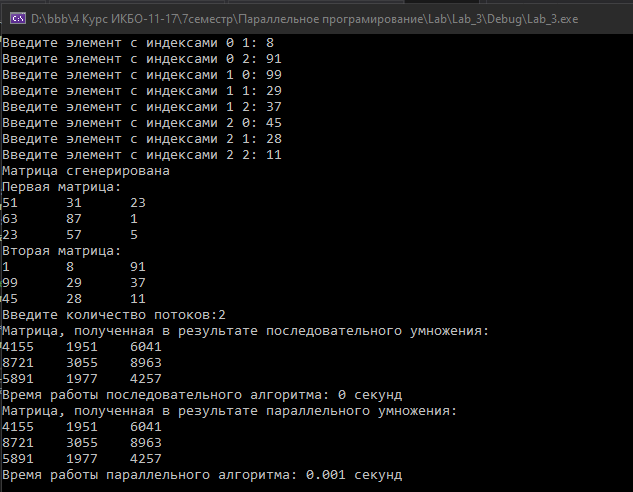
# **Текст исходного кода программы**

|  |
| --- |
| // Lab\_3.cpp : Этот файл содержит функцию "main". Здесь начинается и заканчивается выполнение программы.  //  #include <iostream>  #include <omp.h>  #include <Windows.h>  #include <time.h>  #include <ctime>  // Генерация рандомных чисел  int get\_random\_number(int min, int max)  {  static const double fraction = 1.0 / (static\_cast<double>(RAND\_MAX) + 1.0);  // Равномерно распределяем рандомное число в нашем диапазоне  return static\_cast<int>(rand() \* fraction \* (max - min + 1) + min);  }  // Функция создания двумерной матрицы  int\*\* array\_generator(unsigned int dim)  {  int\*\* ptrary = new int\* [dim];  for (int i = 0; i < dim; i++) {  ptrary[i] = new int[dim];  }  return ptrary;  }  // Функция удаления двумерной матрицы  void array\_destroyer(int\*\* ary, unsigned int dim1)  {  for (int i = 0; i < dim1; i++) {  delete[] ary[i];  }  delete[] ary;  }  // Функция заполнения матрицы нулями  int\*\* matrix\_generate\_empty(int\*\* matrix, int n) {  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  matrix[i][j] = 0;  }  }  return matrix;  }  // Заполнение матрицы вручную  int\*\* fill\_matrix(int\*\* matrix, int n)  {  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  std::cout << "Введите элемент с индексами " << i << " " << j << ": ";  std::cin >> matrix[i][j];  }  }  std::cout << "Матрица сгенерирована\n";  return matrix;  }  // Функция вывода матрицы на экран  int\*\* matrix\_output(int\*\* matrix, int n)  {  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  std::cout << matrix[i][j] << "\t";  }  std::cout << std::endl;  }  return matrix;  }  // Функция заполнения матрицы с помощью генератора случайных чисел  int\*\* matrix\_generate(int\*\* matrix, int n) {  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  matrix[i][j] = get\_random\_number(0, 200);  }  }  std::cout << "Матрица сгенерирована\n";  return matrix;  }  // Процедура последовательного умножения двух матриц  void sequential\_matrix\_multiplication(int\*\* matrix\_one, int\*\* matrix\_two, int n)  {  int\*\* matrix\_result = array\_generator(n);  matrix\_generate\_empty(matrix\_result, n);  int start = clock();  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  for (int k = 0; k < n; k++) {  matrix\_result[i][j] += matrix\_one[i][k] \* matrix\_two[k][j];  }  }  }  int stop = clock();  //std::cout << "Матрица, полученная в результате последовательного умножения: \n";  //matrix\_output(matrix\_result, n);  std::cout << "Время работы последовательного алгоритма: " << ((double)stop - (double)start) / CLOCKS\_PER\_SEC << " секунд" << std::endl;  array\_destroyer(matrix\_result, n);  }  // Процедура параллельного умножения двух матриц  void parallel\_matrix\_multiplication(int\*\* matrix\_one, int\*\* matrix\_two, int n, int thread)  {  int\*\* matrix\_result = array\_generator(n);  matrix\_generate\_empty(matrix\_result, n);  int start = clock();  omp\_set\_num\_threads(thread);  #pragma omp parallel for  for (int i = 0; i < n; i++) {  for (int j = 0; j < n; j++) {  for (int k = 0; k < n; k++) {  matrix\_result[i][j] += matrix\_one[i][k] \* matrix\_two[k][j];  }  }  }  int stop = clock();  //std::cout << "Матрица, полученная в результате параллельного умножения: \n";  //matrix\_output(matrix\_result, n);  std::cout << "Время работы параллельного алгоритма: " << ((double)stop - (double)start) / CLOCKS\_PER\_SEC << " секунд" << std::endl;  array\_destroyer(matrix\_result, n);  }  void menu()  {  std::cout << "Меню программы:\n";  std::cout << "1 - Ручное заполнение\n";  std::cout << "2 - Заполнение случайными числами\n";  std::cout << "0 - Завершение программы\n";  }  int main()  {  // Настройка кодировки в консоли  SetConsoleCP(1251);  SetConsoleOutputCP(1251);  int command; // Выбор раздела меню  int number\_matrix; // Выбор размера матрицы  int threads; // Выбор количества поток  int\*\* matrix\_one = 0;  int\*\* matrix\_two = 0;  srand(time(NULL));  menu();  do  {  std::cout << "\nВведите команду: ";  std::cin >> command;  switch (command)  {  case 1:    do  {  std::cout << "Введите размерность матрицы: ";  std::cin >> number\_matrix;    if (number\_matrix > 0)  {  break;  }  else  {  std::cout << "Проверьте входные данные! Размерность должна быть положительна!\n";  }  } while (true);    matrix\_one = array\_generator(number\_matrix);  matrix\_two = array\_generator(number\_matrix);  std::cout << "Заполнение первой матрицы" << std::endl;  fill\_matrix(matrix\_one, number\_matrix);  std::cout << "Заполнение второй матрицы" << std::endl;  fill\_matrix(matrix\_two, number\_matrix);  std::cout << "Первая матрица:\n";  matrix\_output(matrix\_one, number\_matrix);  std::cout << "Вторая матрица:\n";  matrix\_output(matrix\_two, number\_matrix);  do  {  std::cout << "Введите количество потоков:";  std::cin >> threads;  if (threads > 0)  {  break;  }  else  {  std::cout << "Проверьте входные данные! Размерность должна быть положительна!\n";  }  } while (true);  // Последовательное умножение  sequential\_matrix\_multiplication(matrix\_one, matrix\_two, number\_matrix);  // Распараллеливание умножения матриц  parallel\_matrix\_multiplication(matrix\_one, matrix\_two, number\_matrix, threads);  array\_destroyer(matrix\_one, number\_matrix);  array\_destroyer(matrix\_two, number\_matrix);  break;  case 2:  do  {  std::cout << "Введите размерность матрицы: ";  std::cin >> number\_matrix;  if (number\_matrix > 0)  {  break;  }  else  {  std::cout << "Проверьте входные данные! Размерность должна быть положительна!\n";  }  } while (true);  matrix\_one = array\_generator(number\_matrix);  matrix\_two = array\_generator(number\_matrix);  matrix\_generate(matrix\_one, number\_matrix);  matrix\_generate(matrix\_two, number\_matrix);  do  {  std::cout << "Введите количество потоков:";  std::cin >> threads;  if (threads > 0)  {  break;  }  else  {  std::cout << "Проверьте входные данные! Размерность должна быть положительна!\n";  }  } while (true);  // Последовательное умножение  sequential\_matrix\_multiplication(matrix\_one, matrix\_two, number\_matrix);  // Распараллеливание умножения матриц  parallel\_matrix\_multiplication(matrix\_one, matrix\_two, number\_matrix, threads);  array\_destroyer(matrix\_one, number\_matrix);  array\_destroyer(matrix\_two, number\_matrix);  break;  case 0:  // Завершение программы  std::cout << "Программа завершена";  break;  default:  std::cout << "Неверно введена комманда\n";  break;  }  } while (command);  } |

Листинг файла main.cpp. Исходный код программы.

# **Тестирование**

Тестирование программы происходит при размерности матрицы n = 3, при количестве потоков p = 2. Результат выполнения программы представлен на рисунке 6.



*Рисунок 2. Результаты тестирования*.

# **Контрольные прогоны**

На основе выполненных тестов были составлены сводные таблицы, на основе которых составлены графики, указанные в основной задаче.

*Таблица 1. Результаты вертикального алгоритма (columnwize) методов без распараллеливания*

|  |  |  |
| --- | --- | --- |
| № тестирования | Размер матрицы,  n | Время выполнения работы программы, сек. |
| 1 | 500 | 0.741 |
| 2 | 600 | 1.416 |
| 3 | 700 | 2.255 |
| 4 | 800 | 3.934 |
| 5 | 900 | 5.97 |
| 6 | 1000 | 9.418 |

*Таблица 2 Результаты работы вертикального алгоритма (columnwize) при количестве потоков p = 2*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № тестирования | Размер матрицы, n | Время выполнения программы, сек | Ускорение (Sn) | Эффективность | Стоимость |
| 1 | 500 | 0.373 | 1.986 | 0.993 | 0.746 |
| 2 | 600 | 0.767 | 1.846 | 0.923 | 1.534 |
| 3 | 700 | 1.273 | 1.771 | 0.885 | 2.546 |
| 4 | 800 | 2.007 | 1.976 | 0.98 | 4.014 |
| 5 | 900 | 3.1 | 1.925 | 0.962 | 6.2 |
| 6 | 1000 | 4.766 | 1.976 | 0.988 | 9.532 |

*Таблица 3. Результаты работы вертикального алгоритма (columnwize) при количестве потоков p = 4*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № тестирования | Размер матрицы, n | Время выполнения программы, сек | Ускорение (Sn) | Эффективность | Стоимость |
| 1 | 500 | 0.199 | 3.723 | 0.93 | 0.796 |
| 2 | 600 | 0.372 | 3.806 | 0.951 | 1.488 |
| 3 | 700 | 0.579 | 3.894 | 0.973 | 2.316 |
| 4 | 800 | 0.986 | 3.989 | 0.997 | 3.944 |
| 5 | 900 | 1.535 | 3.889 | 0.972 | 6.14 |
| 6 | 1000 | 2.495 | 3.774 | 0.943 | 9.98 |

*Таблица 4. Результаты работы вертикального алгоритма (columnwize) при количестве потоков p = 8*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № тестирования | Размер матрицы, n | Время выполнения программы, сек | Ускорение (Sn) | Эффективность | Стоимость |
| 1 | 500 | 0.212 | 3.495 | 0.436 | 1.696 |
| 2 | 600 | 0.326 | 4.343 | 0.542 | 2.608 |
| 3 | 700 | 0.538 | 4.191 | 0.523 | 4.304 |
| 4 | 800 | 0.818 | 4.809 | 0.601 | 6.544 |
| 5 | 900 | 1.238 | 4.822 | 0.602 | 9.904 |
| 6 | 1000 | 2.148 | 4.384 | 0.548 | 17.184 |

*Таблица 5. Результаты работы вертикального алгоритма (columnwize) при количестве потоков p = 16*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № тестирования | Размер матрицы, n | Время выполнения программы, сек | Ускорение (Sn) | Эффективность | Стоимость |
| 1 | 500 | 0.247 | 3 | 0.185 | 3.952 |
| 2 | 600 | 0.319 | 4.43 | 0.277 | 5.104 |
| 3 | 700 | 0.547 | 4.122 | 0.257 | 8.752 |
| 4 | 800 | 0.824 | 2.156 | 0.134 | 29.184 |
| 5 | 900 | 1.208 | 4.94 | 0.308 | 19.328 |
| 6 | 1000 | 2.267 | 2.69 | 0.168 | 56 |

*Таблица 6. Результаты вертикального алгоритма (columnwize) при количестве потоков p = 32*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № тестирования | Размер матрицы, n | Время выполнения программы, сек | Ускорение (Sn) | Эффективность | Стоимость |
| 1 | 500 | 0.187 | 3.962 | 0.123 | 5.984 |
| 2 | 600 | 0.347 | 4.080 | 0.127 | 11.104 |
| 3 | 700 | 0.542 | 4.16 | 0.13 | 17.344 |
| 4 | 800 | 0.839 | 4.688 | 0.146 | 26.848 |
| 5 | 900 | 1.235 | 4.834 | 0.151 | 39.52 |
| 6 | 1000 | 2.238 | 4.208 | 0.131 | 71.616 |

*Рисунок 3. Зависимости ускорения при различных потоках от размерности матрицы*

На основе данного рисунка можно сделать вывод о том, что ускорение происходит на небольших величинах больше всего при выполнении алгоритма на 32 потока.

*Рисунок 4. Зависимости времени выполнения при различных потоках от количества элементов матрицы*

На основе данного рисунка можно сделать вывод о том, что самое быстрое выполнение происходит при использовании 32-х поточной программой.

*Рисунок 5. Зависимости эффективности выполнения при различных потоках от количества элементов матрицы*

На основе данного рисунка можно сделать вывод о том, самая высокая эффективность достигается при 4 поточном вычислений значений. Худший вариант – использование 32 потоков.

*Рисунок 6. Зависимости стоимости выполнения при различных потоках от количества элементов матрицы*

На основе данного рисунка можно сделать вывод о том, самая высокая стоимость выполнения достигается при использовании 32 потоков, самая маленькая – при использовании 2 потоков.

# **Выводы**

В результате выполнения лабораторной работы был реализован вертикальный (columnwize) алгоритм разбивки матрицы. Были построены графики времени работы программы двух разных алгоритмов, графики зависимости ускорения от размерности и количества потоков, были рассчитаны эффективности распараллеливания и построены наглядные графики.

1. Максимальное ускорение для n = 500 достигается при p = 32;
2. Максимальное ускорение для n = 600 достигается при p = 16;
3. Максимальное ускорение для n = 700 достигается при p = 8;
4. Максимальное ускорение для n = 800 достигается при p = 8;
5. Максимальное ускорение для n = 900 достигается при p = 16;
6. Максимальное ускорение для n = 1000 достигается при p = 8;
7. Для всех размерностей наибольшая эффективность достигается при p = 4;

# **ЗАДАНИЕ №4**

Установка и настройка интегрированной среды распределенного программирования C++ и MPICH

# **Постановка задачи**

1. Провести обоснованный выбор версии (издания) MPI на основе сопоставления технических характеристик компьютера и версии используемой операционной системы Windows c требованиями, предъявляемыми MPI.

2. Скопировать (загрузить) с официального сайта производителя бесплатно распространяемую версию программного интерфейса передачи сообщений MPI, например, MPICH [2] и произвести ее установку на свой компьютер. Ссылку на официальный сайт привести в списке используемых информационных источников - в обязательном порядке.

3. Создать и настроить проект Microsoft Visual Studio для работы с библиотекой MPI.

4. Написать тестовый проект распределенного приложения для работы с библиотекой MPI, который выводит на экран монитора сообщение «Hello World!» количеством процессов «по умолчанию».

5. Произвести тестовые контрольные прогоны программы с установленным количеством процессов 8, 16 и 64.

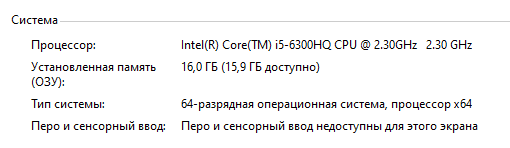
6. Определить максимальное количество процессов, которое может быть установлено на компьютере.

7. Сделать выводы по полученным результатам и оформить отчет по выполненной работе.

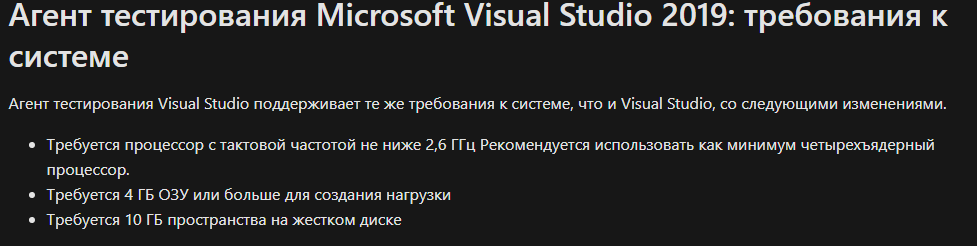
8. Привести характеристики компьютера и его процессора, на котором производилась установка инструментальной среды (в том числе, в обязательном порядке: производителя, марку, тактовую частоту, количество ядер, наличие технологии hyper treading, наличие сопроцессора GPU или количество и длину векторных регистров).

# **Обоснование выбора**

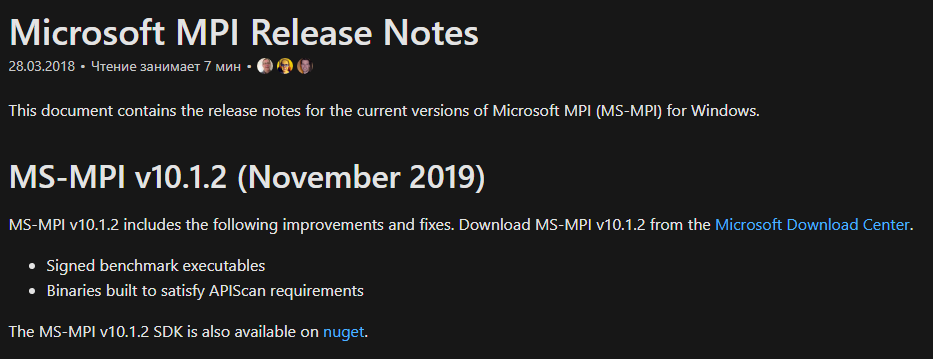
Для выполнения данной практической работы было необходимо провести сопоставление технических характеристик компьютера (рисунок 1), версии Visual Studio (рисунок 2) и рекомендованных систем требований для установки пакета MPI (рисунок 3).



*Рис. 1. Технические характеристики ПК, на котором установлена ОС.*



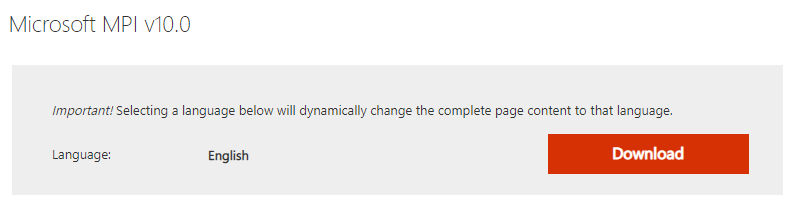
*Рис. 2. Системные требования Microsoft visual studio 2019.*



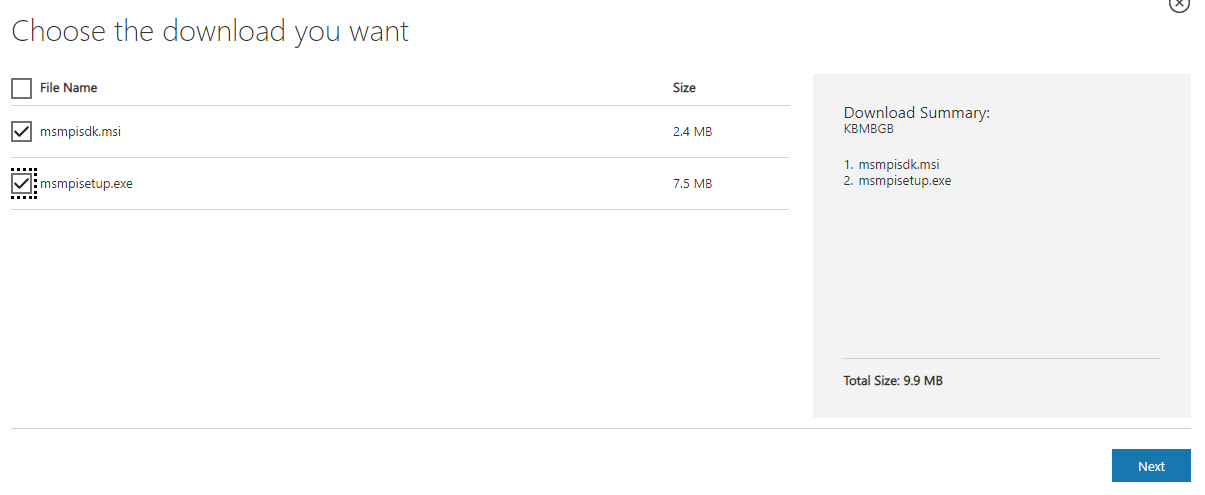
*Рис. 3. Корректная версия пакета MPI с официального сайта Microsoft.*

# **Загрузка пакета MPI**

Для того, чтобы произвести загрузку пакета, необходимо перейти по ссылке, указанной в информационных источниках, после чего нажать на кнопку Download (рисунок 4). После этого необходимо скачать оба файла с расширениями msi, exe (рисунок 5).



*Рис. 4. Официальная страница с пакетом MPI v10.0 на сайте Microsoft.*

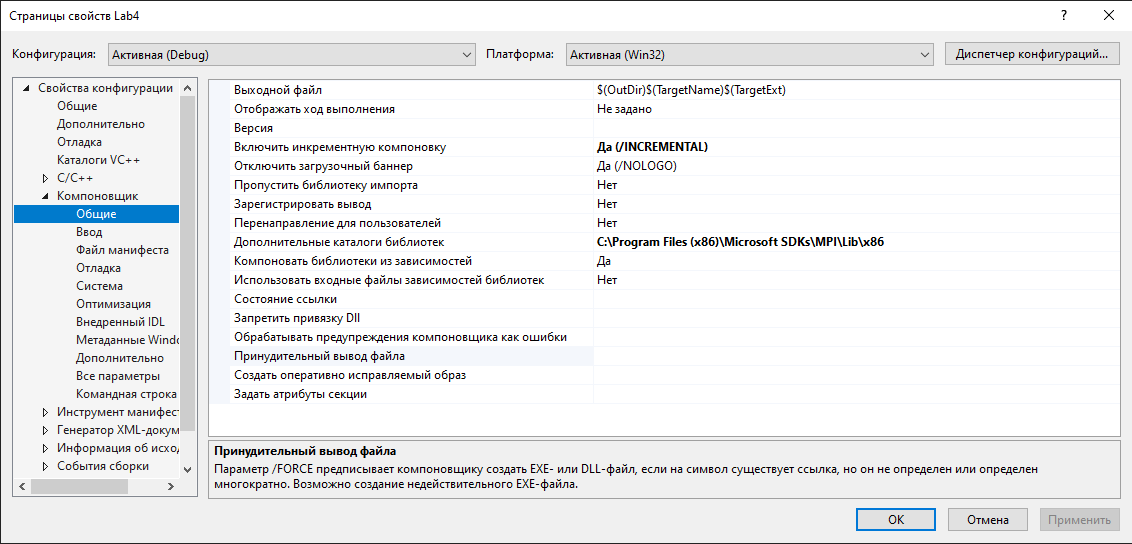


*Рис. 5. Основные файлы пакета MPI, которые необходимо скачать для последующей инсталляции.*

# **Создание и настройка проекта с использованием MPI**

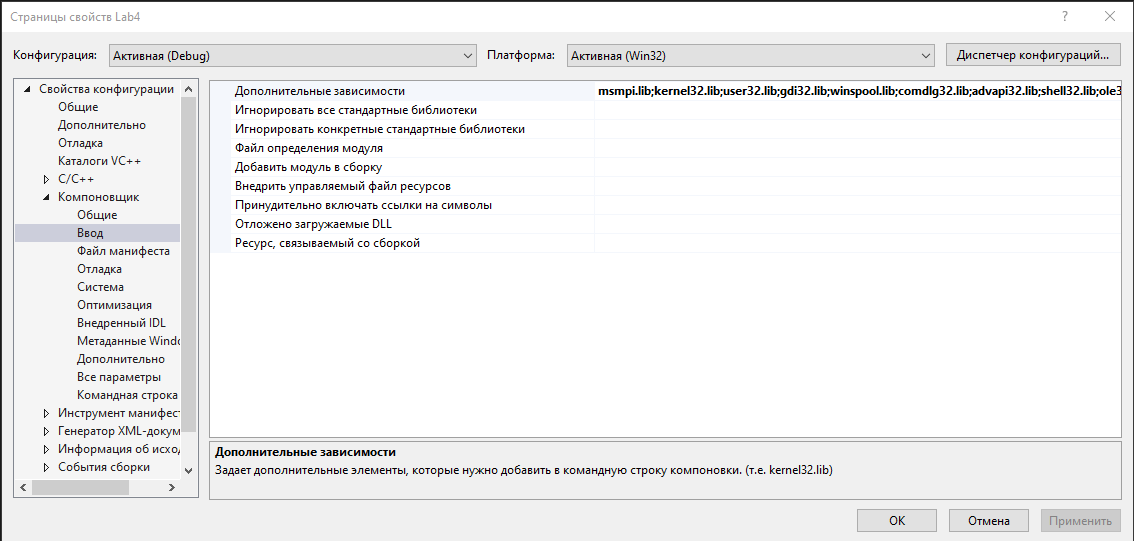
Для создания и настройки проекта в IDE MVS 2019 для работы с пакетом MPI, необходимо провести следующие действия:

1. Провести установку ранее скачанных файлов (msi и exe). Msi необходим для дальнейшего подключения самой библиотеки mpi.h, файл exe – для запуска собранной программы.
2. Перейти в IDE MVS, создать пустой проект на языке программирования C++, добавить главный файл index.cpp.
3. Далее – перейти в раздел «Отладка -> свойство отладки». Слева выбрать вкладку «Компоновщик» и вставить в параметр «Дополнительные параметры библиотек» следующий путь: C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x86 (рисунок 6).



*Рис. 6. Настройка IDE MVS (1).*

1. Затем – во вкладке «Компоновщик», перейти в раздел ввод, где в поле «Дополнительные зависимости» дописать в начало строки msmpi.lib; (рисунок 7).



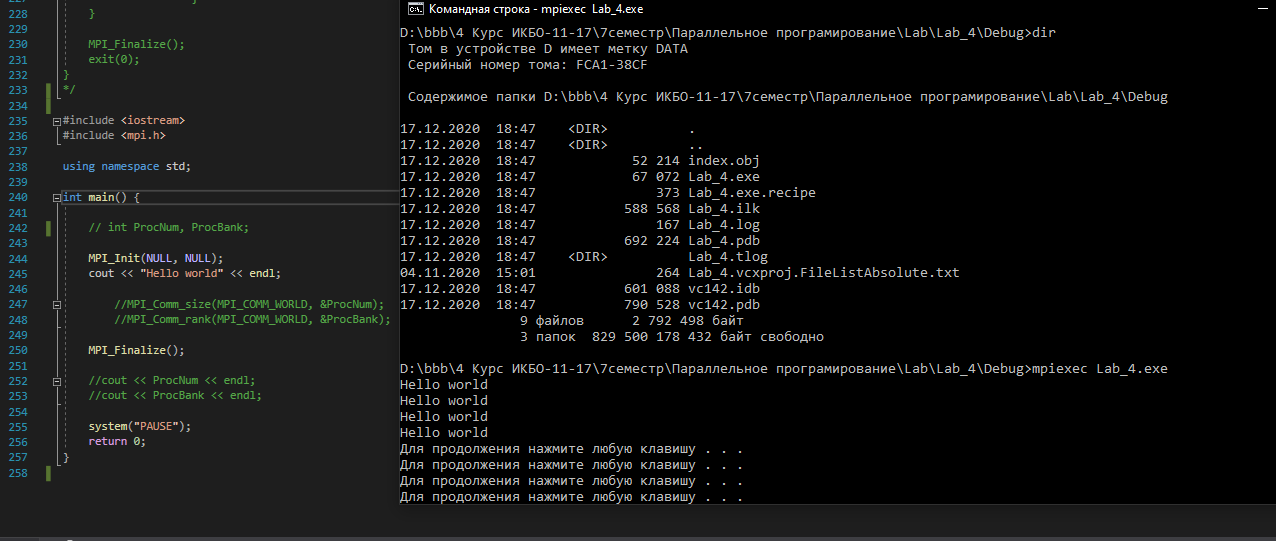
*Рис. 7. Настройка IDE MVS (2).*

1. Последним шагом необходимо написать в index.cpp в начале кода строку #include <omp.h>

# **Написание тестового проекта**

Для написания первой программы необходимо в главной функции программы объявить функции MPI\_Init (NULL, NULL), после которой уже достаточно писать код, после которого в обязательном порядке необходимо прописать MPI\_Finalize (). После этого нужно произвести сборку проекта. Затем – в командной строке перейти в папку со скомпилированным файлом exe и написать команду mpiexec <имя файла.exe>.

Напишем простейшую программу, выводящую в консоль надпись «Hello world» (рисунок 8). Количество процессов по умолчанию – 4 так как в текущем процессоре имеется 4 ядра.

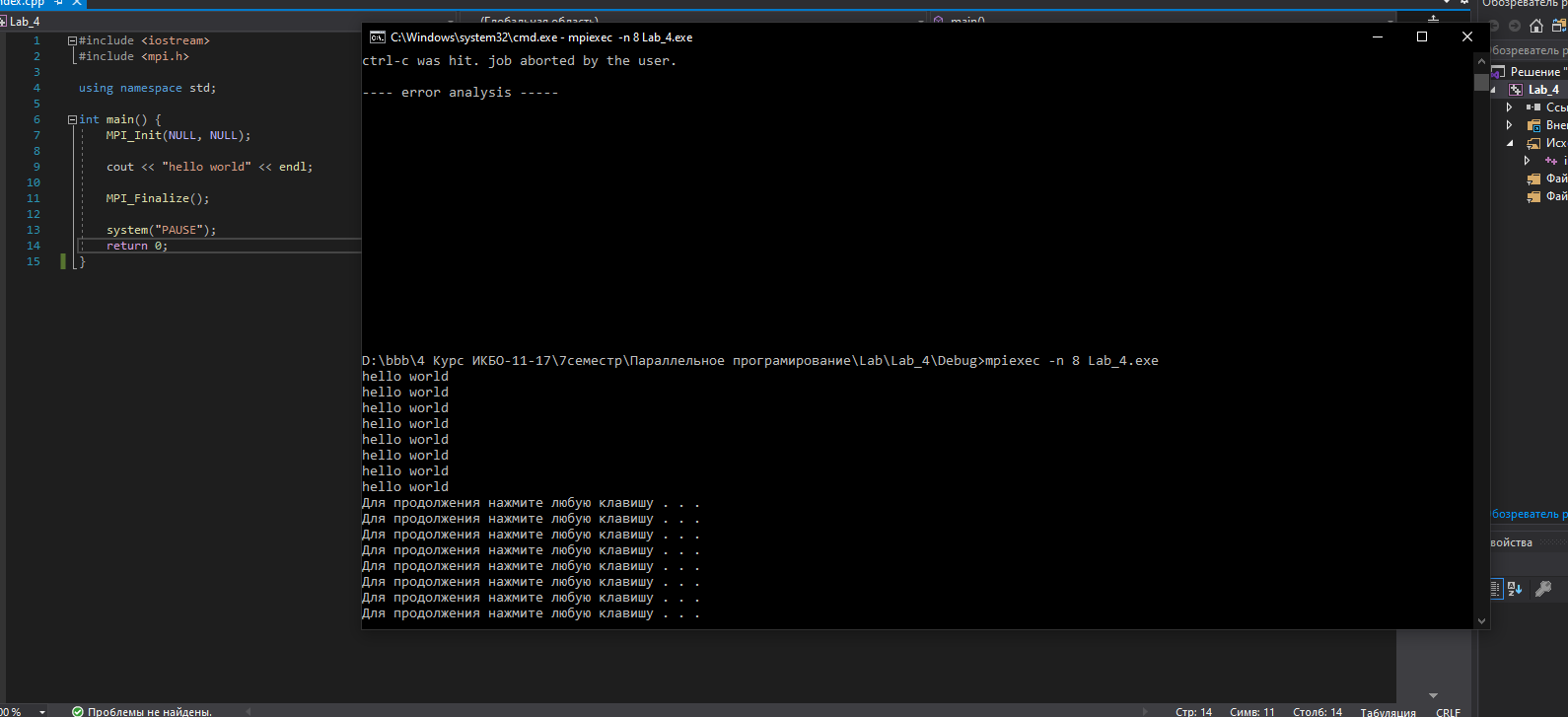


*Рис. 8. Простейшая программа с пакетом MPI, выводящая в консоль «Hello world».*

# **Контрольные прогоны программы**

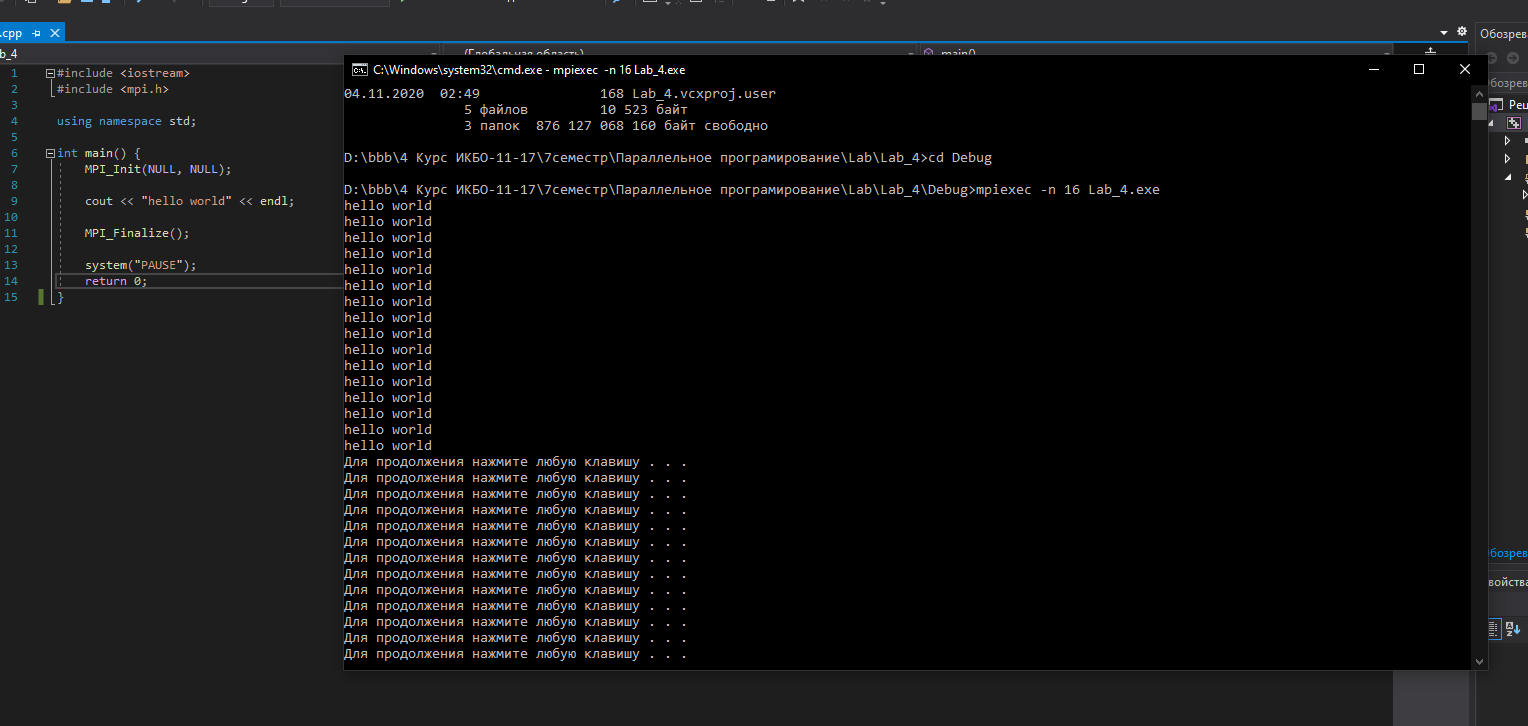
Для того, чтобы запустить программы с различным числом потоков с использованием пакета MPI, достаточно в консоли после сборки проекта прописать следующую команду: mpiexec -n <кол-во процессов> <имя файла.exe>.

Ниже продемонстрирована программа с использованием 8 процессов (рисунок 9).



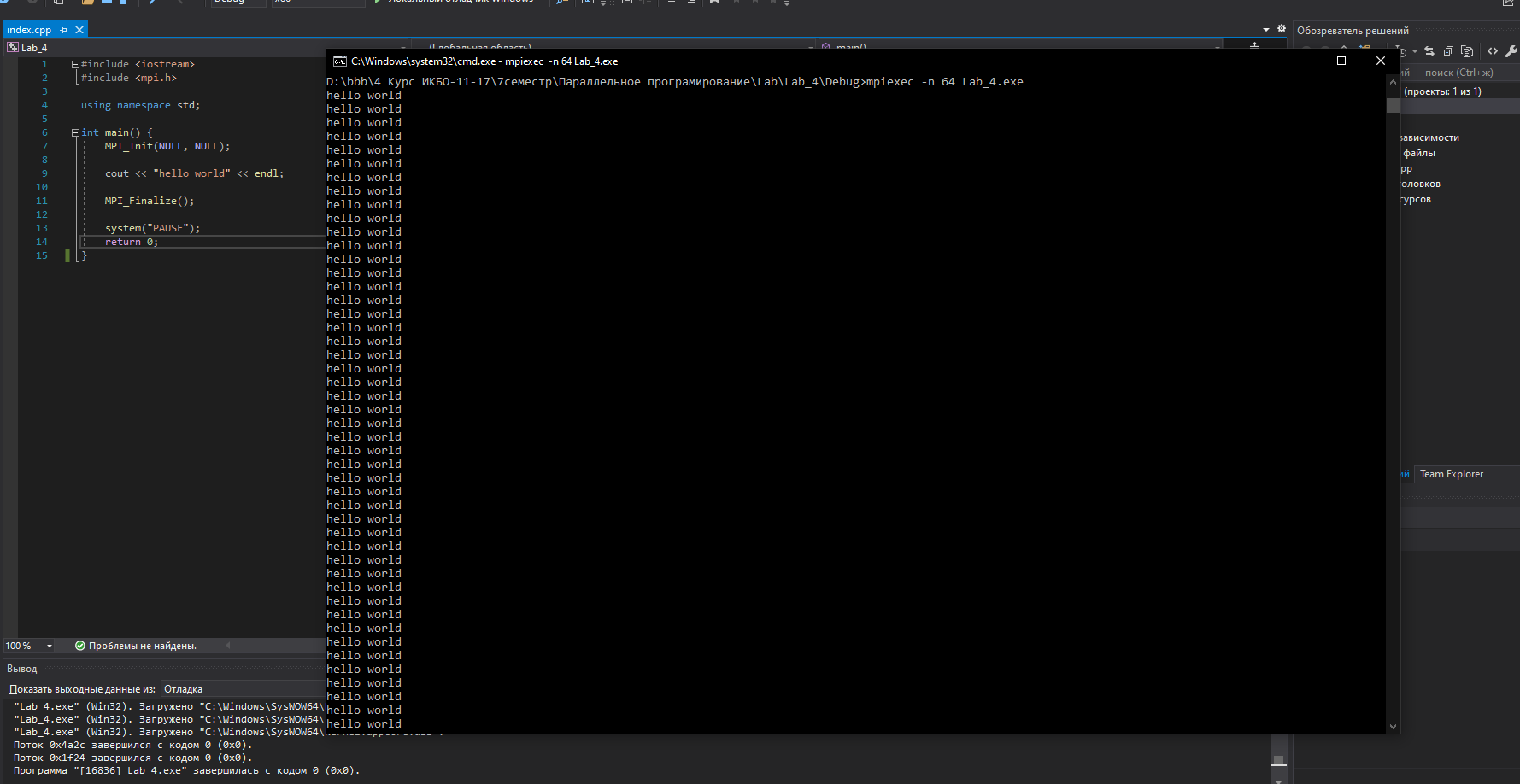
*Рис. 9. Программа, написанная с помощью библиотеки MPI, использующая 8 процессов.*

Ниже продемонстрирована программа с использованием 16 процессов (рисунок 10).



*Рис. 10. Программа, написанная с помощью библиотеки MPI, использующая 16 процессов.*

Ниже продемонстрирована программа с использованием 64 процессов (рисунок 11).



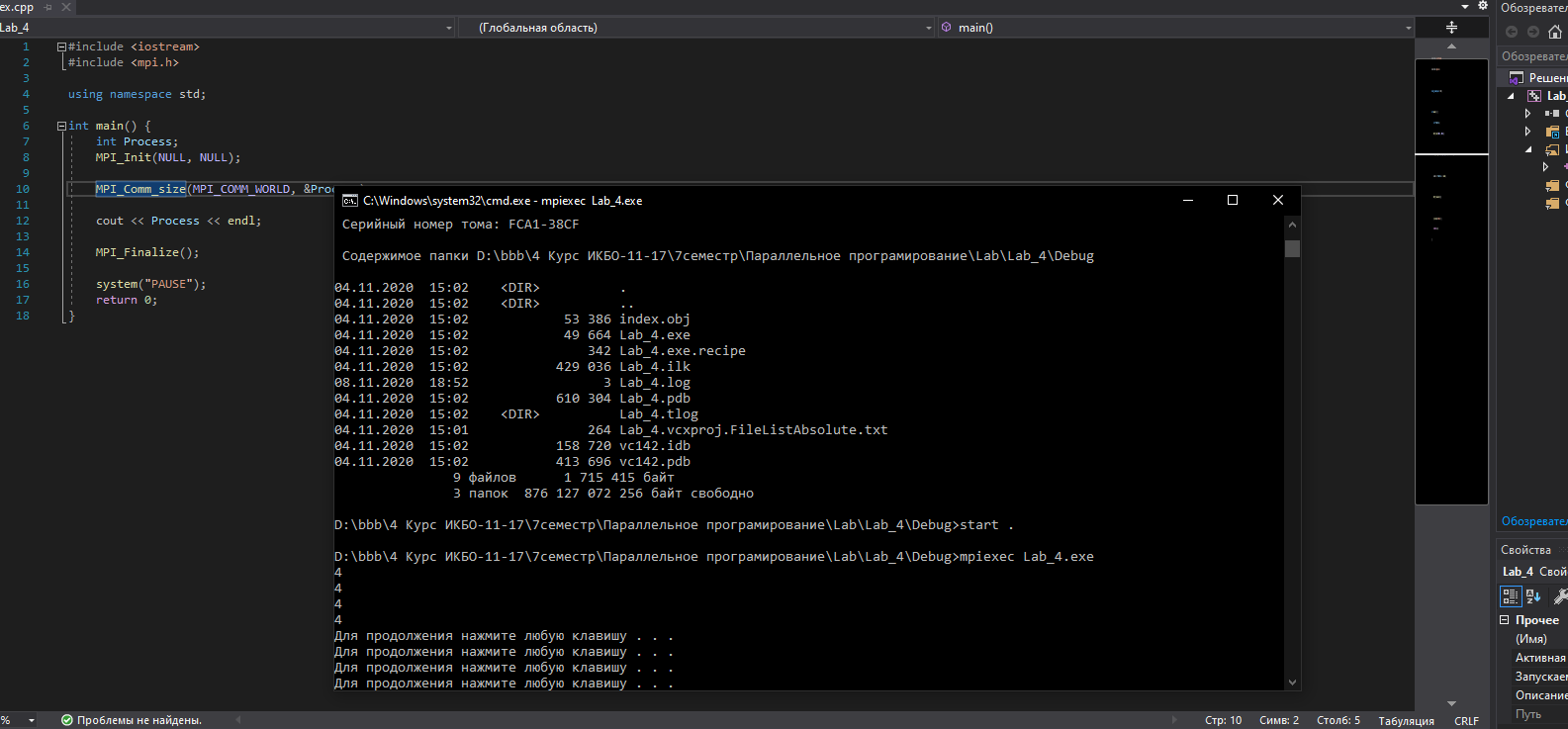
*Рис. 11. Программа, написанная с помощью библиотеки MPI, использующая 64 процесса.*

# **Максимальное количество процессов**

Для вывода максимального количества процессов служит функция int MPI\_Comm\_size(MPI\_Comm comm, int \*size).

• comm — коммуникатор, размер которого определяется,

• size — определяемое количество процессов в коммуникаторе.

*Рис. 12. Максимальное количество процессов.*

# **Выводы**

В ходе выполнения данной практической работы произвел знакомство с пакетом MPI для языка C++, написана простейшая программа при помощи данного пакета, выполнены задачи из раздела «постановка задачи».

# **ЗАДАНИЕ №5**

Распределенное программирование для систем с общей памятью

с использованием основ технологии MPI

# **Постановка задачи**

Вариант 5.7

Составить программу последовательного и параллельного вычисления определённого интеграла методом Симпсона. Провести контрольные прогоны программы для числа разбиений отрезка интегрирования n = 104, 105, 106, 107, 108 и установленных для параллельного варианта количествах потоков p = 4, 8, 16, 32 и 64 с вычислением времени выполнения и ускорения. Полученные результаты свести в сводную таблицу.

Построить графики изменения ускорения при последовательном и параллельных вычислениях в зависимости от числа разбиений отрезка интегрирования. Построить графики изменения ускорения при параллельных вычислениях в зависимости от количества используемых потоков. Вычислить показатели эффективности и стоимости параллельной реализации программы. Провести анализ полученных результатов. Сделать выводы о проделанной работе, основанные на полученных данных.

# **Описание алгоритмов, используемых для решения задачи**

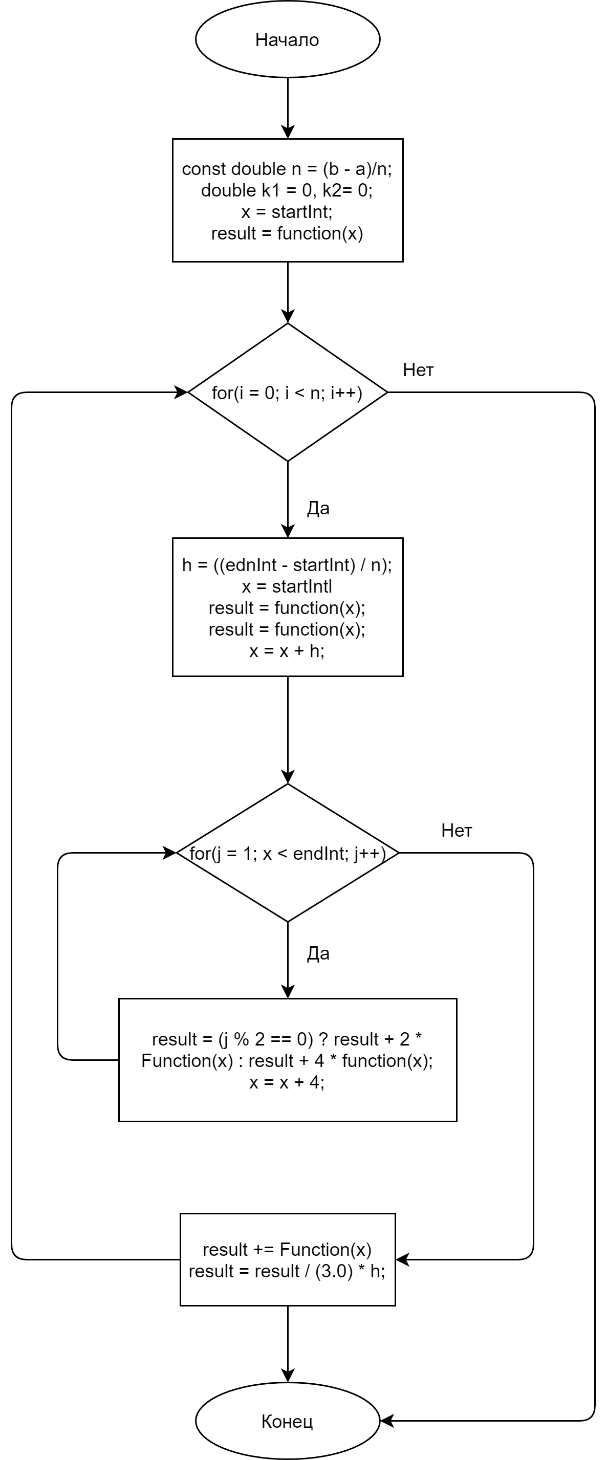
Для выполнения данной практической работы было решено провести декомпозицию общего процесса на более мелкие составляющие (рисунок 2).



*Рисунок 2. Декомпозиция общей задачи на подзадачи*

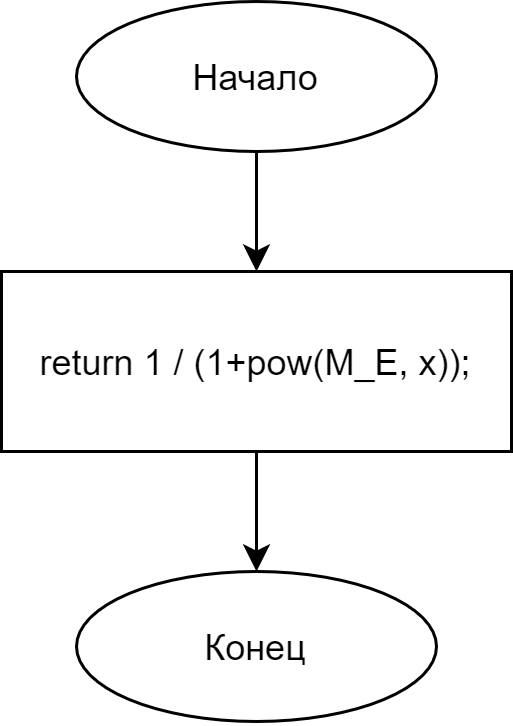
После проведения декомпозиции опишем использование всех алгоритмов, используемых в решении общей задачи.

Во-первых, ниже продемонстрирован рисунок схемы алгоритма работы программы (рисунок 3).



*Рисунок 3. Общего алгоритма работы программы*

Во-третьих, продемонстрирован рисунок схемы алгоритма вычисления значения функции в точке (рисунок 4).



*Рисунок 4. Схема алгоритма вывода значения функции в точке*

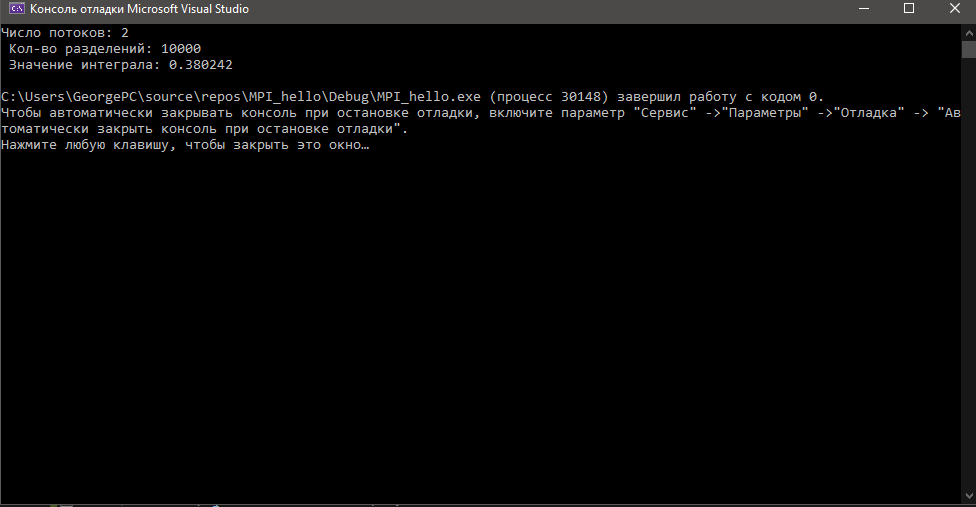
# **Текст исходного кода программы**

|  |
| --- |
| #include <math.h>  #include <mpi.h>  #include <iostream>  using namespace std;  #define MASTER 0  double Function(double x);  int main(int argc, char\*\* argv) {  int myrank, nprocs, a, b, i;  long n, j;  double startInt, endInt, result, h, x, endresult;  setlocale(LC\_ALL, "Rus");  MPI\_Status recv\_status;  MPI\_Request request;  MPI\_Init(&argc, &argv);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &nprocs);  a = 0;  b = 1;  startInt = (double)(b - a) / nprocs \* myrank + a;  endInt = (double)(b - a) / nprocs \* (myrank + 1.0) + a;  for (i = 0; i < n; ++i) {  MPI\_Barrier(MPI\_COMM\_WORLD);  time = MPI\_Wtime();  h = ((endInt - startInt) / n);  x = startInt;  result = function(x);  x = x + h;  for (j = 1; x < endInt; ++j) {  result = (j % 2 == 0) ? result + 2 \* Function(x) :  result + 4 \* function(x);  x = x + h;  }  result += Function(x);  result = result / (3.0) \* h;  MPI\_Reduce(&result, &endresult, 1, MPI\_DOUBLE, MPI\_SUM,  MASTER, MPI\_COMM\_WORLD);  time = MPI\_Wtime() - time;  MPI\_Barrier(MPI\_COMM\_WORLD);  MPI\_Reduce(&time, &slowest, 1, MPI\_DOUBLE, MPI\_MAX, MASTER,  MPI\_COMM\_WORLD);  if (myrank == MASTER) {  times[i] = slowest;  }  }  if (myrank == MASTER) {  cout << "Num thread: " << nprocs << " Partition: " << n  << " Min in thread: " << time << " Res: " <<  endresult;  }  MPI\_Finalize();  }  double function(double x) {  double result = 1 / (1 + pow(2.71, x));  if (isnan(result)) {  result = 0.0f;  }  return result;  } |
|  |

Листинг файла index.cpp. Исходный код программы.

# **Контрольные прогоны программы**

Контрольные прогоны программы происходят при количестве разбиений n = 10000, при количестве потоков p = 2. Результат выполнения программы представлен на рисунке 5.



*Рисунок 5. Результаты тестирования*.

# **Анализ полученных результатов**

Далее – были составлены сводные таблицы по сравнению ключевых параметров работы алгоритмов.

*Таблица 1. Время работы последовательного и параллельного алгоритма вычисления интеграла методом Симпсона*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Количество элементов, n | Время при последовательном вычислении, с | Время работы при  P = 4 | Время работы при  P=8 | Время работы при  P=16 | Время работы при  P=32 | Время работы при  P=64 |
| 10000 | 4.273 | 2.268 | 1.667 | 0.997 | 0.989 | 0.844 |
| 100000 | 14.275 | 7.386 | 5.448 | 3.329 | 3.258 | 2.71 |
| 1000000 | 33.22 | 17.083 | 12.292 | 7.845 | 7.653 | 6.956 |
| 10000000 | 64.128 | 33.126 | 23.835 | 15.145 | 14.85 | 12.134 |
| 100000000 | 110.822 | 56.774 | 41.038 | 26.058 | 25.422 | 23.095 |

На основании этих данных был построен общий график зависимости время выполнения алгоритма от размера массива (рисунок 6).

На основании этого графика можно сделать следующие вывод, что наиболее быстрая работа достигается при работе программы с числом потоков равным 64. Медленнее же всего работает последовательный алгоритм

*Рисунок.6. Графики зависимости время работы программы от размерности массива*

Далее, была составлена сводная таблица изменения ускорения от количества элементов массива

*Таблица 2. Изменение ускорения параллельной реализацией вычисления интеграла методом Симпсона*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество элементов, n | Ускорение Sn, при P = 4 | Ускорение Sn, при P=8 | Ускорение Sn, при P=16 | Ускорение Sn, при P=32 | Ускорение Sn, при P=64 |
| 1000 | 1,8779 | 1,618 | 4,293 | 4,3395 | 4,0626 |
| 10000 | 1,9352 | 2,623 | 4,29 | 4,3849 | 5,1527 |
| 100000 | 1,9344 | 2,691 | 4,225 | 4,3261 | 5,1659 |
| 1000000 | 1,9334 | 2,679 | 4,238 | 4,3174 | 5,2838 |
| 10000000 | 1,9353 | 2,677 | 4,214 | 4,3192 | 5,2853 |

На основании этих данных был построен общий график зависимости ускорения алгоритма от размера массива (рисунок 7).

*Рисунок.7. Графики зависимости ускорения от кол-ва разбиений*

Далее, была составлена сводная таблица изменения эффективности от количества разбиений отрезков

*Таблица 3. Изменение эффективности параллельной реализации вычисления интеграла методом Симпсона*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество элементов, n | Эффективность En, при P = 4 | Эффективность En, при P=8 | Эффективность En, при P=16 | Эффективность En, при P=32 | Эффективность En, при P=64 |
| 1000 | 0,4679 | 0,327 | 0,2684 | 0,1356 | 0,0791 |
| 10000 | 0,4833 | 0,3277 | 0,2681 | 0,137 | 0,08051 |
| 100000 | 0,4846 | 0,3365 | 0,2638 | 0,1352 | 0,07437 |
| 1000000 | 0,4838 | 0,3362 | 0,2645 | 0,14492 | 0,08255 |
| 10000000 | 0,4835 | 0,3345 | 0,2634 | 0,13513 | 0,08335 |

На основании этих данных был построен общий график зависимости

эффективности алгоритма от кол-ва разбиений отрезка интегрирования (рисунок 8).

*Рис.8. Графики зависимости эффективности от количества разбиений*

*Таблица 4. Изменение стоимости параллельной реализации вычисления интеграла методом Симпсона*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Количество элементов, n | Стоимость Cn, при P = 4 | Стоимость Cn, при P=8 | Стоимость Cn, при P=16 | Стоимость Cn, при P=32 | Стоимость Cn, при P=64 |
| 1000 | 9,152 | 13,096 | 15,952 | 31,584 | 54,144 |
| 10000 | 29,532 | 43,544 | 53,232 | 104,16 | 177,28 |
| 100000 | 68,324 | 98,368 | 125,504 | 244,896 | 445,184 |
| 1000000 | 132,504 | 190,68 | 242,32 | 475,2 | 776,576 |
| 10000000 | 227,096 | 328,304 | 416,928 | 813,504 | 1478,08 |

На основании этих данных был построен общий график зависимости

стоимости алгоритма от кол-ва разбиений отрезка интегрирования (рисунок 9).

*Рисунок.9. Графики зависимости стоимости от количества элементов массива*

# **Выводы**

В ходе выполнения данной практической работы реализовал цели данной работы и соответствующего раздела, углубленно познакомился с пакетом MPI, изменил алгоритм работы программы в соответствии с синтаксисом и принципом работы пакета, провел результаты, по которым провел соответствующие выводы.

1. Максимальное ускорение для n = 1000 достигается при p = 32
2. Максимальное ускорение для n = 10000 достигается при p = 64
3. Максимальное ускорение для n = 100000 достигается при p = 64
4. Максимальное ускорение для n = 1000000 достигается при p = 64
5. Максимальное ускорение для n = 10000000 достигается при p = 64
6. Самым эффективный алгоритм, работающий на p = 4, самый неэффективны работающий на p = 64
7. Самым не затратным алгоритм, работающий на p = 4, самый затратный алгоритм работающий на p = 64

# **ЗАДАНИЕ №6**

Параллельное программирование с использованием расширенных средств технологии OpenMP (редукции, атомарных операций, критических секций, замков).

# **Постановка задачи**

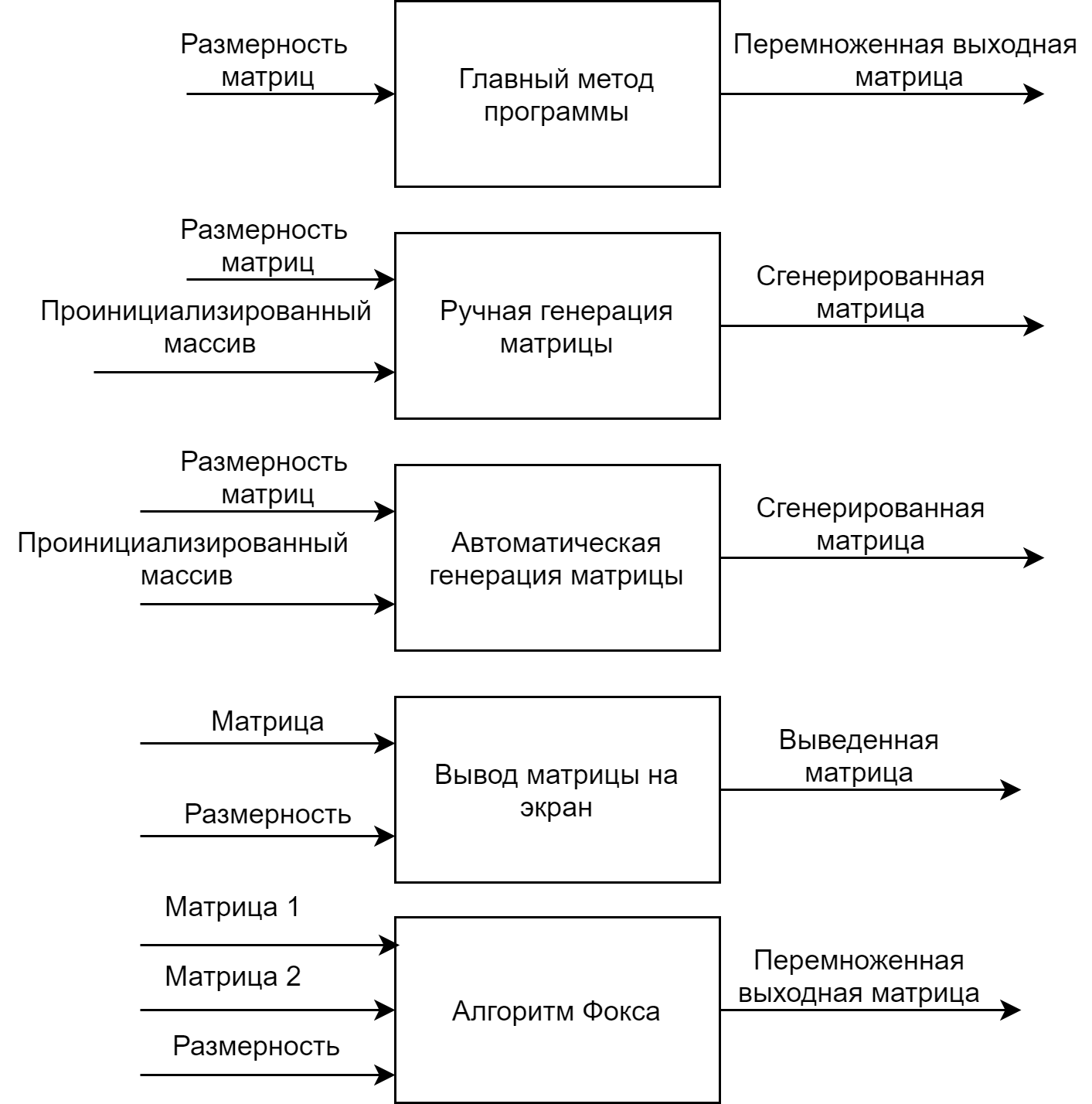
Вариант 6.13

Составить программу последовательного и распределенного вариантов умножения двух матриц A[n,n] и В[n,n], используя блочный алгоритм Фокса. Провести тестирование программы в последовательном и распределенном исполнениях на матрицах размером n = 3, сформированных вводом с клавиатуры. Разработать форму представления результирующей матрицы на экране монитора.

Рабочие матрицы А и В сформировать, используя генератор псевдослучайных чисел. Провести контрольные прогоны программы для размеров n = 600, 700, 800, 900, 1000 в последовательном и распределенном исполнениях с количеством процессов p = 2, 4, 8, 16, 32. Полученные результаты свести в сводную таблицу. Построить графики изменения ускорения умножения в зависимости от размеров матриц. Построить графики изменения ускорения распределенных вычислений с разным количеством процессов. Вычислить показатели эффективности и стоимости распределенной реализации программы. Провести анализ полученных результатов. Сделать выводы по проделанной работе, основанные на полученных данных.

# **Описание алгоритмов, используемых для решения задачи**

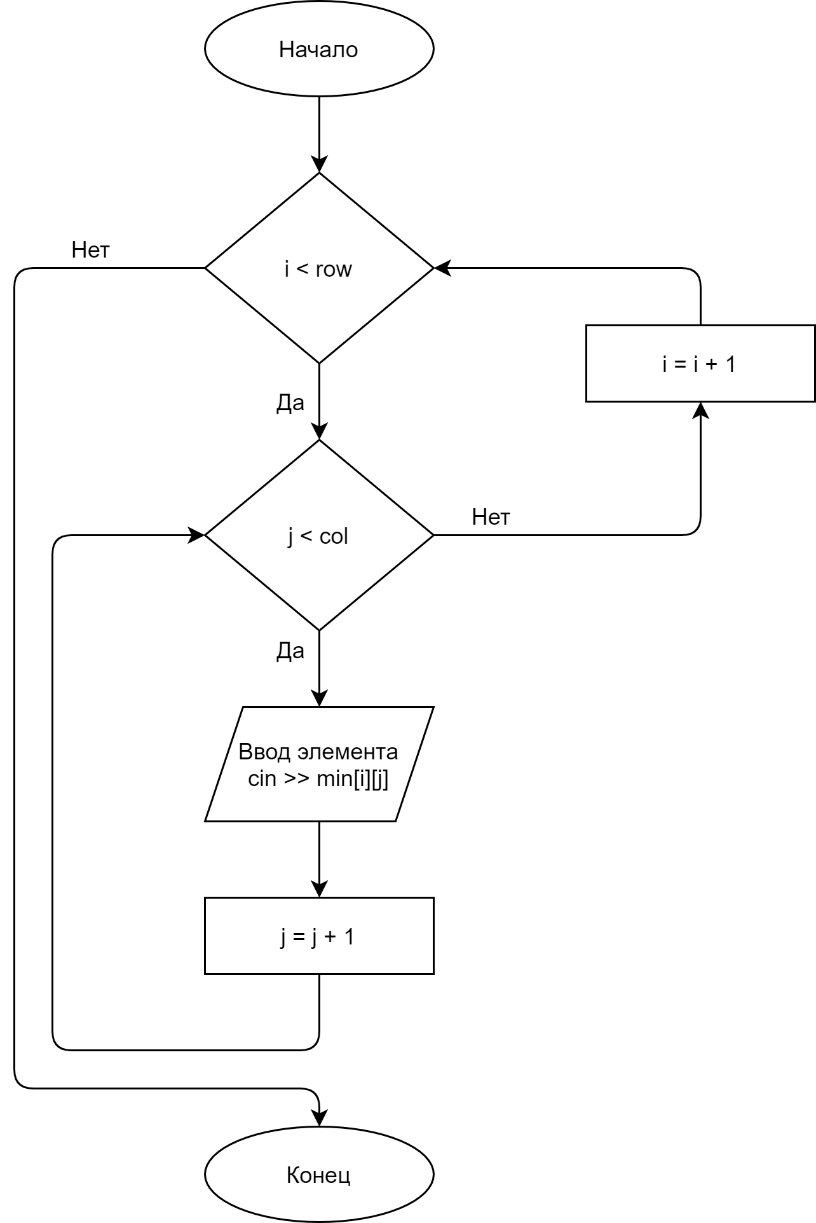
Для выполнения данной практической работы было решено провести декомпозицию общего процесса на более мелкие составляющие (рисунок 2).



*Рисунок. 2. Декомпозиция общей задачи на подзадачи*

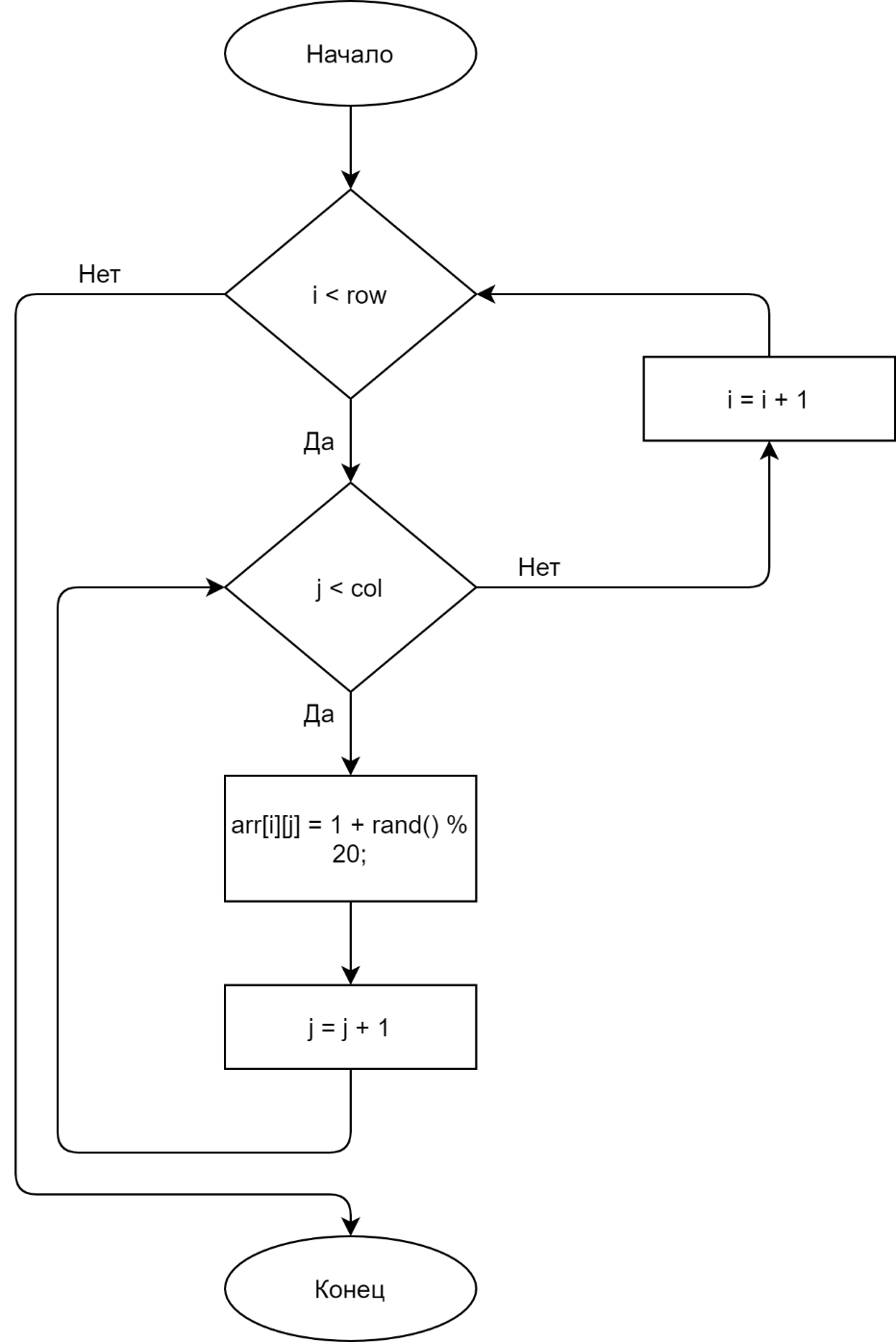
После проведения декомпозиции опишем использование всех алгоритмов, используемых в решении общей задачи.

Во-первых, ниже продемонстрирован рисунок алгоритма ручного заполнения матрицы (рисунок 3).



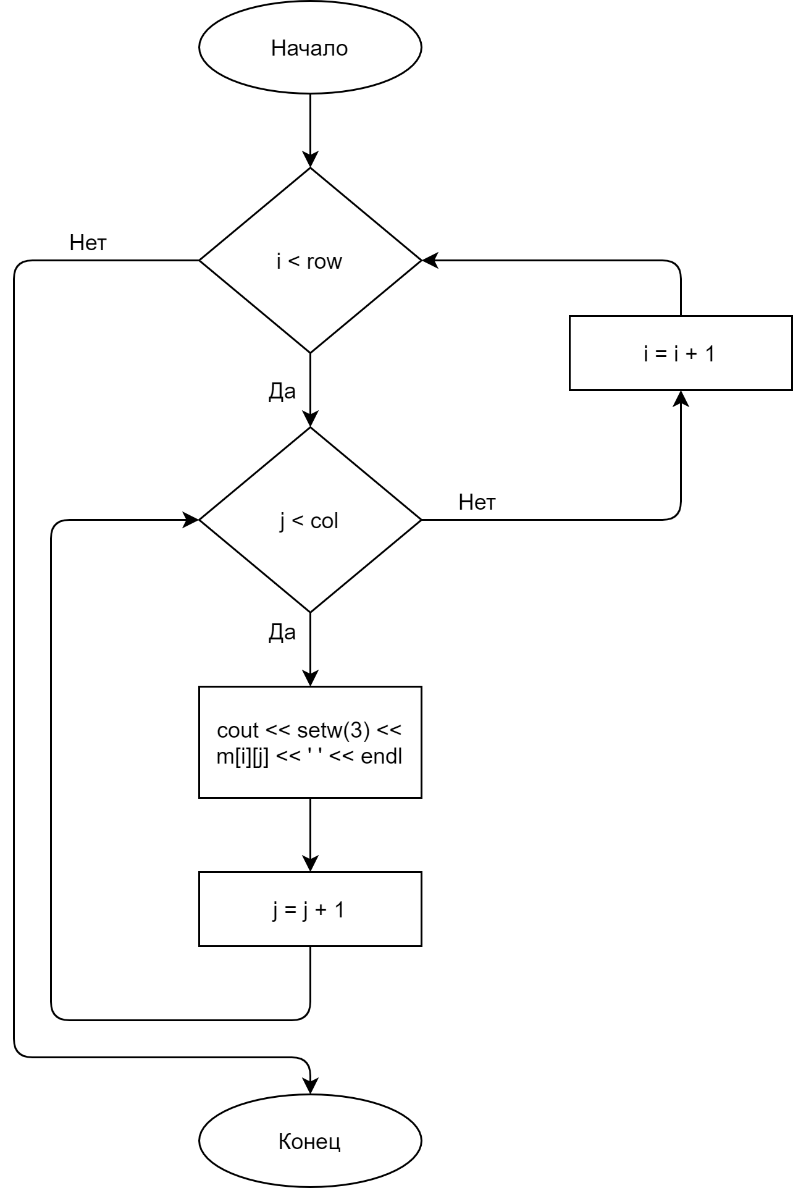
*Рисунок. 3. Схема алгоритма ручного заполнения матрицы*

Во-вторых, ниже продемонстрирован рисунок алгоритма автоматического заполнения матрицы (рисунок 4).



*Рисунок. 4. Схема алгоритма автоматического заполнения матрицы*

Во-третьих, ниже продемонстрирован рисунок алгоритма вывода матрицы на экран пользователя (рисунок 5).



*Рисунок. 5. Схема алгоритма вывода матрицы на экран*

Используется блочная схема разбиения матриц. При таком способе разделения данных исходные матрицы А, В и результирующая матрица С представляются в виде наборов блоков. Далее предполагается что все матрицы являются квадратными размера n×n, количество блоков по горизонтали и вертикали одинаково и равно q (т.е. размер всех блоков равен k×k, k=n/q).

В соответствии с алгоритмом Фокса в ходе вычислений на каждой базовой подзадаче (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:

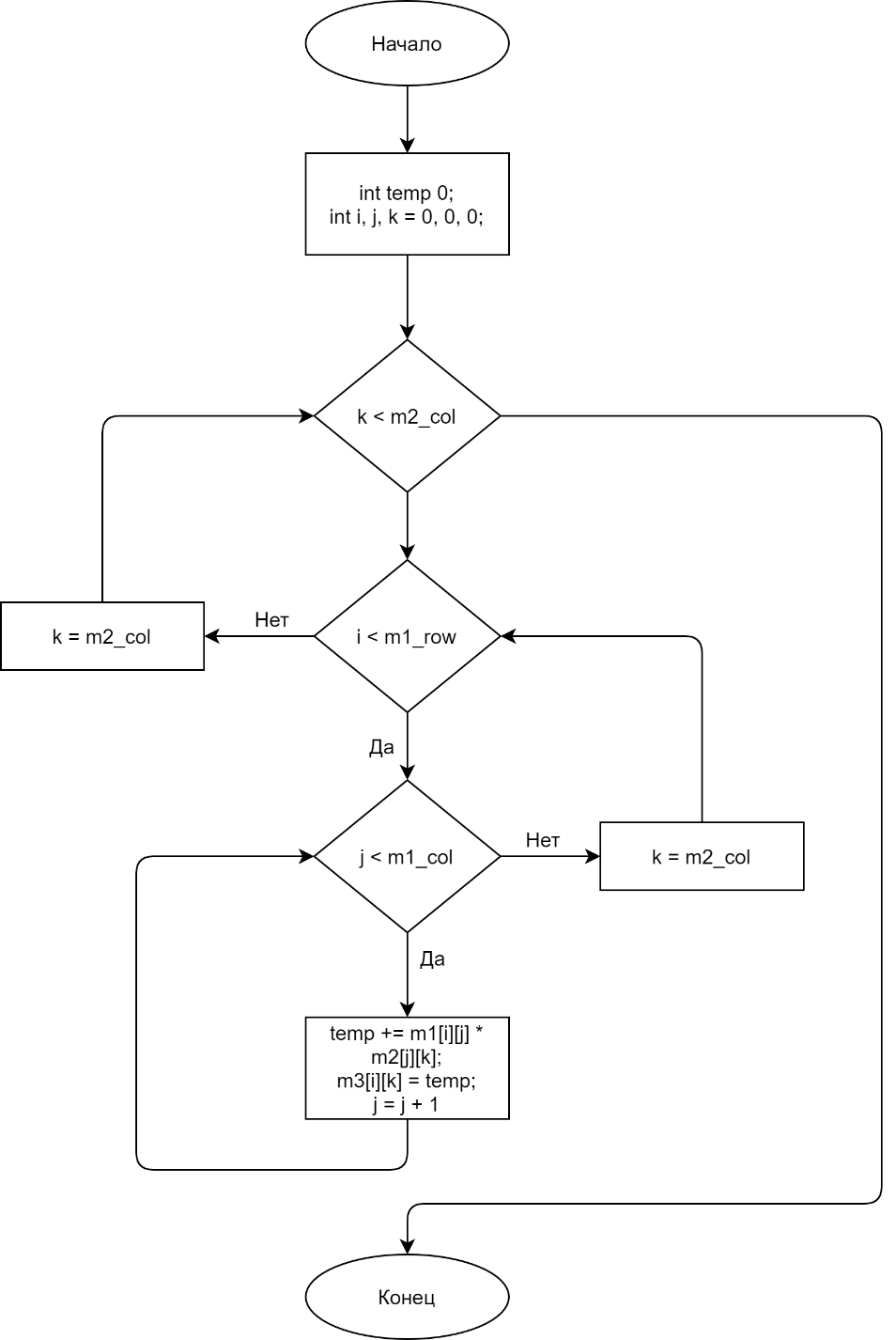
* этап инициализации, на котором каждой подзадаче (i,j) передаются блоки Aij, Bij и обнуляются блоки Cij на всех подзадачах;
* этап вычислений, в рамках которого на каждой итерации l, 0<=l<q, осуществляются следующие операции:

для каждой строки i, 0<=i<q, блок Aij подзадачи (i,j) пересылается на все подзадачи той же строки i решетки; индекс j, определяющий положение подзадачи в строке, вычисляется в соответствии с выражением

полученные в результаты пересылок блоки A'ij, B'ij каждой подзадачи (i,j) перемножаются и прибавляются к блоку Cij

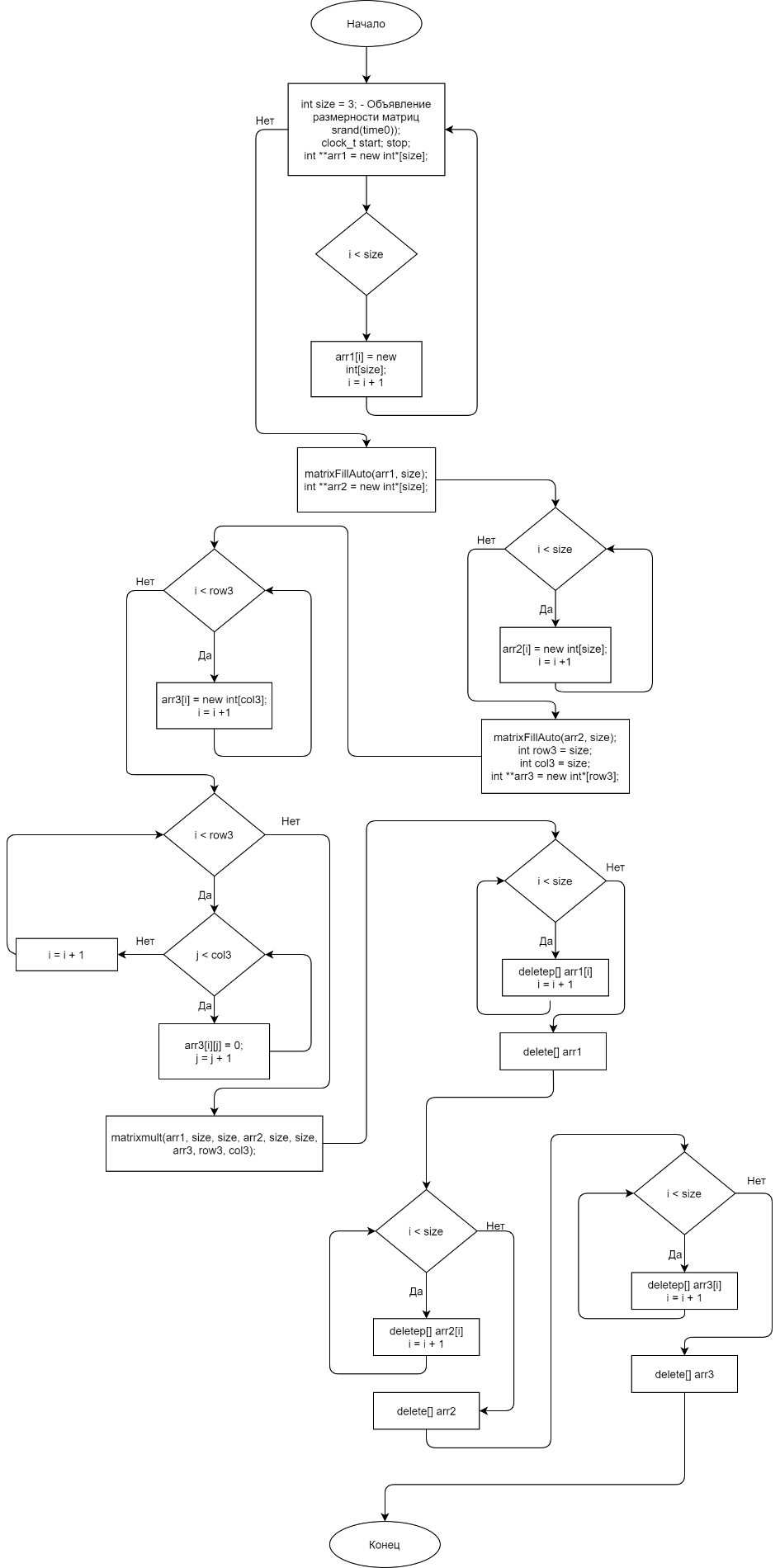
блоки B'ij каждой подзадачи (i,j) пересылаются подзадачам, являющимся соседями сверху в столбцах решетки подзадач (блоки подзадач из первой строки решетки пересылаются подзадачам последней строки решетки).

Ниже продемонстрирован рисунок алгоритма Фокса (рисунок 6).



*Рисунок. 6. Схема алгоритма Фокса перемножения двух матрицы*

Во-пятых, ниже продемонстрирован рисунок алгоритма выполнения главной функции программы main () (рисунок 7).



*Рисунок. 7. Схема главной функции main()*

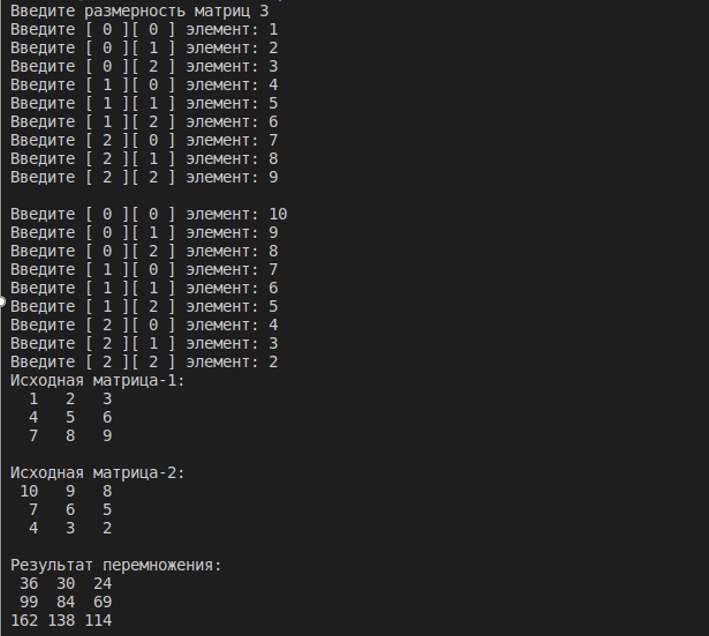
# **Текст исходного кода программы**

|  |
| --- |
| #include<stdlib.h>  #include <ctime>  #include<math.h>  #include"mpi.h"  #include <iostream>  using namespace std;  #define MATRIX\_SIZE 1000  int first\_matrix[MATRIX\_SIZE][MATRIX\_SIZE];  int second\_matrix[MATRIX\_SIZE][MATRIX\_SIZE];  typedef struct {  int proc\_count;  int dim;  int row;  int col;  int rank;  MPI\_Comm grid\_comm;  MPI\_Comm row\_comm;  MPI\_Comm col\_comm;  } GridStructure;  void GridSetup(GridStructure\* grid) {  int dimensions[2];  int wrap\_around[2];  int coordinates[2];  int free\_coords[2];  int world\_rank;  MPI\_Comm\_size(MPI\_COMM\_WORLD, &(grid->proc\_count));  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);  grid->dim = (int)sqrt((double)grid->proc\_count);  dimensions[0] = dimensions[1] = grid->dim;  wrap\_around[0] = 0;  wrap\_around[1] = 1;  MPI\_Cart\_create(MPI\_COMM\_WORLD, 2, dimensions, wrap\_around, 1, &(grid->grid\_comm));  MPI\_Comm\_rank(grid->grid\_comm, &(grid->rank));  MPI\_Cart\_coords(grid->grid\_comm, grid->rank, 2, coordinates);  grid->row = coordinates[0];  grid->col = coordinates[1];  free\_coords[0] = 0;  free\_coords[1] = 1;  MPI\_Cart\_sub(grid->grid\_comm, free\_coords, &(grid->row\_comm));  free\_coords[0] = 1;  free\_coords[1] = 0;  MPI\_Cart\_sub(grid->grid\_comm, free\_coords, &(grid->col\_comm));  }  void MultiplyLocal(int\*\* a, int\*\* b, int\*\* c, int size) {  int temp = 0;  for (int i = 0; i < size; i++)  for (int j = 0; j < size; j++) {  temp = 0;  for (int k = 0; k < size; k++)  temp += (a[i][k] \* b[k][j]);  c[i][j] += temp;  }  }  void UnpackMatrix(int\* buff, int\*\* a, int size) {  int k = 0;  for (int i = 0; i < size; i++)  for (int j = 0; j < size; j++) {  a[i][j] = buff[k];  k++;  }  }  void PackMatrix(int\* buff, int\*\* a, int size) {  int k = 0;  for (int i = 0; i < size; i++)  for (int j = 0; j < size; j++) {  buff[k] = a[i][j];  k++;  }  }  void PrintMatrix(int\*\* matrix, int size) {  for (int i = 0; i < size; i++) {  std::cout << "|";  for (int j = 0; j < size; j++) {  int el = matrix[i][j];  if (el < 10)  std::cout << " ";  std::cout << el;  std::cout << "|";  }  std::cout << std::endl;  }  }  void PrintPackedMatrix(int\* matrix, int size) {  for (int i = 0; i < size; i++) {  std::cout << "|";  for (int j = 0; j < size; j++) {  int el = matrix[i \* size + j];  if (el < 10)  std::cout << " ";  std::cout << el;  std::cout << "|";  }  std::cout << std::endl;  }  }  void GenerateMatrices() {  for (int i = 0; i < MATRIX\_SIZE; i++)  for (int j = 0; j < MATRIX\_SIZE; j++) {  first\_matrix[i][j] = rand() % 100;//(i==j) ? 1 : 0;  second\_matrix[i][j] = rand() % 100;  }  }  void FoxMultiply(int n, GridStructure\* grid, int\*\* a, int\*\* b, int\*\* c) {  int\*\* temp\_a, \* buff, stage, root, submat\_dim, src, dst;  MPI\_Status status;  submat\_dim = n / grid->dim;  temp\_a = new int\* [submat\_dim];  for (int i = 0; i < submat\_dim; ++i)  temp\_a[i] = new int[submat\_dim];  for (int i = 0; i < submat\_dim; i++)  for (int j = 0; j < submat\_dim; j++)  temp\_a[i][j] = 0;  buff = new int[submat\_dim \* submat\_dim];  for (int i = 0; i < submat\_dim \* submat\_dim; i++)  buff[i] = 0;  src = (grid->row + 1) % grid->dim;  dst = (grid->row + grid->dim - 1) % grid->dim;  for (stage = 0; stage < grid->dim; stage++) {  root = (grid->row + stage) % grid->dim;  if (root == grid->col) {  PackMatrix(buff, a, submat\_dim);  MPI\_Bcast(buff, submat\_dim \* submat\_dim, MPI\_INT, root, grid->row\_comm);  UnpackMatrix(buff, a, submat\_dim);  MultiplyLocal(a, b, c, submat\_dim);  }  else {  PackMatrix(buff, temp\_a, submat\_dim);  MPI\_Bcast(buff, submat\_dim \* submat\_dim, MPI\_INT, root, grid->row\_comm);  UnpackMatrix(buff, temp\_a, submat\_dim);  MultiplyLocal(temp\_a, b, c, submat\_dim);  }  PackMatrix(buff, b, submat\_dim);  MPI\_Sendrecv\_replace(buff, submat\_dim \* submat\_dim, MPI\_INT, dst, 0, src, 0, grid->col\_comm, &status);  UnpackMatrix(buff, b, submat\_dim);  }  }  int main(int argc, char\* argv[]) {  int block\_size;  int\*\* local\_a, \*\* local\_b, \*\* local\_c;  clock\_t start, finish;  MPI\_Init(&argc, &argv);  GridStructure grid;  GridSetup(&grid);  GenerateMatrices();  block\_size = MATRIX\_SIZE / grid.dim;  int base\_row = grid.row \* block\_size;  int base\_col = grid.col \* block\_size;  local\_a = new int\* [MATRIX\_SIZE];  local\_b = new int\* [MATRIX\_SIZE];  local\_c = new int\* [MATRIX\_SIZE];  for (int i = 0; i < MATRIX\_SIZE; ++i)  {  local\_a[i] = new int[MATRIX\_SIZE];  local\_b[i] = new int[MATRIX\_SIZE];  local\_c[i] = new int[MATRIX\_SIZE];  }  for (int i = base\_row; i < base\_row + block\_size; i++)  for (int j = base\_col; j < base\_col + block\_size; j++) {  local\_a[i - (base\_row)][j - (base\_col)] = first\_matrix[i][j];  local\_b[i - (base\_row)][j - (base\_col)] = second\_matrix[i][j];  local\_c[i - (base\_row)][j - (base\_col)] = 0;  }  if (grid.rank == 0)  {  std::cout << "Ready..." << std::endl;  }  MPI\_Barrier(grid.grid\_comm);  if (grid.rank == 0)  {  start = clock();  }  FoxMultiply(MATRIX\_SIZE, &grid, local\_a, local\_b, local\_c);  MPI\_Barrier(grid.grid\_comm);  if (grid.rank == 0)  {  finish = clock();  clock\_t result\_time = finish - start;  std::cout << "Time: " << double(result\_time) / 1000. << std::endl;  }  int\* result\_buff = new int[MATRIX\_SIZE \* MATRIX\_SIZE];  int\* local\_buff = new int[block\_size \* block\_size];  PackMatrix(local\_buff, local\_c, block\_size);  MPI\_Gather(local\_buff, block\_size \* block\_size, MPI\_INT, result\_buff, block\_size \* block\_size, MPI\_INT, 0, grid.grid\_comm);  MPI\_Barrier(grid.grid\_comm);  if (grid.rank == 0) {  int\* data = new int[MATRIX\_SIZE \* MATRIX\_SIZE];  int k = 0;  for (int bi = 0; bi < grid.dim; bi++)  for (int bj = 0; bj < grid.dim; bj++)  for (int i = bi \* block\_size; i < bi \* block\_size + block\_size; i++)  for (int j = bj \* block\_size; j < bj \* block\_size + block\_size; j++) {  data[i \* MATRIX\_SIZE + j] = result\_buff[k];  k++;  }  }  MPI\_Finalize();  exit(0);  } |
|  |

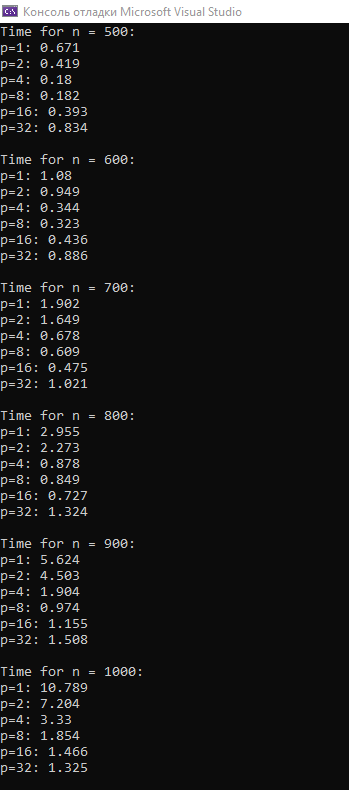
Листинг файла index.cpp. Исходный код программы.

# **Тестирование**

В соответствии с постановкой задачи требовалось произвести ручное тестирования для квадратных матриц размером n=3. На рисунке 7 представлен скриншот результата работы программы, соответствующей исходным требованиям.



*Рисунок.7. Скриншот выполнения работы программы*



*Рисунок.8. Скриншот выполнения работы программы (1)*

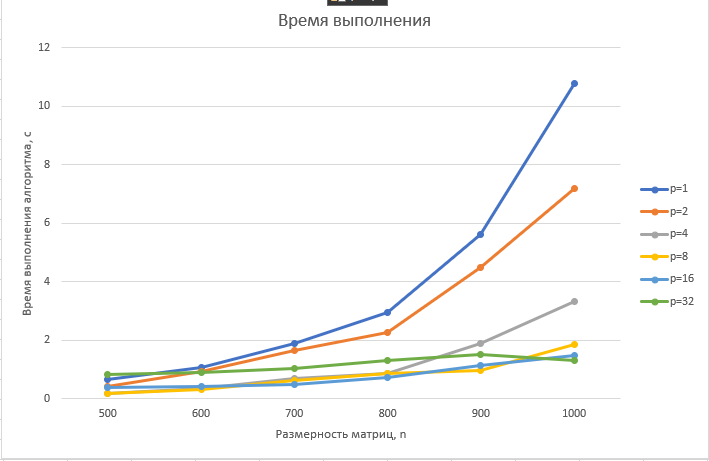
# **Контрольные прогоны**

На основе выполненных тестов были составлены сводные таблицы, на основе которых составлены графики, указанные в основной задаче.

*Таблица 1. Время работы алгоритма Фокса с параллельной и последовательной реализацией*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размерность, n | Время при последовательном перемножении | P = 2 | P=4 | P=8 | P=16 | P=32 |
| 500 | 0,671 | 0,419 | 0,18 | 0,182 | 0,393 | 0,834 |
| 600 | 1,08 | 0,949 | 0,344 | 0,323 | 0,436 | 0,886 |
| 700 | 1,902 | 1,649 | 0,678 | 0,609 | 0,475 | 1,021 |
| 800 | 2,955 | 2,273 | 0,878 | 0,849 | 0,727 | 1,324 |
| 900 | 5,624 | 4,503 | 1,904 | 0,974 | 1,155 | 1,508 |
| 1000 | 10,789 | 7,204 | 3,33 | 1,854 | 1,466 | 1,325 |

На основании этих данных был построен общий график зависимости время выполнения алгоритма от размера матриц (рисунок 8).

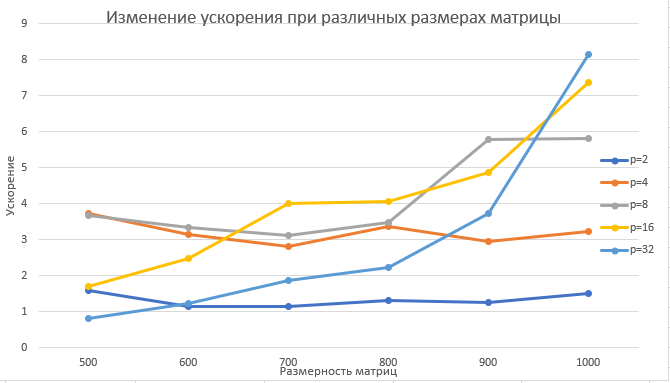


*Рисунок. 8. График зависимости времени выполнения программы от размерности матриц*

*Таблица 2. Изменение ускорения с параллельной и последовательной реализацией алгоритма Фокса в зависимости от размера матриц*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размерность, n | Ускорение Sn, при P = 2 | Ускорение Sn, при P=4 | Ускорение Sn, при P=8 | Ускорение Sn, при P=16 | Ускорение Sn, при P=32 |
| 500 | 1,603 | 3,721 | 3,685 | 1,702 | 0,802 |
| 600 | 1,135 | 3,136 | 3,345 | 2,471 | 1,211 |
| 700 | 1,151 | 2,808 | 3,12 | 4,008 | 1,863 |
| 800 | 1,3 | 3,369 | 3,42 | 4,069 | 2,233 |
| 900 | 1,249 | 2,951 | 5,773 | 4,869 | 3,729 |
| 1000 | 1,492 | 3,233 | 5,811 | 7,359 | 8,142 |

На основании этих данных был построен общий график зависимости ускорения от размера матриц (рисунок 9).

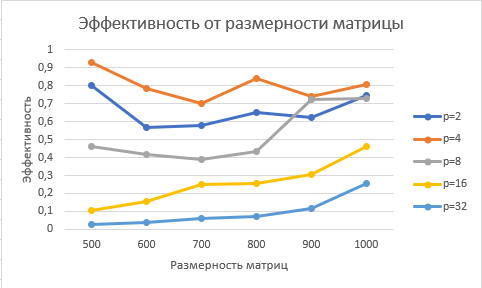


*Рисунок. 9. Зависимость ускорения от размерности матриц*

*Таблица 3. Изменение эффективность с параллельной и последовательной реализацией алгоритма Фокса в зависимости от размера матриц*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размерность, n | Эффективность En, при P = 2 | Эффективность En, при P=4 | Эффективность En, при P=8 | Эффективность En, при P=16 | Эффективность En, при P=32 |
| 500 | 0,8 | 0,931 | 0,41 | 0,102 | 0,025 |
| 600 | 0,569 | 0,784 | 0,415 | 0,151 | 0,038 |
| 700 | 0,576 | 0,701 | 0,33 | 0,26 | 0,058 |
| 800 | 0,65 | 0,841 | 0,435 | 0,255 | 0,069 |
| 900 | 0,624 | 0,738 | 0,726 | 0,303 | 0,116 |
| 1000 | 0,748 | 0,809 | 0,727 | 0,451 | 0,254 |

На основании этих данных был построен общий график зависимости эффективности от размера матриц (рисунок 10).

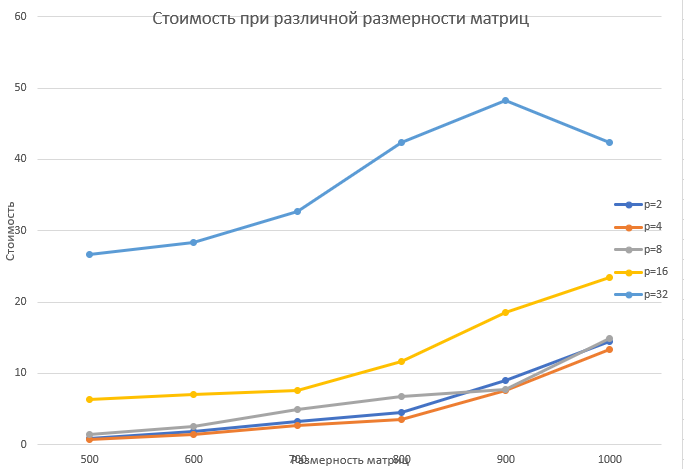


*Рисунок. 10. Зависимость эффективности от размерности матриц*

*Таблица 4. Изменение стоимости с параллельной и последовательной реализацией алгоритма Фокса в зависимости от размера матриц*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Размерность, n | Стоимость Cn, при P = 2 | Стоимость Cn, при P=4 | Стоимость Cn, при P=8 | Стоимость Cn, при P=16 | Стоимость Cn, при P=32 |
| 500 | 0,838 | 0,72 | 1,456 | 6,288 | 26,688 |
| 600 | 1,898 | 1,376 | 2,584 | 6,976 | 28,352 |
| 700 | 3,298 | 2,712 | 4,872 | 7,6 | 32,672 |
| 800 | 4,546 | 3,512 | 6,792 | 11,632 | 42,368 |
| 900 | 9,006 | 7,616 | 7,792 | 18,48 | 48,256 |
| 1000 | 14,408 | 13,32 | 14,832 | 23,456 | 42,4 |

На основании этих данных был построен общий график зависимости стоимости от размера матриц (рисунок 11).



*Рисунок. 11. Зависимость стоимости от размерности матриц*

# **Выводы**

В ходе выполнения данной практической работы реализована программа по перемножению двух матриц при помощи блочного алгоритма Фокса при помощи библиотеки MPI для языка C++. Помимо этого, проведены тестирования программы, составлены сводные таблицы с результатами выполнения, графики.

1. Максимальное ускорение для n = 500 достигается при p = 4
2. Максимальное ускорение для n = 600 достигается при p = 8
3. Максимальное ускорение для n = 700 достигается при p = 16
4. Максимальное ускорение для n = 800 достигается при p = 16
5. Максимальное ускорение для n = 900 достигается при p = 8
6. Максимальное ускорение для n = 1000 достигается при p = 32
7. Для всех размерностей наибольшая эффективность достигается при p = 4;
8. Самый затратный алгоритм – с использованием количества процессов p = 32.
9. Менее затратный алгоритм – с использованием количества процессов p = 4.