# Параллельное и распределённое программирование. Лабораторная работа №1-3. Вилков Е. С. ИВТ-21М. 2025

# Лабораторная работа №1

Задание к лабораторной работе: «Перемножение матриц»

В работе необходимо составить и проверить алгоритм перемножения матриц. Для проверки алгоритма целесообразно использовать небольшие матрицы (до 3х3 включительно), результат перемножения вывести на экран. После успешной проверки задаются большие (1000х1000) матрицы как входные данные, наполнение матриц случайно и необязательно к выводу на экран. На скриншотах выполнения программы должна быть видна разница во времени исполнения в последовательном и в параллельном режимах.

Задание необходимо выполнить при помощи библиотеки MPI (Message-Passing Interface)

За основу был взял код из лабораторной работы №1 прошлого семестра. Так как задача уже хорошо распараллелена, однако это не убавляет проблем с подключением и отладкой программы.

Для начала необходимо подключить все необходимые библиотеки, и включаемые файлы. Они находятся по путям: «C:\Program Files (х86)\Microsoft SDKs\MPI\Include» и «C:\Program Files (x86)\Microsoft SDKs\MPI\Lib\x64», так же, необходимо не забыть подключить статическую библиотеку «**msmpi.lib**» в дальнейших лабораторных работах эти шаги не будут упомянуты. Для удобства выполнения, я в исходный проект добавил функцию, которая проверяет, была ли запущена программа посредством mpiexec или нет, проверяя переменные окружения. Так как если программа запущена через «mpiexec» то будут добавлены некоторые специфические переменные связанные только с MPI, для удобства запуска на любой платформе я проверяю несколько таких переменных.

bool is\_mpiexec\_mode() {

// Check common MPI environment variables

const char\* env\_vars[] = {

"PMI\_SIZE", // Intel MPI, MS-MPI

"OMPI\_COMM\_WORLD\_SIZE", // OpenMPI

"MPI\_LOCALNRANKS", // MPICH

nullptr

};

for (int i = 0; env\_vars[i] != nullptr; i++) {

char\* value = nullptr;

size\_t len = 0;

if (\_dupenv\_s(&value, &len, env\_vars[i]) == 0 && value != nullptr) {

free(value); // Free allocated memory

return true;

}

}

return false;

}

Далее, выполнение задания лабораторной работы:

Долго мучаясь над тем, кто такой MPI и с чем его есть, я начал оставлять себе некоторые полезные комментарии в коде программы, как например вот этот

**mpi\_matrix.cpp**

// главная идея программирования под MPI заключается в том,

// что все процессы запускают один и тот же код

// необходимо согласовать ветвления так,

// чтобы главный процесс (rank == 0) выполнял помимо вычислений

// распределение данных, а также ввод/вывод из консольного окна,

// а остальные процесс только вычисление и получение данных

Сначала я согласовываю между процессами данные, связанные с выделением памяти под контейнеры промежуточных и глобальных результатов

**mpi\_matrix.cpp**

MatrixDimension resultDim;

if (rank == 0)

resultDim = { m1.rowMajorMatrix.size.m, m2.rowMajorMatrix.size.n };

MPI\_Bcast(

&resultDim, // хранилище данных

sizeof(resultDim), // количество элементов

MPI\_BYTE, // тип данных

0, // Кто главный - в данном случае 0

MPI\_COMM\_WORLD); // набор процессов - все

int overallSize = resultDim.m \* resultDim.n;

double\* resultMatrixContainter = new double[overallSize];

double\* localMatrixContainter = new double[overallSize];

Далее, в главном процессе (rank = 0) я подготавливаю аргументы, которые отдельные процессы будут использовать при вычислении своей части задачи.

mpi\_matrix.cpp

// снова, выделяем память и организуем данные только на главном процессе

int\* rows\_count\_data = new int[size \* 6];

int\* offsets = new int[size]; // row\_start \* p + 0;

int\* counts = new int[size]; // (row\_end - row\_start)\* p;

if (rank == 0)

{

double rowsPerProcess = (double)resultDim.m / size;

// -- 1000 / 16 = 62.5; avg rows per thread

int minimalRowsPerProcess = rowsPerProcess;

// floor the 62.5 to 62, so every single thread will calculate at least 62 rows

int remainderRows = resultDim.m - (minimalRowsPerProcess \* size);

// 1000 - (62 \* 16) = 8, how many rows are left

// заполняем данные для процессов, кто сколько будет выполнять строк

for (int i = 0, rowStartCounter = 0; i < size; i++)

{

int rowCount = minimalRowsPerProcess;

if (i < remainderRows)

rowCount += 1;

rows\_count\_data[6 \* i + 0] = rowStartCounter; // указываем для какого процесса с какой строки начинать

rows\_count\_data[6 \* i + 1] = rowStartCounter + rowCount;

// указываем для какого процесса до какой строки считать

rows\_count\_data[6 \* i + 2] = m1.rowMajorMatrix.size.n;

// параметры размеров

rows\_count\_data[6 \* i + 3] = m2.rowMajorMatrix.size.n;

// параметры размеров

rows\_count\_data[6 \* i + 4] = m1.rowMajorMatrix.size.m \* m1.rowMajorMatrix.size.n;

rows\_count\_data[6 \* i + 5] = m2.colMajorMatrix.size.m \* m2.colMajorMatrix.size.n;

offsets[i] = rowStartCounter \* m2.rowMajorMatrix.size.n + 0;

counts[i] = rowCount \* m2.rowMajorMatrix.size.n;

rowStartCounter += rowCount;

}

}

Здесь, переменная «rows\_count\_data» будет согласована между процессами, а переменные «offsets» и «counts» — это хранилища офсетов и количеств элементов различных процессов, подготавливаемые заранее для функции «MPI\_Gatherv». Далее мы синхронизируем данные для каждого процесса.

mpi\_matrix.cpp

// Далее, синхронизируем задачи по процессам

MPI\_Bcast(

rows\_count\_data, // хранилище данных

size \* 6, // количество элементов

MPI\_INT, // тип данных

0, // Кто главный - в данном случае 0

MPI\_COMM\_WORLD); // набор процессов - все

int row\_start = rows\_count\_data[6 \* rank + 0];

int row\_end = rows\_count\_data[6 \* rank + 1];

int m = rows\_count\_data[6 \* rank + 2];

int p = rows\_count\_data[6 \* rank + 3];

int m1rsize = rows\_count\_data[6 \* rank + 4];

int m2csize = rows\_count\_data[6 \* rank + 5];

double\* m1rContainter;

double\* m2cContainter;

if (rank == 0)

{

m1rContainter = m1.rowMajorMatrix.container;

m2cContainter = m2.colMajorMatrix.container;

}

else

{

m1rContainter = new double[m1rsize];

m2cContainter = new double[m2csize];

}

// т.к. перемножение матриц так или иначе требует значение "всех элементов" матрицы

// даже при расчёте одной строки, надо передать данные матрицы первого процесса всем

MPI\_Bcast(

m1rContainter, // хранилище данных

m1rsize, // количество элементов

MPI\_DOUBLE, // тип данных

0, // Кто главный - в данном случае 0

MPI\_COMM\_WORLD); // набор процессов - все

MPI\_Bcast(

m2cContainter, // хранилище данных

m2csize, // количество элементов

MPI\_DOUBLE, // тип данных

0, // Кто главный - в данном случае 0

MPI\_COMM\_WORLD); // набор процессов - все

Так как данные изначально хранятся только у «главного» процесса, то указатели «m1rContainter» и «m2cContainter» для него инициализируются отдельно, иначе все массивы будут пустые и синхронизировать будет нечего.

Дело остаётся за малым, надо запустить код расчёта перемножения матриц на всех процессах.

mpi\_matrix.cpp

// теперь можно пойти и выполнить части данных

calculate\_part\_mpi(

m1rContainter,

m2cContainter,

localMatrixContainter,

row\_start,

row\_end,

m,

p

);

// Умножение матриц в нескольких потоках с использованием стандартных потоков (параллельно)

// Хотя в данном случае "потоками" выступают отдельные процессы MPI, код от "многопоточной версии" не отличается

void calculate\_part\_mpi(double\* m1RowArr, double\* m2ColArr, double\* rRowArr, int n\_start, int n\_end, int m, int p)

{

for (int i = n\_start; i < n\_end; i++)

{

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

{

double sum = 0.0;

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

{

sum += m1RowArr[i \* m + k] \* m2ColArr[j \* m + k];

}

rRowArr[i \* p + j] = sum;

}

}

}

После выполнения вычислений, отдельные части будут записаны в локальном массиве, который довольно удобно можно копировать в результирующий массив. Я думаю оставленный комментарий в коде объясняет идею не плохо:

mpi\_matrix.cpp

// вычисленные данные хранятся в локальном контейнере в следующем формате

// rank = 0

// [0][1][2][3][4]...

// [X][X][X][X][X]...

//

// rank = 1

// [X][X][X][X][X]...

// [5][6][7][8][9]...

//

// соответственно, необходимо синхронизировать эти данные, и отправить в главный процесс

// так как данные в памяти разложены по строкам друг-за-другом, а разбиения "каждая-другая-строка" нет

// так как алгоритм пытается оптимизировать по кэшу, то реально, вычисленные данные всегда находятся

// в одном куске непрерывной памяти, и его можно соответственно передать без циклов

// начало памяти будет первый элемент соответствующей номера строки (row\_start \* p (размер строки) + 0)

// а конец памяти соответственно будет + кол-во элементов

// а количество элементов это (row\_end - row\_start) \* p; пр. (5 - 2) = 3; 3 \* 64 = 192.

Далее, собираем данные в результирующий массив и выходим из функции.

mpi\_matrix.cpp

// Сбор результатов на процессе 0

if (true)

{

MPI\_Gatherv(

&localMatrixContainter[row\_start \* p + 0], (row\_end - row\_start) \* p, MPI\_DOUBLE, // Отправляемые данные

resultMatrixContainter, counts, offsets, MPI\_DOUBLE, // Буфер для приема

0, MPI\_COMM\_WORLD);

if(rank == 0)

std::cout << "Process [" << rank << "/" << size << "]: Results: " << resultMatrixContainter[0] << std::endl;

}

Здесь осталось выражение «if (true)» которое я изначально поставил для того чтобы попытаться вызвать MPI\_Gather только на тех процессах, которые имели хоть какую-либо вычислительную нагрузку (row\_end - row\_start > 0), однако, как оказалось, это приводит к приостановке выполнения программы, так как для продолжения, все процессы из «MPI\_COMM\_WORLD» должны выполнить эту строку, я решил что создавать отдельный коммуникатор будет сложно, потому просто убрал условие и оставил код как есть.

Однако, для чистоты кода, лучше бы этот условный оператор убрать в принципе. Уберу в репозитории.

Далее, про обёртку основного кода. Код который не должен выполняться на дочерних процессах MPI системы я обернул в типовое условие «if (!is\_mpiexec\_mode() || mpi\_rank == 0)» которое проверяет, либо MPI не запущен вовсе (то есть процесс один – основной, либо, если MPI всё таки есть, то код должен выполняться только на основном процессе.

Код связанный с запуском функции вычисления при помощи MPI выглядит следующим образом:

main.cpp

if (is\_mpiexec\_mode())

{

if (mpi\_rank == 0)

cout << "Running MPI multi-process..." << endl;

// All processes wait here until everyone arrives

MPI\_Barrier(MPI\_COMM\_WORLD);

startMPI = high\_resolution\_clock::now();

m6 = multiply\_matrix\_mpi(c1, c2);

endMPI = high\_resolution\_clock::now();

}

И конечно же, на выходе, необходимо завершить работу MPI правильно.

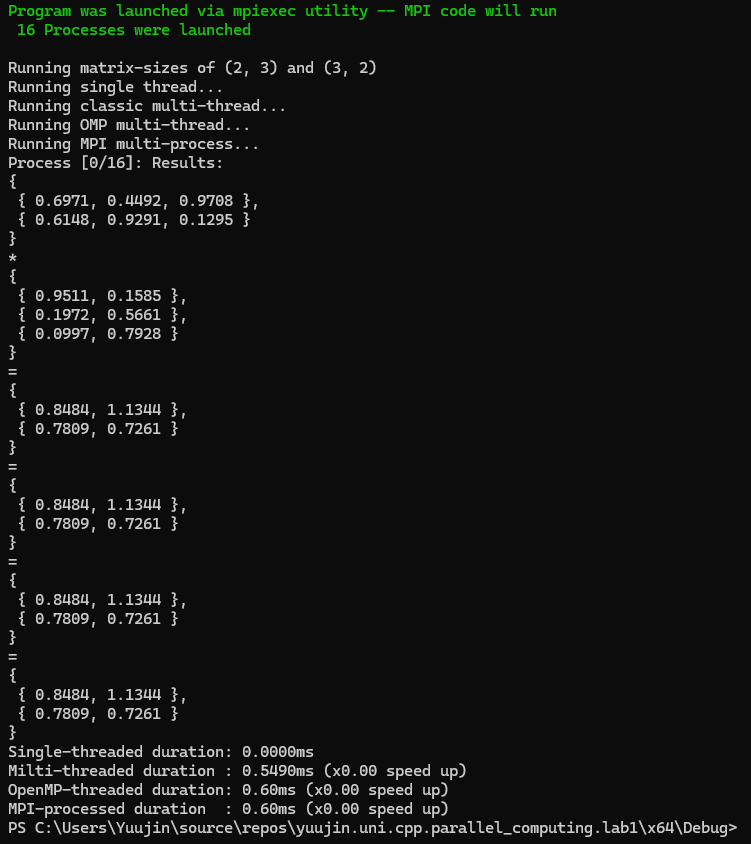
if(is\_mpiexec\_mode())

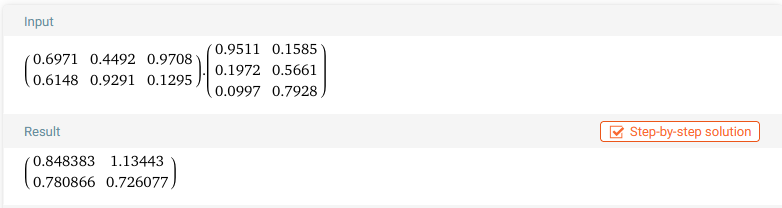
MPI\_Finalize();

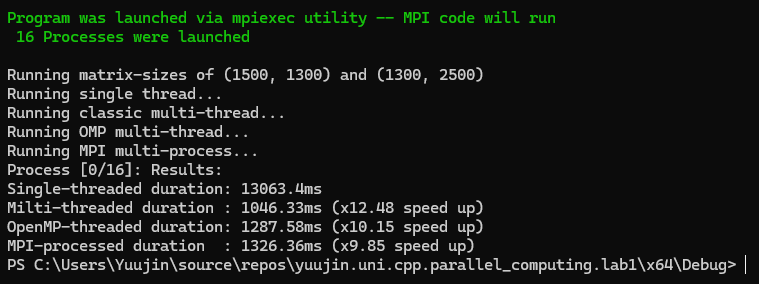
Далее, запускаем программу на малых матрицах, чтобы проверить целостность данных (Рисунок 1) и потом на больших размерах , чтобы оценить прирост производительности (Рисунок 3). Проверка результатов в WolframAlpha показала, что да, результат верный (Рисунок 2).

**Выводы:**

* MPI – не совсем библиотека для «многопоточности», она скорее для распределённых вычислений, и запускаем мы их на наших ПК не разворачивая HPC кластеры, а при помощи костыля по сути, от Майкрософт. Процессы – это не совсем «потоки», да, конечно, у процесса помимо хедера, модулей и памяти обязательно должен быть хотя бы один поток, исполняющий код, однако в данных условиях вполне ожидаемо, что прирост производительности будет несколько меньше, чем при многопоточном подходе, поскольку существует большое количество «оверхеда» вычислений, например:
  + Память между процессами надо копировать, это занимает определённое время, особенно, когда данных много. В отличии от многопоточного подхода, где все потоки могут сразу обращаться к одним и тем же адресам памяти.
  + Коммуникатор не может просто так взять и скопировать память, ядро ОС Windows так не работает, т.к. множество указателей неизвестны заранее, необходимо открывать Handle к процессу, и уже только после этого копировать память
  + Память приходится дублировать, и аллоцировать к каждому процессу отдельно, что тоже занимает время у ядра, особенно в случае с большим объёмом, ведь память надо выделить одним непрерывным куском.
* Дополнительно – выделение памяти, как упоминалось, память приходится дублировать, как под результаты, так и под входные данные, в итоге, в худшем случае, исполнение такой программы будет занимать ровно в столько раз больше памяти, сколько было запущено процессов, это не есть хорошо.
* Отладка – так как это отдельные процессы, их не получится отлаживать и профилировать одним инструментом, приходится использовать вывод в консоль – что не есть хорошо.

  
Рисунок 1. Запуск программы на малых размерах матрицы.

  
Рисунок 2. Проверка результатов в Wolfram Alpha

  
Рисунок 3. Проверка прироста производительности

# Лабораторная работа №2

Задание к лабораторной работе: 2. «Решение уравнений методом Гаусса»

Принцип лабораторной работы аналогичен первой: составление и проверка алгоритма решения уравнений методом Гаусса на примерах с малым количеством уравнений (2-3), сравнение времени выполнения программы в разных режимах на примере с большим количеством уравнений (от 500).

Выполнение задания лабораторной работы

В данной лабораторной работе интересно. Учитывая что в MPI есть довольно удобные инструменты для синхронизации памяти между процессами, а если точнее, в MPI 3.0 и выше можно использовать «shared» память, которую могут использовать одновременно все процессы одного кластера. У нас это так, потому как мы запускаем программу через «mpiexec»

Это распараллеливать по сути алгоритм, который был написан под один поток, буквально одной строкой кода, в нашем случае это так, потому как алгоритм хорошо распараллеливается на единую память, если операции записи не пересекаются, а они не пересекаются, так как каждый процесс может обрабатывать разную строку.

Как упоминалось ранее, идея выполнения задания данной лабораторной работы состоит в использовании общей памяти

Matrix solve\_gauss\_mpi(const Matrix& matrix)

{

int rank;

int size;

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

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

MatrixDimension matrixSize;

if (rank == 0)

matrixSize = matrix.size;

MPI\_Bcast(&matrixSize, sizeof(MatrixDimension), MPI\_BYTE, 0, MPI\_COMM\_WORLD);

MPI\_Win win;

double\* shared\_data;

MPI\_Win\_allocate\_shared(

(rank == 0) ? sizeof(double) \* matrixSize.m \* matrixSize.n : 0,

sizeof(double),

MPI\_INFO\_NULL,

MPI\_COMM\_WORLD,

&shared\_data, &win

);

memcpy(shared\_data, matrix.container, sizeof(double) \* matrixSize.m \* matrixSize.n);

MPI\_Win\_fence(0, win);

// Не забыть тыкнуться в общую память

double\* localContainer = nullptr;

MPI\_Aint local\_size;

int disp\_unit;

MPI\_Win\_shared\_query(win, 0, &local\_size, &disp\_unit, &localContainer);

// Делаем матрицу треугольной

make\_upper\_triangle\_matrix\_mpi(localContainer, matrixSize, rank, size, win);

// Решаем гаусса

solve\_mpi(localContainer, matrixSize, rank, size, win);

if (rank == 0)

return { localContainer , matrixSize };

else

return {};

}

Мы выделяем память под матрицу на основном процессе (rank = 0) и после на всех процессах выбираем себе локальный указатель и «handle» окна памяти, его и передаём в функции вычисления.

В самих функциях вычисления изменения от оригинального алгоритма для одного потока не большие: мы синхронизируем память в окне, и ставим барьер на внешний цикл, чтобы процессы не могли уйти обрабатывать следующий проход, используя не готовые данные.

void make\_upper\_triangle\_matrix\_mpi(double\* matrixContainer, MatrixDimension matrixSize, int rank, int size, MPI\_Win memoryWindow)

{

// for every row

for (int row = 0; row < matrixSize.m - 1; row++)

{

// Все процессы должны циклично вычесть подготовленную строку из оставшихся

double item = matrixContainer[row \* matrixSize.n + row];

for (int changeRow = row + 1; changeRow < matrixSize.m; changeRow++)

{

// Пропускаем строку, если она имеет не наш номер

if (changeRow % size != rank)

continue;

double item\_to\_null = matrixContainer[changeRow \* matrixSize.n + row];

double coeffitient = item\_to\_null / item;

for (int col = row; col < matrixSize.n; col++)

{

matrixContainer[changeRow \* matrixSize.n + col] -= coeffitient \* matrixContainer[row \* matrixSize.n + col];

}

}

// синхронизируем память

MPI\_Win\_fence(0, memoryWindow);

// чтобы не дай бог какой-либо из процессов пошёл обрабатывать следующую строку и использовал не готовый item

MPI\_Barrier(MPI\_COMM\_WORLD);

}

}

Функция решения системы из треугольной матрицы мало чем отличается от функции составления, изменён порядок обработки матриц на обратный и добавлена нормализация текущей строки в главном процессе:

void solve\_mpi(double\* matrixContainer, MatrixDimension matrixSize, int rank, int size, MPI\_Win memoryWindow)

{

for (int row = matrixSize.m - 1; row >= 0; row--)

{

// Все процессы должны циклично вычесть подготовленную строку из оставшихся

double item = matrixContainer[row \* matrixSize.n + row];

for (int changeRow = row - 1; changeRow >= 0; changeRow--)

{

// Пропускаем строку, если она имеет не наш номер

if (changeRow % size != rank)

continue;

double item\_to\_null = matrixContainer[changeRow \* matrixSize.n + row];

double coeffitient = item\_to\_null / item;

for (int col = row; col < matrixSize.n; col++)

{

matrixContainer[changeRow \* matrixSize.n + col] -= coeffitient \* matrixContainer[row \* matrixSize.n + col];

}

}

// нормализуем текущую строку, если мы главный процесс

if (rank == 0)

{

for (int col = 0; col < matrixSize.n; col++)

{

matrixContainer[row \* matrixSize.n + col] /= item;

}

}

// синхронизируем память

MPI\_Win\_fence(0, memoryWindow);

// чтобы не дай бог какой-либо из процессов пошёл обрабатывать следующую строку и использовал не готовый item

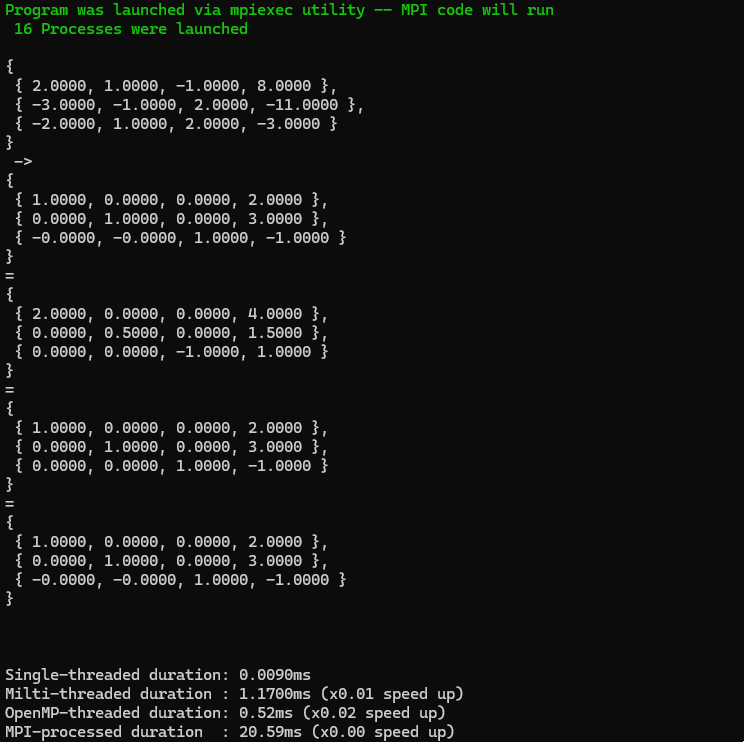
MPI\_Barrier(MPI\_COMM\_WORLD);

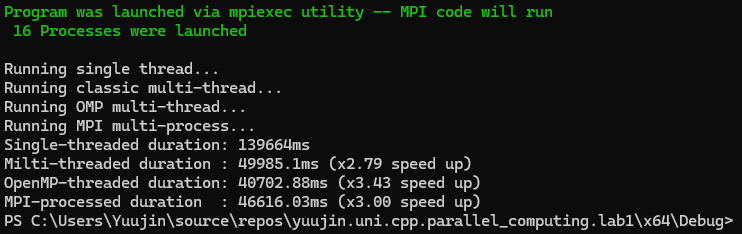
}

}

После проверки запуска программы на матрице малого размера (Рисунок 4), видно что система была решена верно. То есть алгоритм работает, можно дальше посмотреть на прирост производительности на матрице большого размера (Рисунок 5, 5000х5000 элементов).

Как видно, прирост составил даже больше, чем с чисто многопоточным решением, хотя откровенно говоря, в многопоточном решении используется немного другой подход к вычислению, но даже тут можно заметить, по сравнению с первой лабораторной работой, насколько важно иметь возможность обращения к общей памяти, и насколько сильно копирования памяти туда-сюда влияют на производительность.

  
Рисунок 4. Запуск программы на матрице малого размера

  
Рисунок 5. Проверка прироста быстродействия

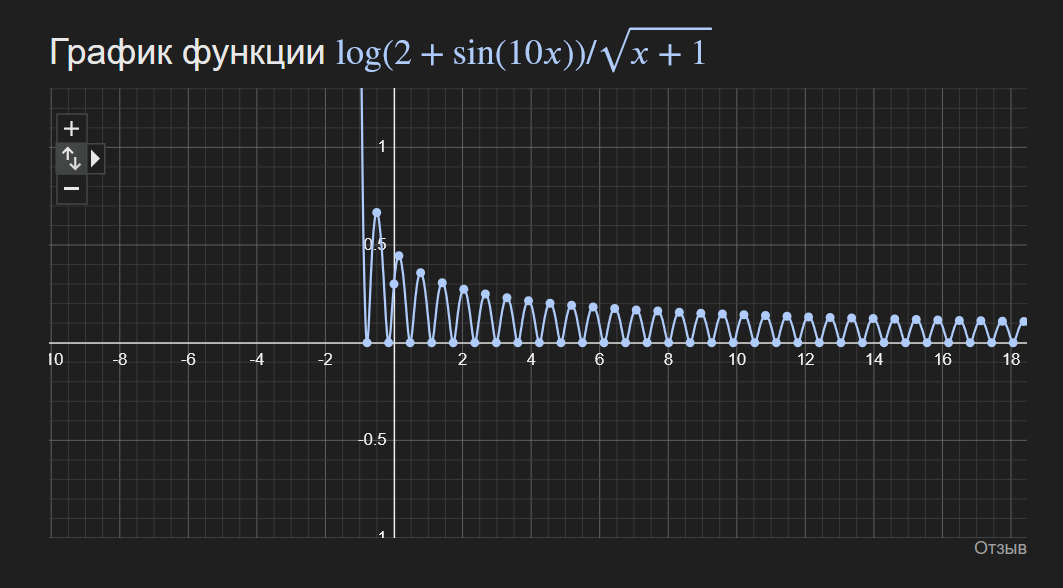
# Лабораторная работа №3.

Задание к лабораторной работе: 3. «Решение интеграла»

В работе необходимо вычислить случайный интеграл средней сложности на диапазоне от (например) 1 до 1000 с шагом 1 (от 1 до 2, от 2 до 3 и т.д.) для проверки алгоритма вычисления. В дальнейшем шаг вычислений уменьшить в 100-1000 раз и запустить в последовательном и параллельном режимах работы аналогично предыдущим лабораторным работам.

Для вычисления интеграла была выбрана следующая функция:

Выглядит она следующим образом:

****  
Рисунок 6. График выбранной функции.

Как и в предыдущем семестре, критерий того какая функция должна быть выбрана звучи так: «даст интеграл «средней сложности»», потому, был выбран такой интеграл, от всего сердца. И сердце подсказывает что интеграл средней сложности.

Выполнение лабораторной работы:

Был выбран простейший метод приближения конечного интеграла: Суммы Римана. Так как геометрический смысл интеграла – знаковая площадь под графиком, то тогда интеграл можно представить как предел сумм «прямоугольников», высота которых между значениями функции в точках расположения боковых сторон такого прямоугольника, умноженных на ширину данных прямоугольников. И при ширине таких прямоугольников, стремящихся к нулю, такая сумма стремится к значению интеграла функции.

Можно было бы выбрать и другой метод, например аппроксимация трапецией, но зачем напрягаться, когда можно не напрягаться? (в задании лабораторной работы не уточнено какой метод необходимо использовать).

В случае решения с использованием библиотеки MPI применяется всего лишь одна функция библиотеки MPI: «MPI\_Reduce»

double calc\_integral\_mpi(double start, double end, double step)

{

int rank;

int size;

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

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

double area = 0;

int n = (end - start) / step;

double rem = (end - start) - (n \* step);

double processStep = (end - start) / size;

double processStart = start + rank \* processStep;

double processEnd = start + (rank + 1) \* processStep;

for (double pos = processStart; pos < processEnd; pos += step)

{

area += (function(pos) + function(pos + step)) \* 0.5 \* step;

}

double global\_area = 0;

// Сбор и суммирование результатов на процессе 0 (MPI\_Reduce)

MPI\_Reduce(&area, &global\_area, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (rank == 0)

return global\_area;

return area;

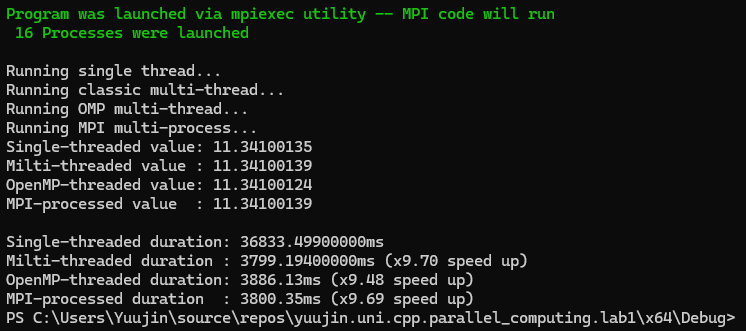
}

Она позволяет очень удобно сложить несколько данных в одну переменную с разных процессов, и потому, ожидаем что производительность такого алгоритма не сильно будет отличаться от многопоточного выполнения, потому как в данном случае не присутствует главного «бутылочного» горлышка при программировании на MPI – переноса больших объёмов данных.

И действительно, как видно на Рисунке 7, время выполнения практически равно времени выполнения программы на многопоточном исполнении, что подтверждает утверждение всех этих работ: работы над памятью дорогие (в плане затрат времени).

Как всегда, полный код лабораторных работ можно найти в репозитории:

<https://github.com/ZecosMAX/yuujin.uni.cpp.parallel_computing>

  
Рисунок 7. Проверка прироста быстродействия.