

**Филиал федерального государственного бюджетного  
образовательного учреждения высшего образования  
«Национальный исследовательский университет «МЭИ»  
в г. Смоленске**

Кафедра вычислительной техники

Направление: 09.04.01. «Информатика и вычислительная техника»  
Профиль: «Программное обеспечение средств вычислительной техники и  
автоматизированных систем»

Расчетно-графическая работа  
**«Вычисление определенного интеграла с использованием  
технологий OpenMP и MPI»**  
по курсу:  
**«Вычислительные системы»**

Студент: Старостенков А.А.

Группа: ВМ-22(маг)

Вариант: 19

Преподаватель: Федулов А.С.

Смоленск, 2023

## 1 Рабочее задание

1. Написать, отладить, скомпилировать и запустить на гибридном вычислительном кластере СФМЭИ программу **последовательного** вычисления определенного интеграла. Предусмотреть замер времени выполнения программы с использованием функции **omp\_get\_wtime ()**. *Необходимо предусмотреть контроль правильности вычисления определенного интеграла.* Индивидуальные задания в соответствии с номером по журналу взять из таблицы 1.

2. Получить зависимость времени выполнения **последовательной** программы от числа отрезков разбиения интервала интегрирования  $n$ . Максимальное значение  $n$  выбрать таким, чтобы время выполнения последовательной версии достигало величины порядка 1- 5 секунд.

3. Написать, отладить, скомпилировать и запустить на гибридном вычислительном кластере СФМЭИ (на узле управления (УУ)) программу **параллельного** вычисления определенного интеграла с помощью OpenMP. Предусмотреть замер времени выполнения **параллельной** программы. Число нитей для реализации параллельной программы выбрать по умолчанию (максимально возможным для узла управления). При этом необходимо определить это число с помощью функции **omp\_get\_num\_threads()** или переменной окружения **OMP\_NUM\_THREADS** и вывести на консоль (или в файл) в качестве одного из результатов работы программы.

4. Получить зависимость времени выполнения параллельной программы от числа отрезков разбиения интервала интегрирования  $n$ .

5. Получить зависимость ускорения параллельного алгоритма  $S_{\text{пар}}=(T_{\text{посл}}/T_{\text{пар}})$ , где  $T_{\text{посл}}$  - время выполнения последовательного алгоритма,  $T_{\text{пар}}$  — время выполнения параллельного алгоритма, от числа отрезков разбиения интервала интегрирования  $n$ .

6. При максимальном  $n$  (из пункта 2 рабочего задания) получить зависимость времени выполнения параллельной программы от числа нитей, использующихся в параллельной секции. Число нитей изменять с помощью

функции `omp_set_num_threads()` или переменной окружения `OMP_NUM_THREADS`.

7. Запустить параллельную программу на вычислительном узле 1 (BY1) и вычислительном узле 2 (BY2).

8. Сравнить время вычисления параллельной программы при максимальном  $n$  (из пункта 2) и **максимальном** числе нитей на узлах УУ, BY1, BY2.

9. Сравнить время вычисления параллельной программы при максимальном  $n$  и **одинаковом (32)** числе нитей на узлах УУ, BY1, BY2.

10. Написать, отладить, скомпилировать и запустить на гибридном вычислительном кластере СФМЭИ (на УУ) MPI-программу вычисления определенного интеграла. Предусмотреть замер времени выполнения программы, контроль правильности вычисления интеграла. Индивидуальные задания в соответствии с номером по журналу взять из таблицы 1.

11. Повторить пункты 4-9 для MPI-программы. Естественно, число процессов для MPI задается средой выполнения.

12. Сравнить полученную в пункте 5 зависимость для OpenMP и MPI (число нитей (процессов) в OpenMP и MPI должно быть одинаковым, запуск производится на одном и том же узле).

13. При максимальном  $n$  продлить полученную в пункте 6 для MPI-программы зависимость времени выполнения параллельной программы от числа процессов, используя запуск программы на двух вычислительных узлах (число процессов – до 80), на трех узлах (число процессов – до 112).

14. Сравнить полученное в MPI максимальное ускорение с максимальным ускорением, полученным с помощью OpenMP.

15. Все полученные зависимости оформить в виде графиков. При необходимости использовать табличную форму представления.

19.  $\int_1^2 \frac{5x}{(x+1)^2} dx$

## Ход работы

1. Написать, отладить, скомпилировать и запустить на гибридном вычислительном кластере СФМЭИ программу **последовательного** вычисления определенного интеграла. Предусмотреть замер времени выполнения программы с использованием функции **omp\_get\_wtime ()**. *Необходимо предусмотреть контроль правильности вычисления определенного интеграла.* Индивидуальные задания в соответствии с номером по журналу взять из таблицы 1.

### Программа:

```
#include <stdio.h>
#include <omp.h>

double function(double x)
{ return 5 * x / ((x + 1) * (x + 1)); }

const double MIN_VALUE = 1;
const double MAX_VALUE = 2;
int main()
{
    printf("Counts;\t\t\tResult;\t\t\tTime\n");
    long iteration_count = 30000000;
    double step = (MAX_VALUE - MIN_VALUE) / iteration_count;
    double start_time = omp_get_wtime();
    double integral = 0;
    for (int i = 0; i < iteration_count - 1; ++i)
    {
        double left_point = MIN_VALUE + step * i;
        double right_point = MIN_VALUE + step * (i + 1);
        double result = function((right_point + left_point) / 2);
        integral += (right_point - left_point) * result;
    }
    printf("%20ld;\t%.20f;\t%.20f\n",
        iteration_count,
        integral,
        omp_get_wtime() - start_time);

    return 0;
}
```

## Выполнение программы:

```
• [starostenkov_aa@mng1 calc-task]$ module load GCC
• [starostenkov_aa@mng1 calc-task]$ gcc -fopenmp 1.c -o 1
• [starostenkov_aa@mng1 calc-task]$ ./1
Counts;           Result;           Time
          300000000; 1.19399220350379597910; 2.75725505081936717033
○ [starostenkov_aa@mng1 calc-task]$
```

Рисунок 1 – Последовательное вычисление интеграла.

Проверим правильность вычисления в математическом пакете Octave

8.2.0.

```
f = @(x) 5 * x / ((x + 1) * (x + 1));
a = 1;
b = 2;

integral_value = quad(f, a, b);
disp(['Функция: ', func2str(f)]);
disp(['Значение интеграла: ', num2str(integral_value)]);

>> rgr

Функция: @(x) 5 * x / ((x + 1) * (x + 1))
Значение интеграла: 1.194
>>
```

Рисунок 2 – Вычисление интеграла.

Полученные значения совпадают, следовательно, выбран правильный числовой ряд. Можно приступить к выполнению рабочего задания.

2. Получить зависимость времени выполнения **последовательной** программы от числа отрезков разбиения интервала интегрирования  $n$ . Максимальное значение  $n$  выбрать таким, чтобы время выполнения последовательной версии достигало величины порядка 1- 5 секунд.

## Программа:

```
#include <stdio.h>
#include <omp.h>

double function(double x) {
    return 5.0 * x / ((x + 1.0) * (x + 1.0));
}

const double MIN_VALUE = 1;
const double MAX_VALUE = 2;
const double MAX_ITERATIONS = 300000000;
const double MIN_ITERATIONS = 3;
const double ITERATIONS_MULTIPLIER = 10;

int main() {
    printf("Counts;\t\t\tResult;\t\t\tTime\n");

    for (long iteration_count = MIN_ITERATIONS;
        iteration_count <= MAX_ITERATIONS;
        iteration_count *= ITERATIONS_MULTIPLIER) {

        double step = (MAX_VALUE - MIN_VALUE) / iteration_count;
        double start_time = omp_get_wtime();
        double integral = 0;

        for (int i = 0; i < iteration_count-1; ++i) {
            double left_point = MIN_VALUE + step * i;
            double right_point = MIN_VALUE + step * (i+1);
            double result = function((right_point + left_point) / 2);
            integral += (right_point - left_point) * result;
        }

        printf("%20ld;\t%.20f;\t%.20f\n",
            iteration_count,
            integral,
            omp_get_wtime() - start_time
        );
    }

    return 0;
}
```

## Выполнение программы:

```
Counts;          Result;          Time
      3;          0.81420118343195246879; 0.00000032992102205753
     30;          1.15686086348399697066; 0.00000064703635871410
    300;          1.19028756043113781438; 0.00000562705099582672
   3000;          1.19362182740639166667; 0.00005445210263133049
  30000;          1.19395517007615681315; 0.00054387305863201618
 300000;          1.19398850350284346788; 0.00549062713980674744
3000000;          1.19399183683712739601; 0.04788037599064409733
30000000;         1.19399217017026515464; 0.36601664405316114426
300000000;        1.19399220350379597910; 2.80837081209756433964
[starostenkov_aa@mng1 calc-task]$
```

Рисунок 4 – Последовательное выполнение с интервалами интегрирования.

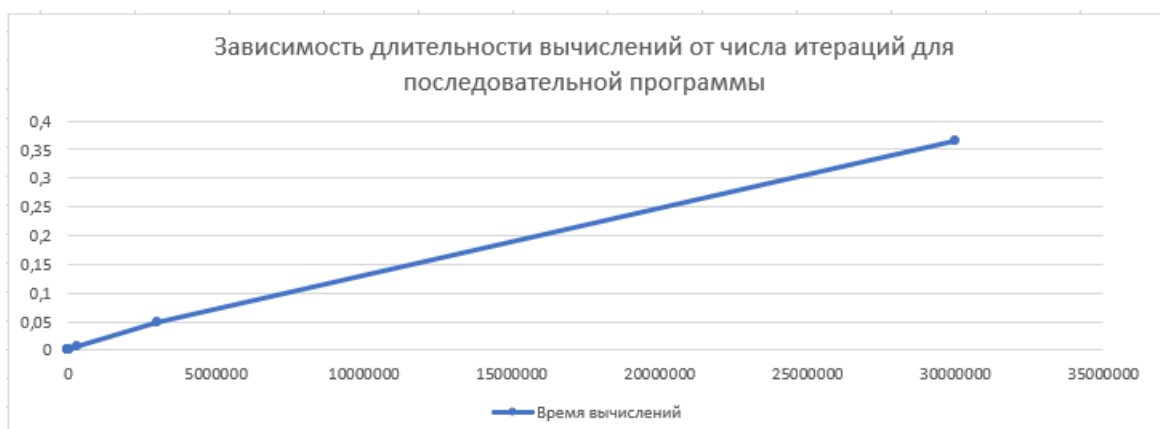


Рисунок 5 – График зависимости длительности вычислений от числа итераций

По оси x: число итераций, по Y время выполнения

3. Написать, отладить, скомпилировать и запустить на гибридном вычислительном кластере СФМЭИ (на узле управления (УУ)) программу **параллельного** вычисления определенного интеграла с помощью OpenMP. Предусмотреть замер времени выполнения **параллельной** программы. Число нитей для реализации параллельной программы выбрать по умолчанию (максимально возможным для узла управления). При этом необходимо определить это число с помощью функции `omp_get_num_threads()` или переменной окружения `OMP_NUM_THREADS` и вывести на консоль (или в файл) в качестве одного из результатов работы программы.

## Программа:

```
#include <stdio.h>
#include <omp.h>

double function(double x)
{
    return 5 * x / ((x + 1) * (x + 1));
}

const double MIN_VALUE = 1;
const double MAX_VALUE = 2;

int main()
{
    omp_set_num_threads(32);
    printf("Counts;\t\t\tThreads;\t\t\tResult;\t\t\t\tTime\n");
    long iteration_count = 200000000;
    double step = (MAX_VALUE - MIN_VALUE) / iteration_count;
    double start_time = omp_get_wtime();
    double integral = 0;

#pragma omp parallel for reduction(+ : integral)
    for (int i = 0; i < iteration_count - 1; ++i)
    {
        double left_point = MIN_VALUE + step * i;
        double right_point = MIN_VALUE + step * (i + 1);
        double result = function((right_point + left_point) / 2);
        integral += (right_point - left_point) * result;
    }
    printf("%20ld;\t\t%20ld;\t\t%.20f;\t\t%.20f\n",
        iteration_count,
        omp_get_max_threads(),
        integral,
        omp_get_wtime() - start_time);

    return 0;
}
```

## Результат выполнения программы:

```
● [starostenkov_aa@mng1 calc-task]$ gcc -fopenmp 3.c -o 3
● [starostenkov_aa@mng1 calc-task]$ ./3
Counts;          Threads;          Result;          Time
      200000000;           32;  1.19399220165192354592; 0.16077431198209524155
○ [starostenkov_aa@mng1 calc-task]$
```

Рисунок 6 – Параллельное вычисление интеграла.

Параллельная программа справилась быстрее.



4. Получить зависимость времени выполнения параллельной программы от числа отрезков разбиения интервала интегрирования  $n$ .

### Программа:

```
#include <stdio.h>
#include <omp.h>

double function(double x)
{
    return 5.0 * x / ((x + 1.0) * (x + 1.0));
}

const double MIN_VALUE = 1;
const double MAX_VALUE = 2;
const double MAX_ITERATIONS = 300000000;
const double MIN_ITERATIONS = 3;
const double ITERATIONS_MULTIPLIER = 10;

int main()
{
    printf("Counts;\t\tThreads;\t\tResult;\t\t\tTime\n");

    for (long iteration_count = MIN_ITERATIONS;
        iteration_count <= MAX_ITERATIONS;
        iteration_count *= ITERATIONS_MULTIPLIER)
    {
        double step = (MAX_VALUE - MIN_VALUE) / iteration_count;
        double start_time = omp_get_wtime();
        double integral = 0;

#pragma omp parallel for reduction(+ : integral)
        for (int i = 0; i < iteration_count - 1; ++i)
        {
            double left_point = MIN_VALUE + step * i;
            double right_point = MIN_VALUE + step * (i + 1);
            double result = function((right_point + left_point) / 2);
            integral += (right_point - left_point) * result;
        }

        printf("%20ld;\t%20ld;\t%.20f;\t%.20f\n",
            iteration_count,
            omp_get_max_threads(),
            integral,
            omp_get_wtime() - start_time);
    }

    return 0;
}
```

### Результат выполнения программы:

Counts;	Threads;	Result;	Time
3;	32;	0.81420118343195246879;	0.00821120687760412693
30;	32;	1.15686086348399697066;	0.00265633407980203629
300;	32;	1.19028756043113759233;	0.00024750898592174053
3000;	32;	1.19362182740639100054;	0.00026386789977550507
30000;	32;	1.19395517007614460070;	0.00025541288778185844
300000;	32;	1.19398850350284146948;	0.00597935495898127556
3000000;	32;	1.19399183683711074266;	0.00335375801660120487
30000000;	32;	1.19399217017045300437;	0.03002636902965605259
300000000;	32;	1.19399220350377222033;	0.22206078213639557362

o [starostenkov\_aa@mng1 calc-task]\$ █

Рисунок 7 – Параллельное выполнение с интервалами интегрирования.



Рисунок 8 – Зависимость длительности вычислений от числа итераций для параллельной программы

5. Получить зависимость ускорения параллельного алгоритма  $S_{\text{пар}} = (T_{\text{послед}} / T_{\text{пар}})$ , где  $T_{\text{послед}}$  - время выполнения последовательного алгоритма,  $T_{\text{пар}}$  - время выполнения параллельного алгоритма, от числа отрезков разбиения интервала интегрирования  $n$ .



Рисунок 9 – Зависимость ускорения параллельного алгоритма

6. При максимальном  $n$  (из пункта 2 рабочего задания) получить зависимость времени выполнения параллельной программы от числа нитей, использующихся в параллельной секции. Число нитей изменять с помощью функции `omp_set_num_threads()` или переменной окружения `OMP_NUM_THREADS`.

Программа :

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    //Объявление переменных
    long i; // Переменная цикла
    int N = 200000000; // Число разбиений
    int threads_count = 0, j = 0;
    //double h = 1/(double)N; //Вычисление шага
    double timer_Start, timer_Stop, sum = 0;
    omp_set_num_threads(32);
    for(j = 2; j <= 32; j+=3)
    {
        omp_set_num_threads(j);
        sum = 0;
        double h = 1/(double)N;
        //Начальный отсчет таймера
        timer_Start = omp_get_wtime();
        //Распаралеливание программы
        #pragma omp parallel reduction(+:sum) private(i)
        {
            //Главный цикл
            #pragma omp for
            for(i = 1; i < N; i++)
            {
                double x = (h*(i-0.5)+1);
                sum += 5.0 * x / ((x + 1.0) * (x + 1.0));
            }
            threads_count = omp_get_num_threads();
        }
        //Конечный отсчет таймера
        timer_Stop = omp_get_wtime();
        //Вывод результатов
        printf("Number threads %d\n", threads_count);
        printf("Number partitions: %d\n", N);
        printf("Approximate value: %.5f\n", sum);
        printf("Calculation time: %.6f\n\n", timer_Stop - timer_Start);
    }
    return 0;
}
```

```
Number threads 8
Number partitions: 200000000
Approximate value: 238798440.33038
Calculation time: 0.157045

Number threads 11
Number partitions: 200000000
Approximate value: 238798440.33038
Calculation time: 0.118963

Number threads 14
Number partitions: 200000000
Approximate value: 238798440.33039
Calculation time: 0.097943

Number threads 17
Number partitions: 200000000
Approximate value: 238798440.33039
Calculation time: 0.101069

Number threads 20
Number partitions: 200000000
Approximate value: 238798440.33039
Calculation time: 0.108679

Number threads 23
Number partitions: 200000000
Approximate value: 238798440.33039
Calculation time: 0.094539

Number threads 26
Number partitions: 200000000
Approximate value: 238798440.33039
Calculation time: 0.084207

Number threads 29
Number partitions: 200000000
Approximate value: 238798440.33038
Calculation time: 0.076160

Number threads 32
Number partitions: 200000000
Approximate value: 238798440.33039
Calculation time: 0.071516

[starostenkov_aa@mng1 calc-task]$
```

Рисунок 10 – Выявление зависимости времени выполнения параллельной OpenMP программы от количества потоков

На рисунке 11 изображена рассматриваемая зависимость в виде графика.

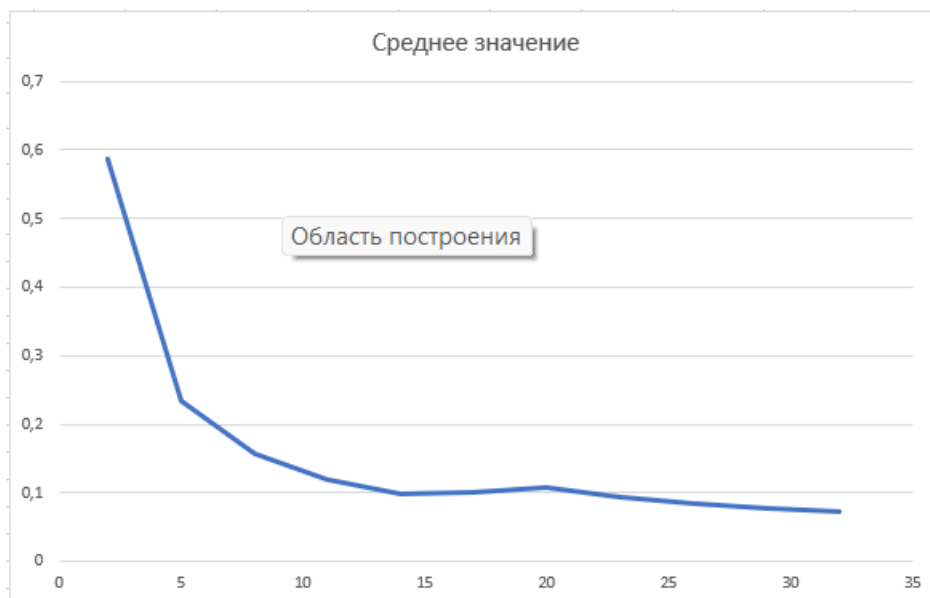


Рисунок 11 – График, отображающий зависимость времени выполнения параллельной OpenMP программы от количества потоков

На оси абсцисс отложено количество потоков в параллельной OpenMP программе, а на оси ординат время выполнения параллельной OpenMP программы.

Из данного графика:

1. Зависимость между временем выполнения и числом потоков не является пропорциональной.
2. Чем меньше количество потоков, тем дольше программа будет выполняться и, соответственно, наоборот, чем больше потоков, тем программа работает быстрее

7. Запустить параллельную программу на вычислительном узле 1 (ВУ1) и вычислительном узле 2 (ВУ2).

```

[starostenkov_aa@mng1 calc-task]$ srun -p node1_only -N 1 2

OPENMP DISPLAY ENVIRONMENT BEGIN
_OPENMP = '201511'
_OMP_DYNAMIC = 'FALSE'
_OMP_NESTED = 'FALSE'
_OMP_NUM_THREADS = '40'
_OMP_SCHEDULE = 'DYNAMIC'
_OMP_PROC_BIND = 'FALSE'
_OMP_PLACES = ''
_OMP_STACKSIZE = '0'
_OMP_WAIT_POLICY = 'PASSIVE'
_OMP_THREAD_LIMIT = '4294967295'
_OMP_MAX_ACTIVE_LEVELS = '2147483647'
_OMP_CANCELLATION = 'FALSE'
_OMP_DEFAULT_DEVICE = '0'
_OMP_MAX_TASK_PRIORITY = '0'
OPENMP DISPLAY ENVIRONMENT END
Counts;                Threads;                Result;                Time
200000000;             40;          1.193992201651910503; 0.11361449398100376129
^[[A^[[A^[[A[starostenkov_aa@mng1 calc-task]$ srun -p node2_only -N 1 2

OPENMP DISPLAY ENVIRONMENT BEGIN
_OPENMP = '201511'
_OMP_DYNAMIC = 'FALSE'
_OMP_NESTED = 'FALSE'
_OMP_NUM_THREADS = '40'
_OMP_SCHEDULE = 'DYNAMIC'
_OMP_PROC_BIND = 'FALSE'
_OMP_PLACES = ''
_OMP_STACKSIZE = '0'
_OMP_WAIT_POLICY = 'PASSIVE'
_OMP_THREAD_LIMIT = '4294967295'
_OMP_MAX_ACTIVE_LEVELS = '2147483647'
_OMP_CANCELLATION = 'FALSE'
_OMP_DEFAULT_DEVICE = '0'
_OMP_MAX_TASK_PRIORITY = '0'
OPENMP DISPLAY ENVIRONMENT END
Counts;                Threads;                Result;                Time
200000000;             40;          1.19399220165191932708; 0.12371742818504571915
[starostenkov_aa@mng1 calc-task]$

```

Рисунок 12 – Запуск параллельной программы на ВУ1 и ВУ2

8. Сравнить время вычисления параллельной программы при максимальном  $n$  (из пункта 2) и **максимальном** числе нитей на узлах УУ, ВУ1, ВУ2.

9. Сравнить время вычисления параллельной программы при максимальном  $n$  и **одинаковом (32)** числе нитей на узлах УУ, ВУ1, ВУ2.

Результат запуска параллельной OpenMP-программы при максимальном  $n$  и максимальном числе нитей, а также при максимальном  $n$  и одинаковом числе нитей на узлах УУ, ВУ1, ВУ2 представлен на рисунках ниже:

```

● [starostenkov_aa@mng1 calc-task]$ srun -p node1_only -N 1 2

OPENMP DISPLAY ENVIRONMENT BEGIN
_OPENMP = '201511'
_OMP_DYNAMIC = 'FALSE'
_OMP_NESTED = 'FALSE'
_OMP_NUM_THREADS = '40'
_OMP_SCHEDULE = 'DYNAMIC'
_OMP_PROC_BIND = 'FALSE'
_OMP_PLACES = ''
_OMP_STACKSIZE = '0'
_OMP_WAIT_POLICY = 'PASSIVE'
_OMP_THREAD_LIMIT = '4294967295'
_OMP_MAX_ACTIVE_LEVELS = '2147483647'
_OMP_CANCELLATION = 'FALSE'
_OMP_DEFAULT_DEVICE = '0'
_OMP_MAX_TASK_PRIORITY = '0'
OPENMP DISPLAY ENVIRONMENT END
Counts;          Threads;          Result;          Time
                200000000;          40;  1.19399220165191910503; 0.11361449398100376129
● ^[[A^[[A^[[A[starostenkov_aa@mng1 calc-task]$ srun -p node2_only -N 1 2

OPENMP DISPLAY ENVIRONMENT BEGIN
_OPENMP = '201511'
_OMP_DYNAMIC = 'FALSE'
_OMP_NESTED = 'FALSE'
_OMP_NUM_THREADS = '40'
_OMP_SCHEDULE = 'DYNAMIC'
_OMP_PROC_BIND = 'FALSE'
_OMP_PLACES = ''
_OMP_STACKSIZE = '0'
_OMP_WAIT_POLICY = 'PASSIVE'
_OMP_THREAD_LIMIT = '4294967295'
_OMP_MAX_ACTIVE_LEVELS = '2147483647'
_OMP_CANCELLATION = 'FALSE'
_OMP_DEFAULT_DEVICE = '0'
_OMP_MAX_TASK_PRIORITY = '0'
OPENMP DISPLAY ENVIRONMENT END
Counts;          Threads;          Result;          Time
                200000000;          40;  1.19399220165191932708; 0.12371742818504571915
○ [starostenkov_aa@mng1 calc-task]$ █

● [starostenkov_aa@mng1 calc-task]$ srun -p mng1_only -N 1 2

OPENMP DISPLAY ENVIRONMENT BEGIN
_OPENMP = '201511'
_OMP_DYNAMIC = 'FALSE'
_OMP_NESTED = 'FALSE'
_OMP_NUM_THREADS = '32'
_OMP_SCHEDULE = 'DYNAMIC'
_OMP_PROC_BIND = 'FALSE'
_OMP_PLACES = ''
_OMP_STACKSIZE = '0'
_OMP_WAIT_POLICY = 'PASSIVE'
_OMP_THREAD_LIMIT = '4294967295'
_OMP_MAX_ACTIVE_LEVELS = '2147483647'
_OMP_CANCELLATION = 'FALSE'
_OMP_DEFAULT_DEVICE = '0'
_OMP_MAX_TASK_PRIORITY = '0'
OPENMP DISPLAY ENVIRONMENT END
Counts;          Threads;          Result;          Time
                200000000;          32;  1.19399220165192287979; 0.16240891581401228905

```

Рисунок 13 – Вычисления параллельной OpenMP-программы при максимальном n на узлах УУ, ВУ1, ВУ2

```
[starostenkov_aa@mng1 calc-task]$ srun -p node1_only -N 1 2

OPENMP DISPLAY ENVIRONMENT BEGIN
_OPENMP = '201511'
_OMP_DYNAMIC = 'FALSE'
_OMP_NESTED = 'FALSE'
_OMP_NUM_THREADS = '32'
_OMP_SCHEDULE = 'DYNAMIC'
_OMP_PROC_BIND = 'FALSE'
_OMP_PLACES = ''
_OMP_STACKSIZE = '0'
_OMP_WAIT_POLICY = 'PASSIVE'
_OMP_THREAD_LIMIT = '4294967295'
_OMP_MAX_ACTIVE_LEVELS = '2147483647'
_OMP_CANCELLATION = 'FALSE'
_OMP_DEFAULT_DEVICE = '0'
_OMP_MAX_TASK_PRIORITY = '0'
OPENMP DISPLAY ENVIRONMENT END
Counts;          Threads;          Result;          Time
                200000000;          32;    1.19399220165192332388; 0.13592280866578221321
[starostenkov_aa@mng1 calc-task]$ srun -p node2_only -N 1 2

OPENMP DISPLAY ENVIRONMENT BEGIN
_OPENMP = '201511'
_OMP_DYNAMIC = 'FALSE'
_OMP_NESTED = 'FALSE'
_OMP_NUM_THREADS = '32'
_OMP_SCHEDULE = 'DYNAMIC'
_OMP_PROC_BIND = 'FALSE'
_OMP_PLACES = ''
_OMP_STACKSIZE = '0'
_OMP_WAIT_POLICY = 'PASSIVE'
_OMP_THREAD_LIMIT = '4294967295'
_OMP_MAX_ACTIVE_LEVELS = '2147483647'
_OMP_CANCELLATION = 'FALSE'
_OMP_DEFAULT_DEVICE = '0'
_OMP_MAX_TASK_PRIORITY = '0'
OPENMP DISPLAY ENVIRONMENT END
Counts;          Threads;          Result;          Time
                200000000;          32;    1.19399220165192310183; 0.15437670005485415459
[starostenkov_aa@mng1 calc-task]$
```

```
• [starostenkov_aa@mng1 calc-task]$ srun -p mng1_only -N 1 2

OPENMP DISPLAY ENVIRONMENT BEGIN
_OPENMP = '201511'
_OMP_DYNAMIC = 'FALSE'
_OMP_NESTED = 'FALSE'
_OMP_NUM_THREADS = '32'
_OMP_SCHEDULE = 'DYNAMIC'
_OMP_PROC_BIND = 'FALSE'
_OMP_PLACES = ''
_OMP_STACKSIZE = '0'
_OMP_WAIT_POLICY = 'PASSIVE'
_OMP_THREAD_LIMIT = '4294967295'
_OMP_MAX_ACTIVE_LEVELS = '2147483647'
_OMP_CANCELLATION = 'FALSE'
_OMP_DEFAULT_DEVICE = '0'
_OMP_MAX_TASK_PRIORITY = '0'
OPENMP DISPLAY ENVIRONMENT END
Counts;          Threads;          Result;          Time
                200000000;          32;    1.19399220165192287979; 0.16240891581401228905
```

Рисунок 14 – Вычисления параллельной OpenMP-программы при одинаковом числе нитей на узлах УУ, ВУ1, ВУ2



При максимальном числе нитей на узлах ВУ1 и ВУ2 программа работает быстрее, чем на УУ. Также можно отметить, что на УУ вычисления медленнее, чем на ВУ1 и ВУ2 даже при одинаковом числе нитей, хоть и в меньшей степени.

10. Написать, отладить, скомпилировать и запустить на гибридном вычислительном кластере СФМЭИ (на УУ) MPI-программу вычисления определенного интеграла. Предусмотреть замер времени выполнения программы, контроль правильности вычисления интеграла. Индивидуальные задания в соответствии с номером по журналу взять из таблицы 1.

Листинг параллельной MPI программы:

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
    int N;                // Число разбиений
    double sum = 0;        // Сумма ряда
    double sum1;          // Частичная сумма ряда
    double timer_start, timer_stop; // Переменные измерения времени
    int size;              // Количество процессов
    int rank;              // Номер процесса
    sum = 0;
    sum1 = 0;
    // Инициализация
    MPI_Init(&argc, &argv);
    // Сведения о количестве процессов
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Сведения о номерах процессов
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    for (N = 2000; N <= 200000000; N += 2222000)
    {
        // Начальный отсчет таймера
        if (rank == 0)
        {
            timer_start = MPI_Wtime();
        }
        // Задаем шаг
        double h = 1 / (double)N;
        sum1 = 0;
        double x;
        for (int i = rank + 1; i <= N; i += size)
        {
```

```

        x = (h * (i - 0.5) + 1);
        sum1 += 5.0 * x * h / ((x + 1.0) * (x + 1.0));
    }
    // Подсчет общей суммы ряда в процесс с номером 0
    MPI_Reduce(&sum1, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    // Если процесс №0
    if (rank == 0)
    {
        timer_stop = MPI_Wtime();
        // Вывод результатов
        printf("Число разбиений: %d\n", N);
        printf("Количество процессов: %d\n", size);
        printf("Приближенное значение: %.5f\n", sum);
        printf("Время вычислений: %.6f\n\n", timer_stop - timer_start);
    }
}
MPI_Finalize();
return 0;
}

```

Необходимо получить зависимость времени выполнения параллельной MPI программы от числа N. На рисунке 15 продемонстрированы замеры времени. Замеры проводились на УУ.

```
● [starostenkov_aa@mng1 calc-task]$ srun 6
Число разбиений: 2000
Количество процессов: 1
Приближенное значение: 1.19399
Время вычислений: 0.000037

Число разбиений: 22224000
Количество процессов: 1
Приближенное значение: 1.19399
Время вычислений: 0.169715

Число разбиений: 44446000
Количество процессов: 1
Приближенное значение: 1.19399
Время вычислений: 0.246085

Число разбиений: 66668000
Количество процессов: 1
Приближенное значение: 1.19399
Время вычислений: 0.369227

Число разбиений: 88890000
Количество процессов: 1
Приближенное значение: 1.19399
Время вычислений: 0.455278

Число разбиений: 111112000
Количество процессов: 1
Приближенное значение: 1.19399
Время вычислений: 0.512950

Число разбиений: 133334000
Количество процессов: 1
Приближенное значение: 1.19399
Время вычислений: 0.615866

Число разбиений: 155556000
Количество процессов: 1
Приближенное значение: 1.19399
Время вычислений: 0.718114

Число разбиений: 177778000
Количество процессов: 1
Приближенное значение: 1.19399
Время вычислений: 0.822446

Число разбиений: 200000000
Количество процессов: 1
Приближенное значение: 1.19399
Время вычислений: 0.924463

○ [starostenkov_aa@mng1 calc-task]$
```

Рисунок 15 - Зависимость времени от числа разбиений в параллельной  
MPI программе

Занесем полученные результаты в таблицу 2.

Таблица 1 – Выявление зависимости времени от числа N в параллельной MPI программе

Число отрезков N	Время в секундах
2000	0,000037
22224000	0,169715
44446000	0,246085
66668000	0,369227
88890000	0,455278
111112000	0,512950
133334000	0,615866
155556000	0,718114
177778000	0,822446
200000000	0,924463

Зависимость времени выполнения параллельной MPI программы от числа отрезков разбиения интервала интегрирования на рисунке 16.



Рисунок 16 - График зависимости времени от числа N для параллельной MPI программы

Из данного графика видна зависимость времени выполнения параллельной MPI программы от числа N. При достаточно больших значениях N эту зависимость можно принять прямо пропорциональной.

Необходимо получить зависимость ускорения параллельной MPI программы по сравнению с последовательной программой в зависимости от числа N. Посчитаем коэффициент ускорения и результаты представим в виде таблицы 3.

Таблица 2 – Вычисление коэффициента ускорения

Число отрезков N	Время выполнения последовательной программы в секундах	Время выполнения параллельной MPI программы в секундах	Коэффициент ускорения $S_{omp}$
2000	0,000073	0,000037	0,58139535
22224000	0,471294	0,169715	1,89106268
44446000	0,751228	0,246085	3,26441449
66668000	1,047871	0,369227	8,61180645
88890000	1,275462	0,455278	8,74119139
111112000	1,593059	0,512950	9,21562614
133334000	1,909386	0,615866	19,1859508
155556000	2,346871	0,718114	14,4057306
177778000	2,661239	0,822446	30,2745377
200000000	2,876409	0,924463	19,745725

На рисунке 17 продемонстрирован график зависимости коэффициента ускорения параллельной MPI программы от числа N



Рисунок 17 – График зависимости коэффициента ускорения параллельной MPI программы от числа N

Выявление зависимости времени выполнения параллельной MPI программы от количества процессов:

Необходимо получить зависимость коэффициента ускорения от числа процессов распараллеливание при количестве отрезков разбиения равном  $N_{\max}$ .

Результаты выполнения программы представлены на рисунках 18, 19.

```

● [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 2 mpirun 7
salloc: Granted job allocation 43725
Number of partitions: 200000000
Number of processes: 2
Approximate value: 1.19399
Calculation time: 1.597172

salloc: Relinquishing job allocation 43725
● [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 5 mpirun 7
salloc: Granted job allocation 43729
Number of partitions: 200000000
Number of processes: 5
Approximate value: 1.19399
Calculation time: 0.766767

salloc: Relinquishing job allocation 43729
● [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 8 mpirun 7
salloc: Granted job allocation 43730
Number of partitions: 200000000
Number of processes: 8
Approximate value: 1.19399
Calculation time: 0.520363

salloc: Relinquishing job allocation 43730
● [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 11 mpirun 7
salloc: Granted job allocation 43731
Number of partitions: 200000000
Number of processes: 11
Approximate value: 1.19399
Calculation time: 0.383678

salloc: Relinquishing job allocation 43731
● [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 14 mpirun 7
salloc: Granted job allocation 43733
Number of partitions: 200000000
Number of processes: 14
Approximate value: 1.19399
Calculation time: 0.289383

salloc: Relinquishing job allocation 43733
● [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 17 mpirun 7
salloc: Granted job allocation 43735
Number of partitions: 200000000
Number of processes: 17
Approximate value: 1.19399
Calculation time: 0.239580

salloc: Relinquishing job allocation 43735

```

Рисунок 18 – Исследование зависимости времени выполнения программы от количества процессов (часть 1)

```

salloc: Relinquishing job allocation 43735
● [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 20 mpirun 7
salloc: Granted job allocation 43736
Number of partitions: 200000000
Number of processes: 20
Approximate value: 1.19399
Calculation time: 0.202581

salloc: Relinquishing job allocation 43736
● [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 23 mpirun 7
salloc: Granted job allocation 43738
^[[ANumber of partitions: 200000000
Number of processes: 23
Approximate value: 1.19399
Calculation time: 0.167229

salloc: Relinquishing job allocation 43738
● [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 26 mpirun 7
salloc: Granted job allocation 43739
Number of partitions: 200000000
Number of processes: 26
Approximate value: 1.19399
Calculation time: 0.149522

salloc: Relinquishing job allocation 43739
● [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 29 mpirun 7
salloc: Granted job allocation 43740
Number of partitions: 200000000
Number of processes: 29
Approximate value: 1.19399
Calculation time: 0.144465

salloc: Relinquishing job allocation 43740
● [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 32 mpirun 7
salloc: Granted job allocation 43741
Number of partitions: 200000000
Number of processes: 32
Approximate value: 1.19399
Calculation time: 0.127424

salloc: Relinquishing job allocation 43741
○ [starostenkov_aa@mng1 calc-task]$ █

```

Рисунок 19 – Исследование зависимости времени выполнения программы от количества процессов (часть 2)

Сведем полученные результаты в таблицу 4.



Таблица 3 – Выявление зависимости времени выполнения параллельной MPI программы от количества потоков

Число потоков	Время в секундах
2	1,597172
5	0,766767
8	0,520363
11	0,383678
14	0,289383
17	0,239580
20	0,202581
23	0,167229
26	0,149522
29	0,144465
32	0,127424

График зависимости времени выполнения параллельной MPI программы от количества используемых процессов представлен на рисунке 20.

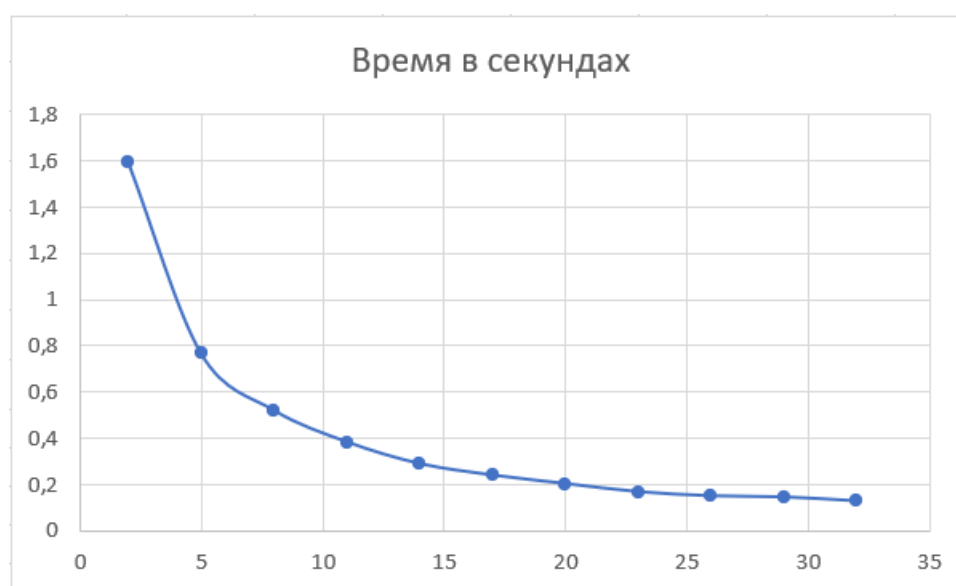
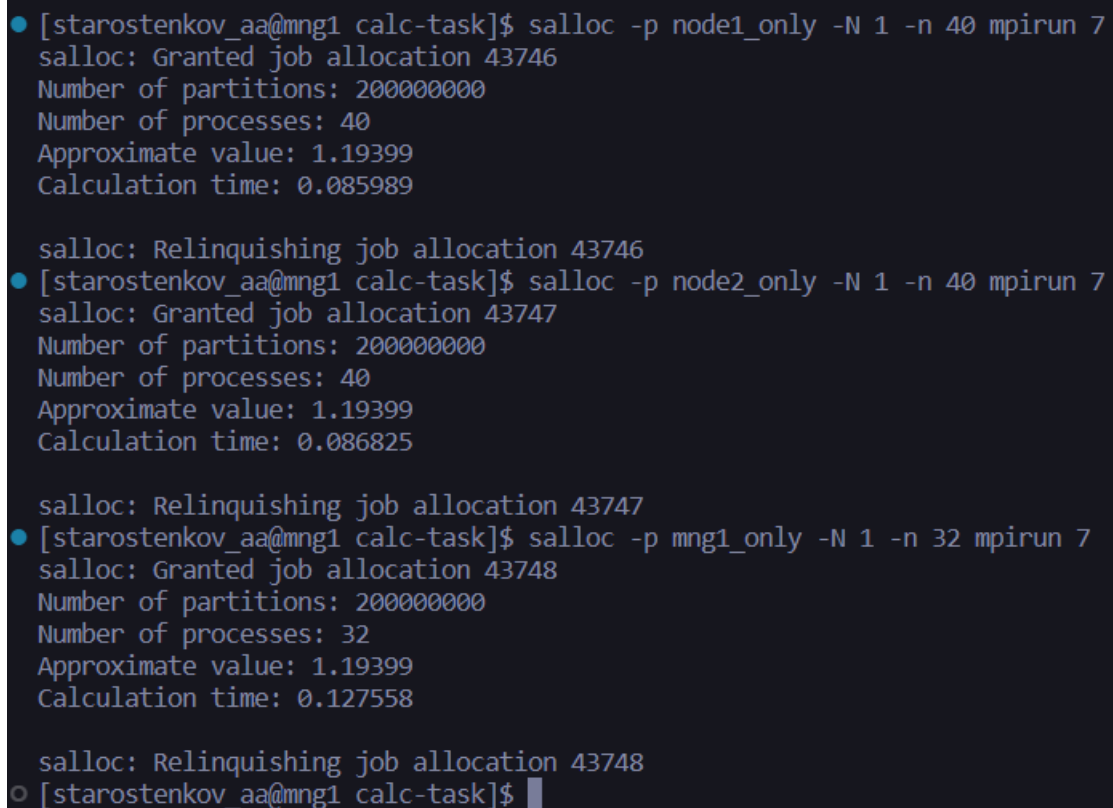


Рисунок 20 - График зависимости времени выполнения параллельной MPI программы от количества используемых процессов

На данном графике ось абсцисс – это количество нитей, ось ординат – это время выполнения. Видно, что зависимость не является пропорциональной.

Необходимо запустить параллельную MPI программу на узлах ВУ1, ВУ2 и УУ с максимальным числом процессов. Результаты представлены на рисунке 21.



```
• [starostenkov_aa@mng1 calc-task]$ salloc -p node1_only -N 1 -n 40 mpirun 7
salloc: Granted job allocation 43746
Number of partitions: 200000000
Number of processes: 40
Approximate value: 1.19399
Calculation time: 0.085989

salloc: Relinquishing job allocation 43746
• [starostenkov_aa@mng1 calc-task]$ salloc -p node2_only -N 1 -n 40 mpirun 7
salloc: Granted job allocation 43747
Number of partitions: 200000000
Number of processes: 40
Approximate value: 1.19399
Calculation time: 0.086825

salloc: Relinquishing job allocation 43747
• [starostenkov_aa@mng1 calc-task]$ salloc -p mng1_only -N 1 -n 32 mpirun 7
salloc: Granted job allocation 43748
Number of partitions: 200000000
Number of processes: 32
Approximate value: 1.19399
Calculation time: 0.127558

salloc: Relinquishing job allocation 43748
○ [starostenkov_aa@mng1 calc-task]$
```

Рисунок 21 – Запуск MPI программы на разных узлах с максимальным числом процессов

ВУ1 и ВУ2 при максимальном числе процессоров были бы быстрее УУ из-за того, что у них больше процессоров. При равном числе процессоров они все равно будут немного быстрее.

При максимальном  $n$  продлить полученную в пункте 6 для MPI-программы зависимость времени выполнения параллельной программы от числа процессов, используя запуск программы на двух вычислительных узлах (число процессов – до 80), на трех узлах (число процессов – до 112).

На двух ВУ время вычисления равно 0.064018

На двух ВУ и УУ время вычисления равно 0.044761

```
[starostenkov_aa@mng1 calc-task]$ salloc -p comp_nodes -N 2 -n 80 mpirun 7
salloc: Granted job allocation 43749
Number of partitions: 200000000
Number of processes: 80
Approximate value: 1.19399
Calculation time: 0.064018

salloc: Relinquishing job allocation 43749
[starostenkov_aa@mng1 calc-task]$ salloc -p all_nodes -N 3 -n 112 mpirun 7
salloc: Granted job allocation 43750
Number of partitions: 200000000
Number of processes: 112
Approximate value: 1.19399
Calculation time: 0.044761

salloc: Relinquishing job allocation 43750
[starostenkov_aa@mng1 calc-task]$
```

Рисунок 22 – Запуск MPI программы на 80 и 112 процессорах

### Сравнение параллельных OpenMP и MPI программ

Для сравнения параллельных OpenMP и MPI программ необходимо изобразить полученные зависимости на одних графиках. Графики представлены на рисунках 23-24.

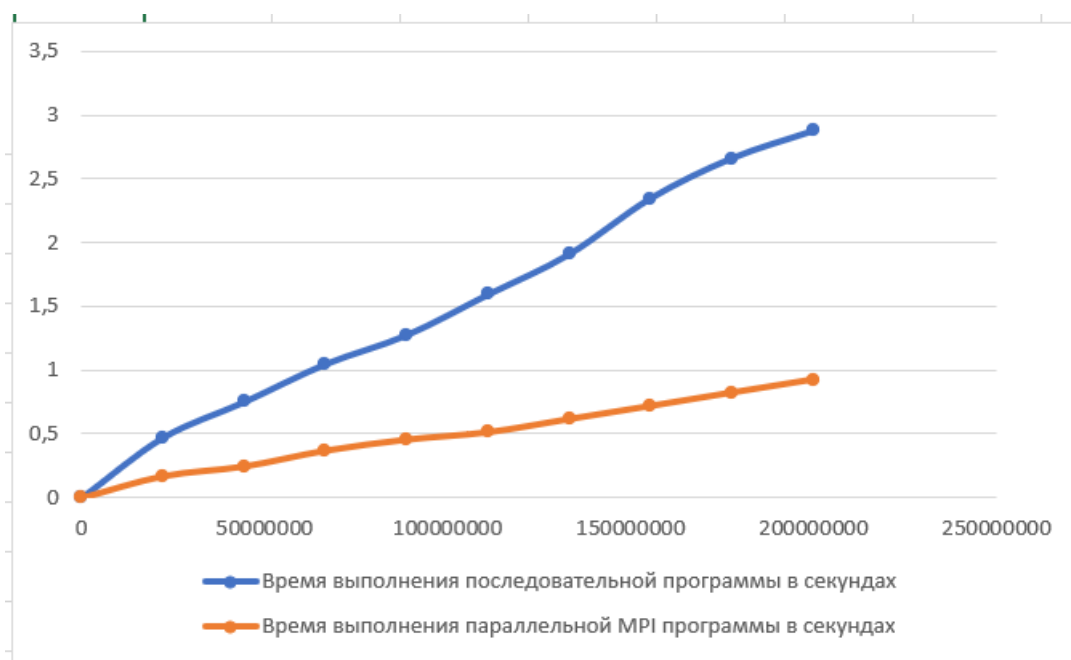


Рисунок 23 – Зависимость времени выполнения программы от числа N

Из графика 23 можно сделать следующие выводы:

1. В зависимости времени выполнения OpenMP и MPI программ от числа N результаты получились схожими, как только значения N становятся достаточно большими MPI программа становится сильно быстрее быстрее.
2. Зависимость времени выполнения программ от числа потоков/процессов практически одинаковая.

### **Заключение**

В ходе выполнения расчетно-графической работы были написаны следующие программы:

- Последовательная программа приближенного вычисления определенного интеграла.
- Параллельная OpenMP программа приближенного вычисления определенного интеграла.
- Параллельная MPI программа приближенного вычисления определенного интеграла.

Также была исследована работа программ на гибридном вычислительном кластере СФМЭИ при различных параметрах (количество параллельных нитей/процессов, количество отрезков разбиения). В ходе исследования были получены зависимости временных характеристик от исследуемых параметров.

Запущена программа на нескольких узлах и проверено время обработки.

Наибольшее ускорение по сравнению с последовательной программой при равных количествах нитей/процессов удалось получить при помощи MPI подхода к распараллеливанию.