**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 Sпар=(Tпосл/Tпар), где Tпосл - время выполнения последовательного алгоритма, Tпар – 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 (ВУ1) и вычислительном узле 2 (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.

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.

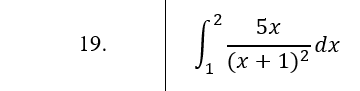
11. Повторить пункты 4-9 для MPI-программы. Естественно, число процессов для MPI задается средой выполнения.

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.



**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 = 300000000;

    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:**

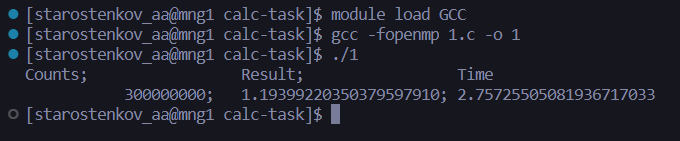
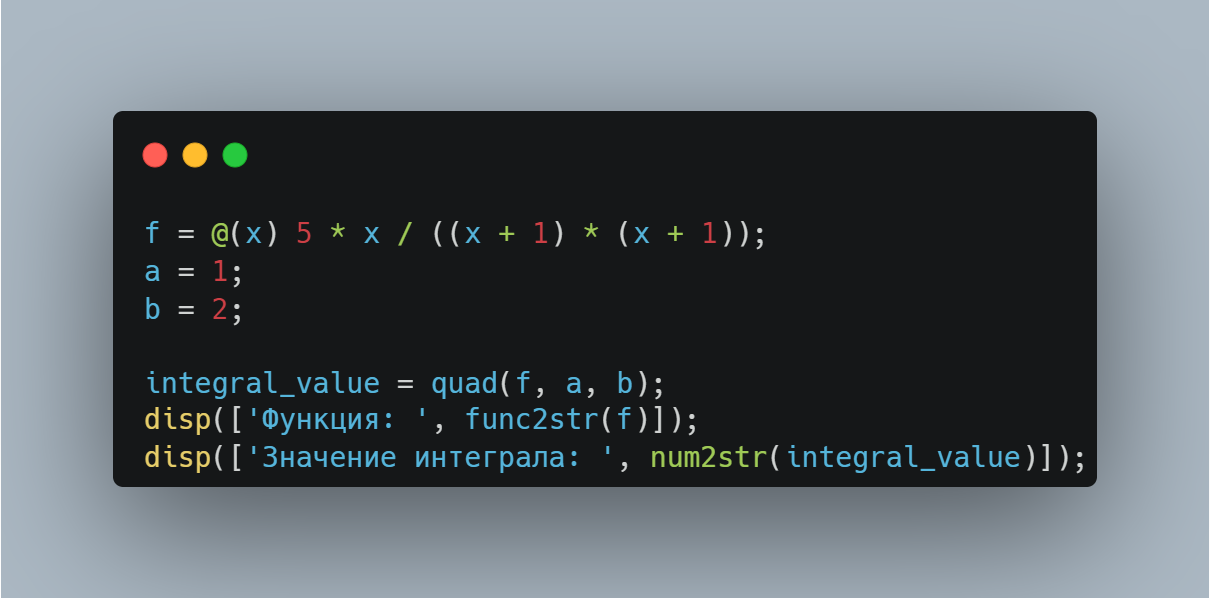
****

Рисунок 1 – Последовательное вычисление интеграла.

Проверим правильность вычисления в математическом пакете Octave 8.2.0.



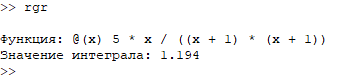


Рисунок 2 – Вычисление интеграла.

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:**

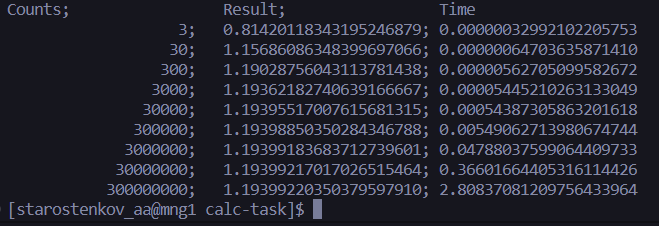
****

Рисунок 4 – Последовательное выполнение с интервалами интегрирования.

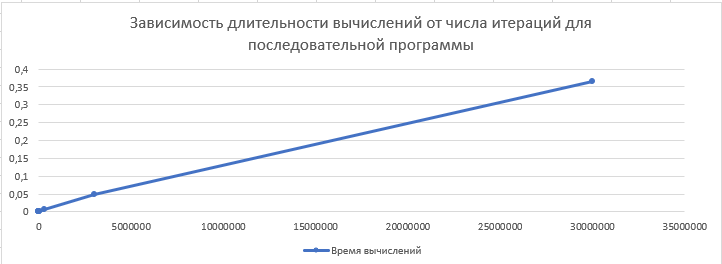


Рисунок 5 – График зависимости длительности вычислений от числа итераций

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\tResult;\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%20ld;\t%.20f;\t%.20f\n",

           iteration\_count,

           omp\_get\_max\_threads(),

           integral,

           omp\_get\_wtime() - start\_time);

    return 0;

}

Результат выполнения программы:

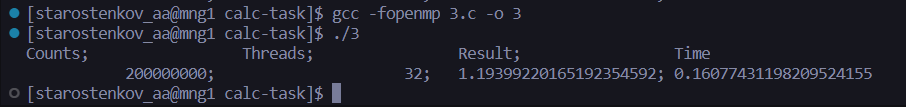
****

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\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:**

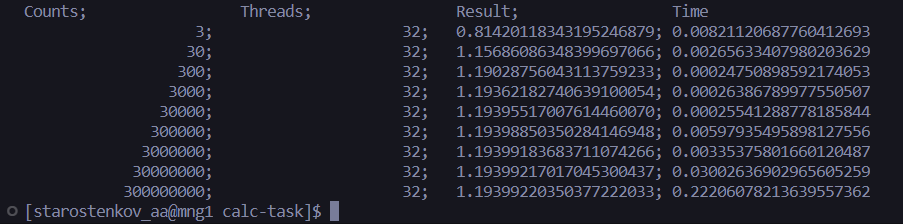


Рисунок 7 – Параллельное выполнение с интервалами интегрирования.

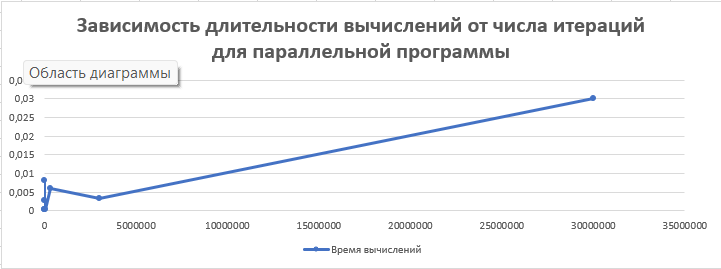
****

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

5. Получить зависимость ускорения параллельного алгоритма Sпар=(Tпосл/Tпар), где Tпосл - время выполнения последовательного алгоритма, Tпар – время выполнения параллельного алгоритма, от числа отрезков разбиения интервала интегрирования *n.*

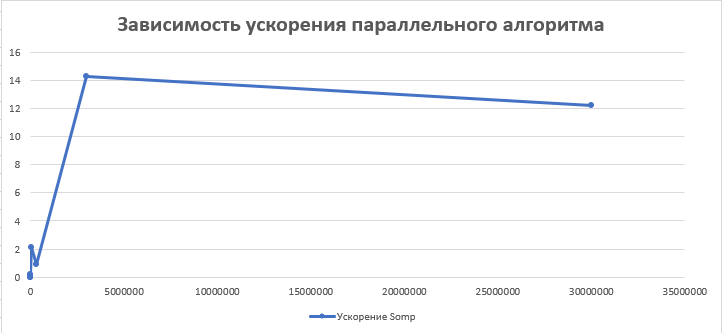
****

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();

*//Расспаралеливание 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;

}

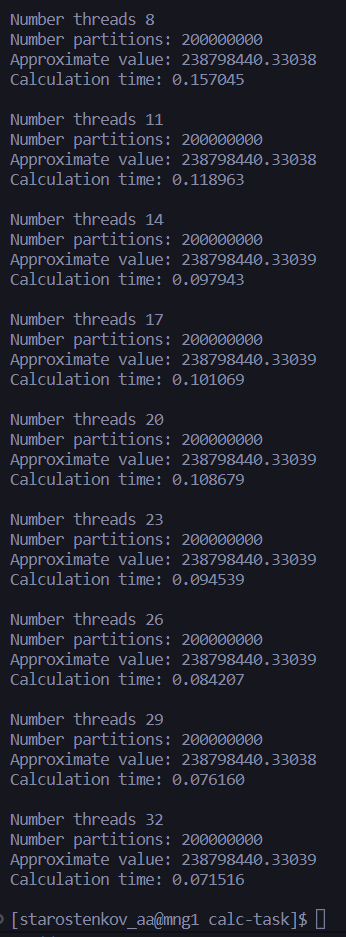


Рисунок 10 – Выявление зависимости времени выполнения параллельной OpenMP программы от количества потоков

На рисунке 11 изображена рассматриваемая зависимость в виде графика.

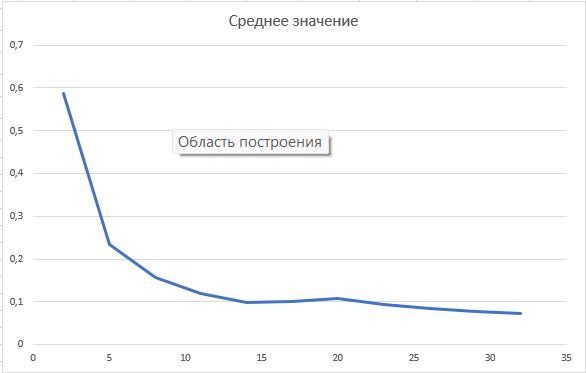


Рисунок 11 – График, отображающий зависимость времени выполнения параллельной OpenMP программы от количества потоков

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).

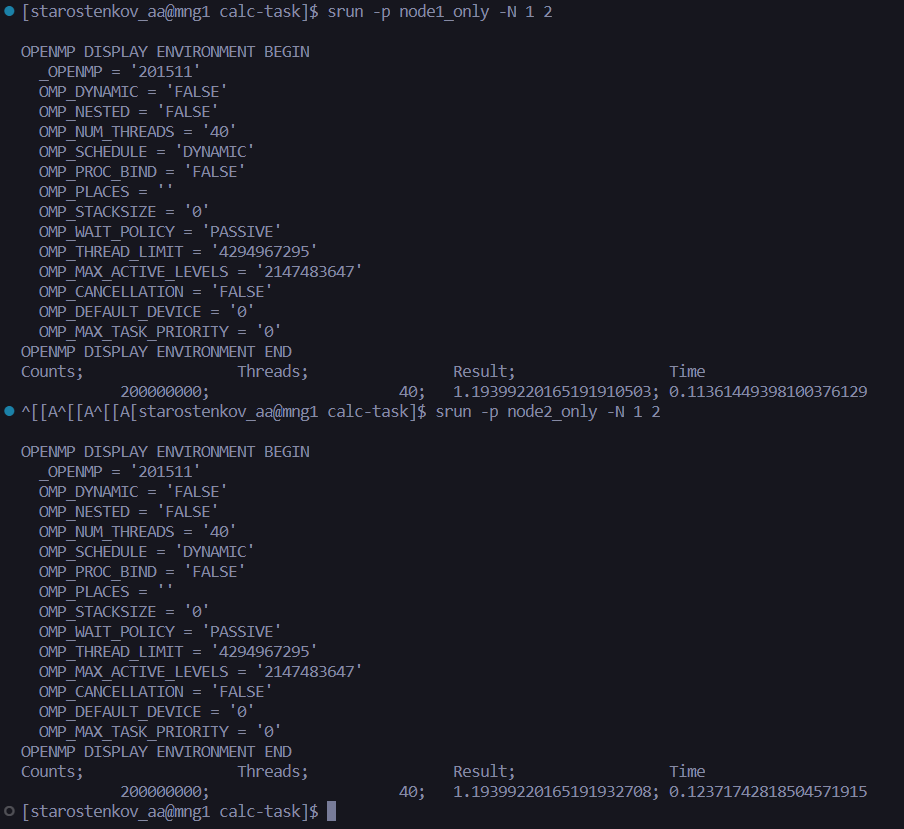
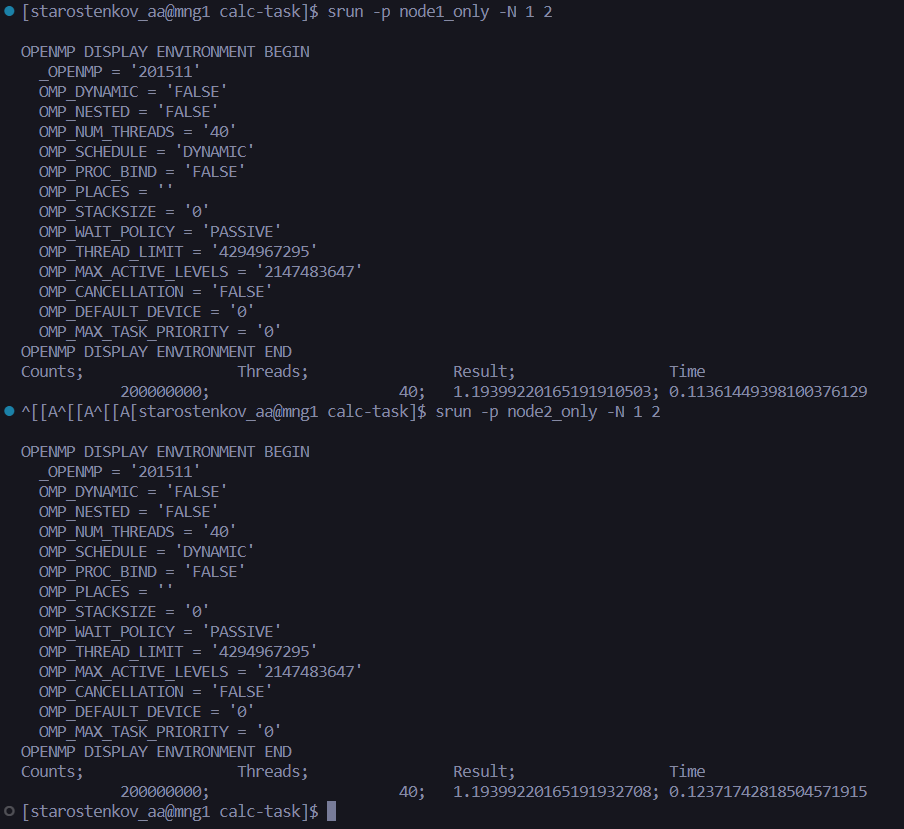


Рисунок 12 – Запуск параллельной программы на ВУ1 и ВУ2

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:



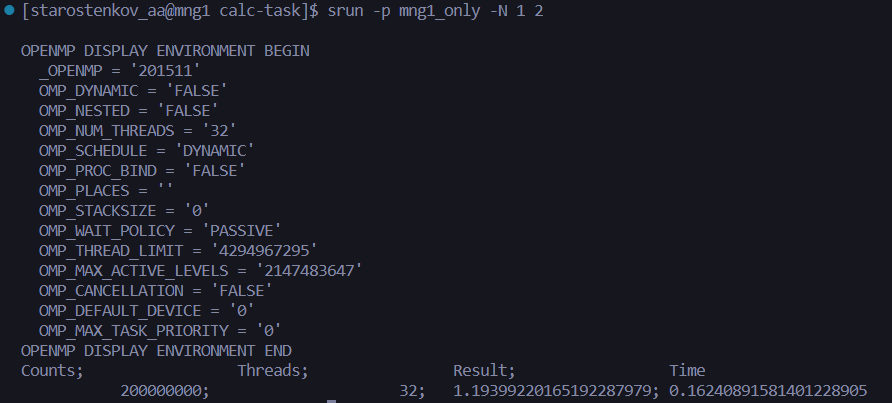
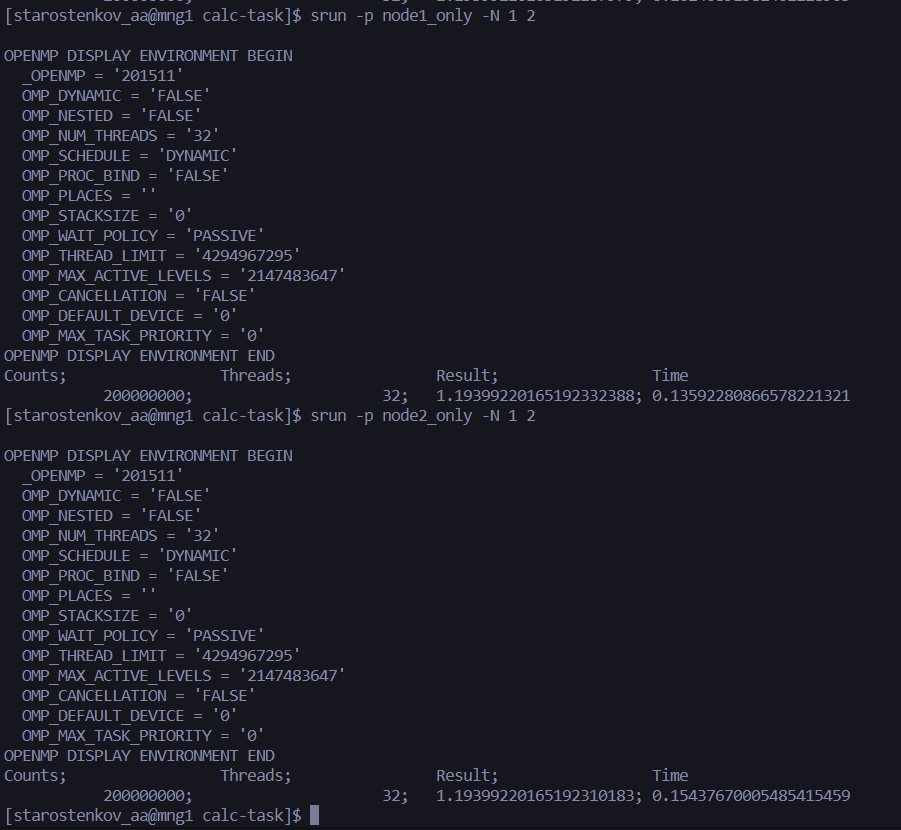


Рисунок 13 – Вычисления параллельной OpenMP-программы при максимальном n на узлах УУ, ВУ1, ВУ2



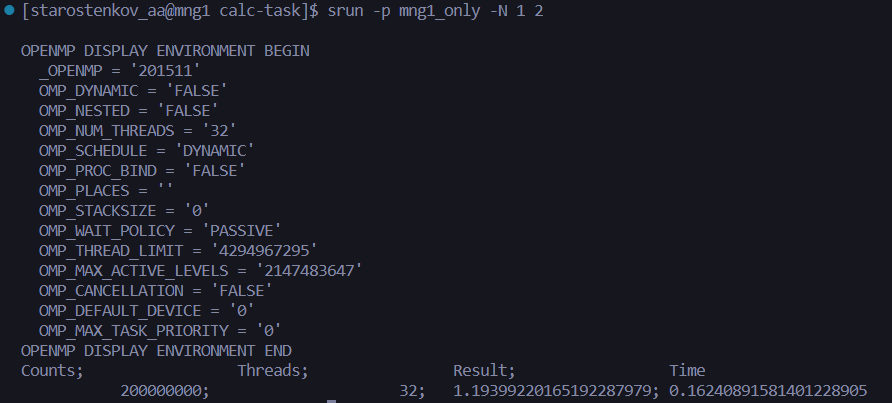


Рисунок 14 – Вычисления параллельной OpenMP-программы при одинаковом числе нитей на узлах УУ, ВУ1, ВУ2

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 += 22222000)

    {

*// 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.

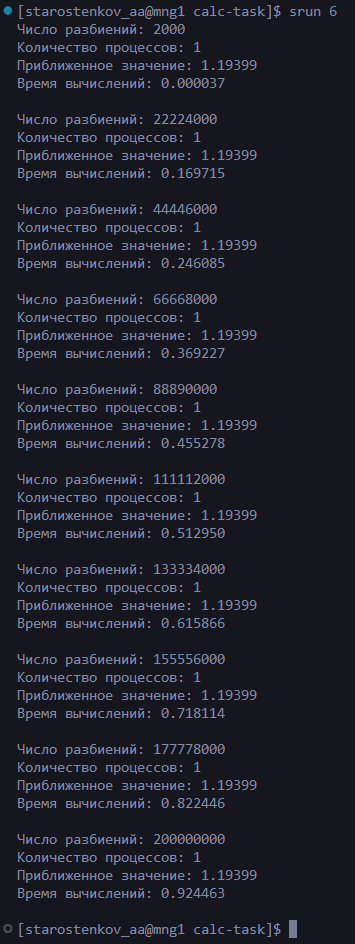


Рисунок 15 - Зависимость времени от числа разбиений в параллельной MPI программе

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 Somp |
| 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 Nmax.

Результаты выполнения программы представлены на рисунках 18, 19.

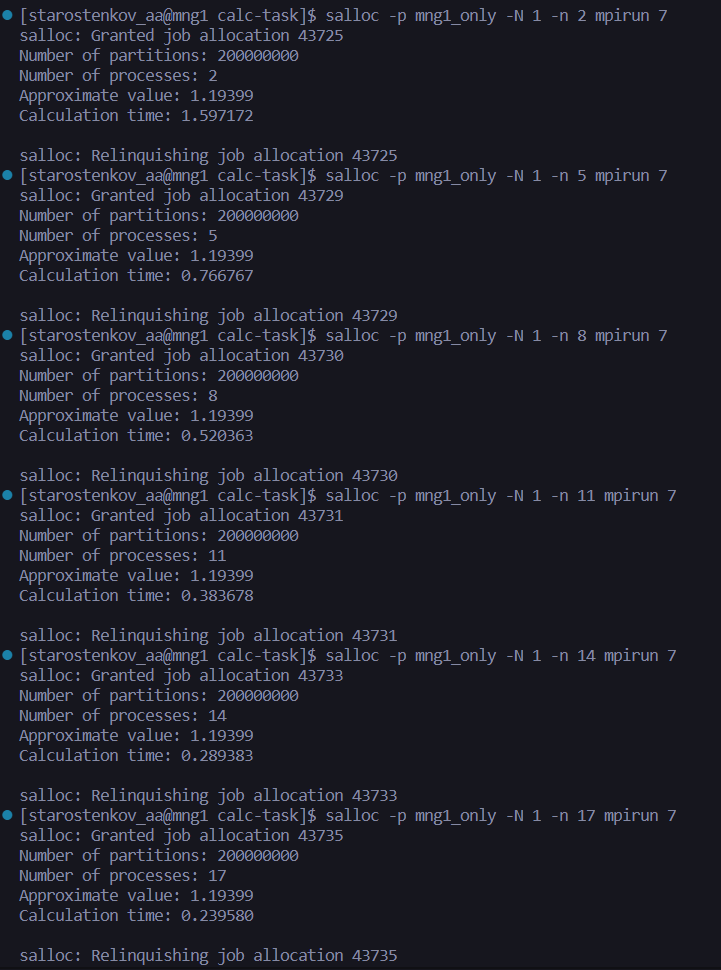


Рисунок 18 – Исследование зависимости времени выполнения программы от количества процессов (часть 1)

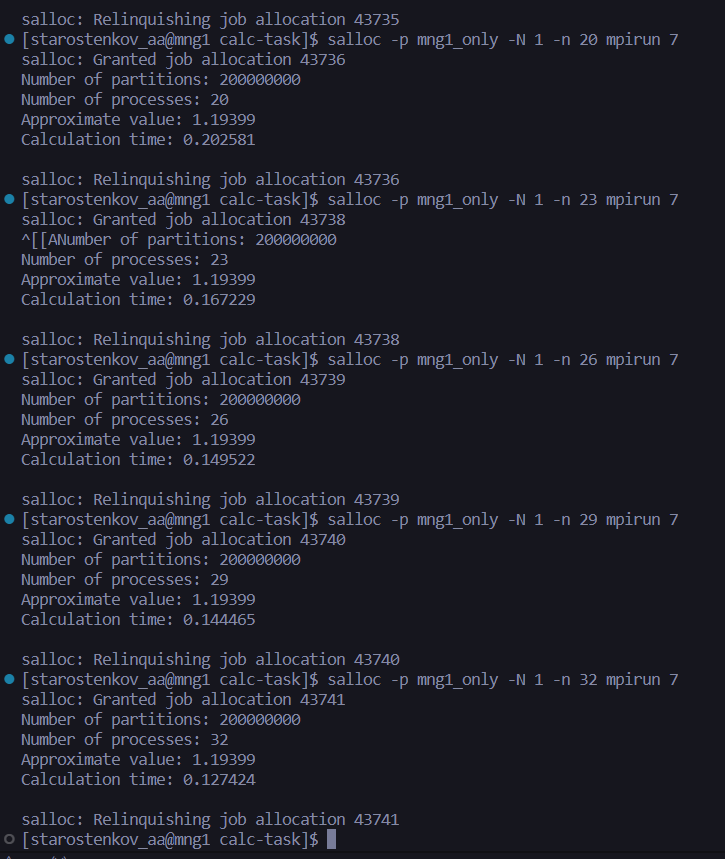


Рисунок 19 – Исследование зависимости времени выполнения программы от количества процессов (часть 2)

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.

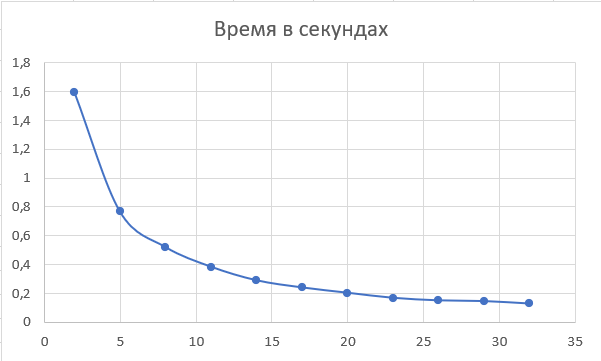


Рисунок 20 - График зависимости времени выполнения параллельной MPI программы от количества используемых процессов

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.

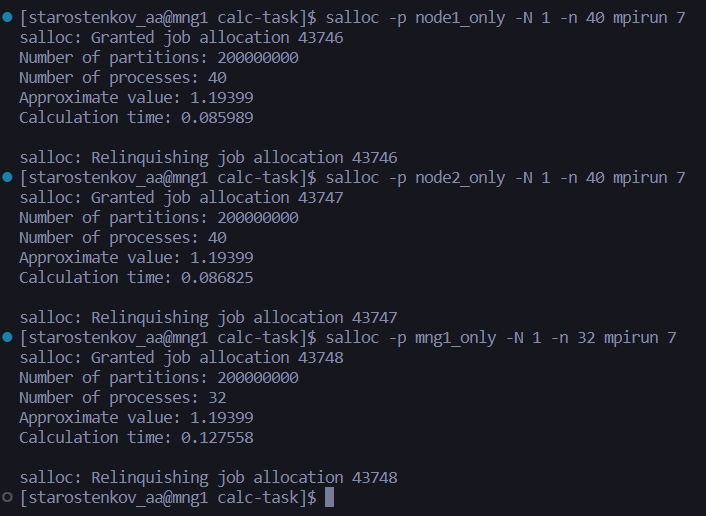


Рисунок 21 – Запуск MPI программы на разных узлах с максимальным числом процессов

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

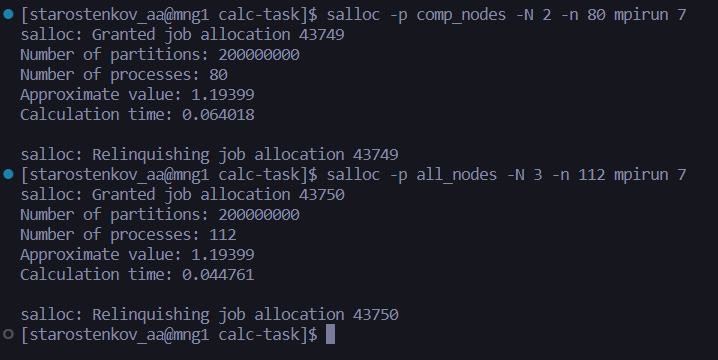


Рисунок 22 – Запуск MPI программы на 80 и 112 процессорах

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.

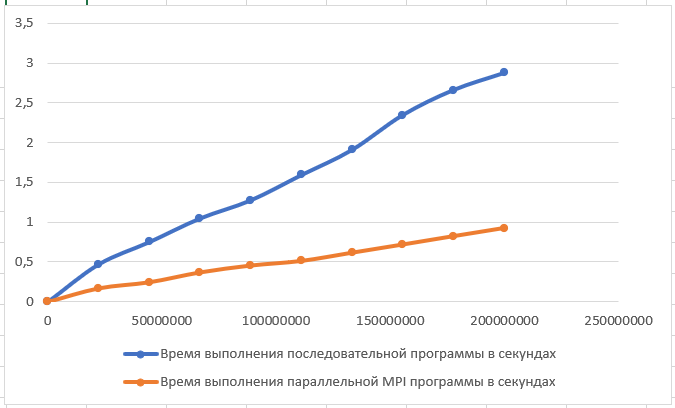


Рисунок 23 – Зависимость времени выполнения программы от числа N

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 становиться сильно быстрее быстрей.

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.