**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»

Лабораторная работа №1

«**Защита от гонок в OpenMP**»

по курсу:

«Вычислительные системы»

Студент: Старостенков А.А.

Группа: ВМ-22(маг)

Вариант: 19

Преподаватель: Федулов А.С.

Смоленск, 2023

1. Написать на языке Си, скомпилировать, отладить и запустить на гибридном вычислительном кластере (ГВК) СФМЭИ **последовательную** программу вычисления суммы числового ряда: , где  - общий член ряда. Вариант задания (общий член ряда) выбрать в таблице (по номеру журнала). Предусмотреть вывод результата. В теле цикла выделить вычисление общего члена ряда и накапливающее суммирование членов ряда.
2. Выполнить проверку правильности вычисления суммы членов ряда, например, с помощью математических пакетов.
3. Предусмотреть замер времени вычисления суммы членов ряда. Число членов ряда *N* выбрать таким, чтобы время вычисления в последовательной программе было порядка 2- 5 сек.
4. На основе последовательной программы отладить **параллельную** программу вычисления суммы числового ряда. Использовать то же значение числа членов ряда *N,* что и в последовательной программе*.* Предусмотреть замер времени. Использовать для распараллеливания цикла директиву OpenMP: **#pragma omp forreduction *(+:<имя переменной суммирования членов ряда>)* private (<*имя переменной вычисления общего члена ряда*>).** Приэтом отдельные члены ряда (локальные переменные) будут вычисляться параллельно разными нитями, а сумма (как разделяемая переменная) будет обрабатываться корректно с использованием механизма **reduction.** Сравнить результаты и время вычисления последовательной и параллельной программ.
5. Запустить параллельную программу без опции **reduction *(+:<имя переменной суммирования членов ряда>)***. Оценить результат. Убедиться, что он некорректен. Убедиться, что от запуска к запуску результат меняется.
6. В программе из пункта 5 защитить от гонок общую переменную суммы с помощью директивы **#pragma omp atomic.** Оценить результат и время вычисления.
7. В программе из пункта 5 защитить от гонок общую переменную суммы с помощью директивы **#pragma omp critical.** Оценить результат и время вычисления.
8. В программе из пункта 5 защитить от гонок общую переменную суммы с помощью механизма замков (**lock**)**.** Оценить результат и время вычисления.
9. Выполнение примеров показать преподавателю.
10. Все выполненные задания оформить в виде отчета.

Задание на работу:



**Пункт 1 -5:**

Текст программы:

#include <stdio.h>

*const* int N = 100;

int main()

{

    double sum = 0;

    for (int n = 1; n < N; ++n)

    {

        double result = 3.0 / (10 \* n \* n - 2 \* n - 3);

        sum += result;

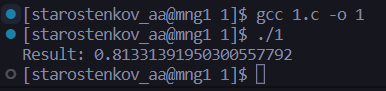
    }

    printf("Result: %.20f\n", sum);

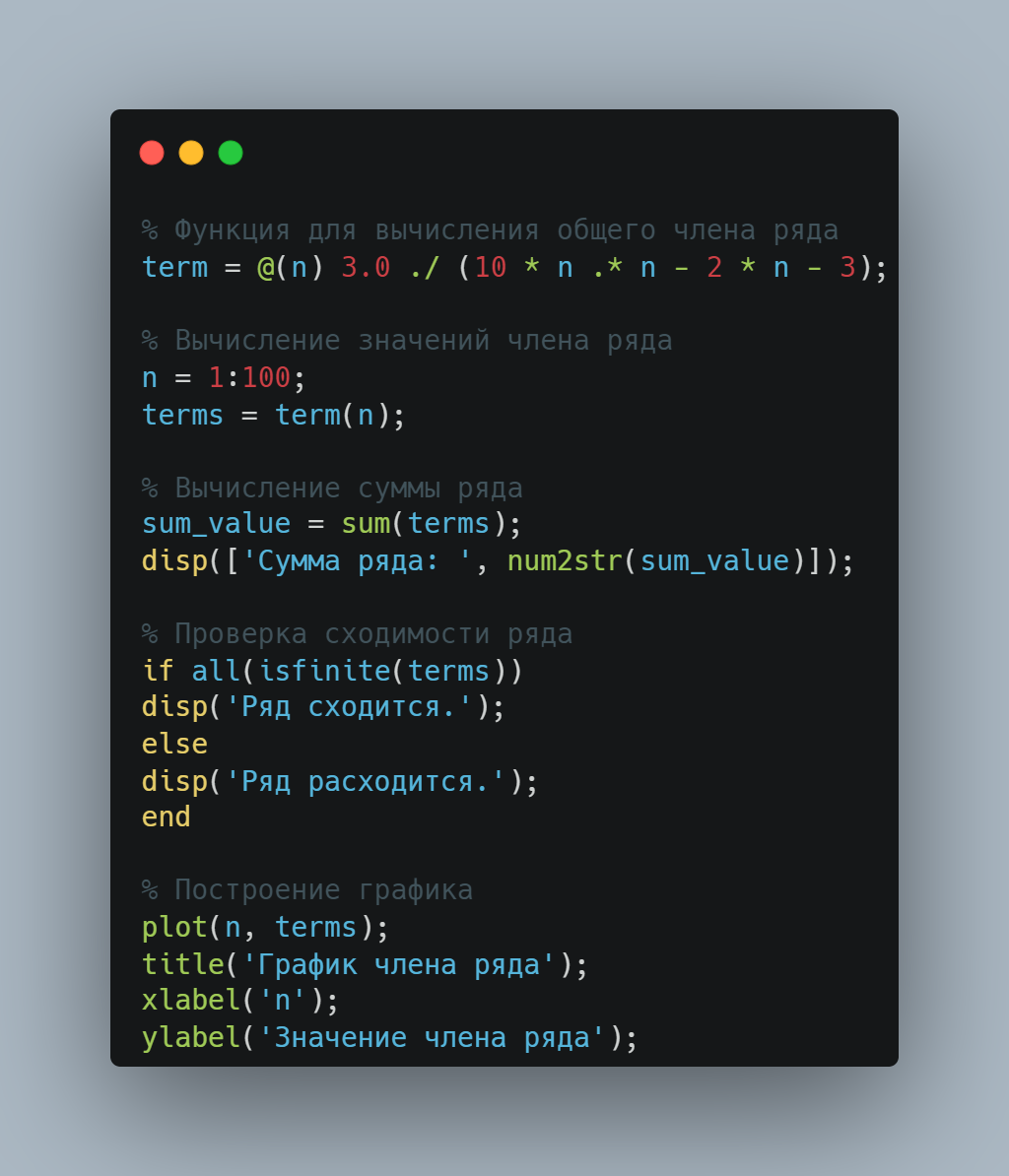
    return 0;

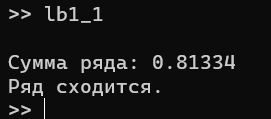
}

Результат выполнения программы:

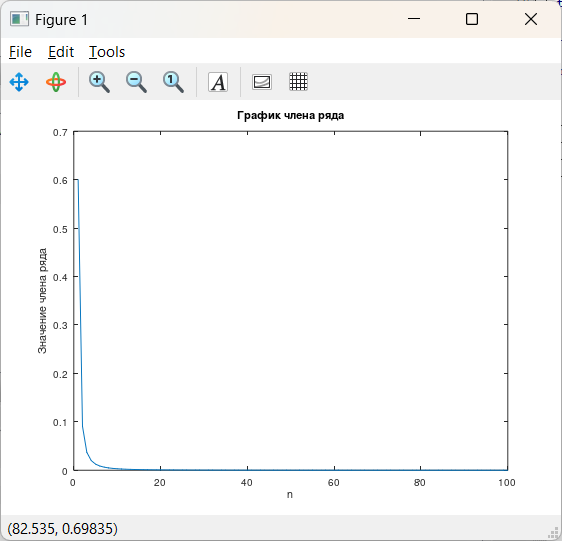


Проверка достоверности с помощью мат пакета (Octave 8.2.0):





Скорость сходимости ряда:



Пункт 3:

Предусмотреть замер времени вычисления суммы членов ряда. Число членов ряда *N* выбрать таким, чтобы время вычисления в последовательной программе было порядка 2- 5 сек.

**Программа :**

#include <stdio.h>

#include <omp.h>

*const* int N = 550000000;

int main() {

      double sum = 0;

    double start\_time = omp\_get\_wtime();

    for (int n = 0; n < N; ++n) {

    double result = 3.0 / ((10 \* n \* n) - (2 \* n) - 3);

        sum += result;

    }

    printf("Result: %.20f; Time: %.20f seconds;\n",

        sum,

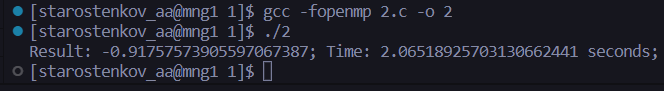
        omp\_get\_wtime() - start\_time

    );

    return 0;

}

**Результат работы программы:**

****

**Пункт 4**

На основе последовательной программы отладить **параллельную** программу вычисления суммы числового ряда. Использовать то же значение числа членов ряда *N,* что и в последовательной программе*.* Предусмотреть замер времени. Использовать для распараллеливания цикла директиву OpenMP: **#pragma omp forreduction *(+:<имя переменной суммирования членов ряда>)* private (<*имя переменной вычисления общего члена ряда*>).** Приэтом отдельные члены ряда (локальные переменные) будут вычисляться параллельно разными нитями, а сумма (как разделяемая переменная) будет обрабатываться корректно с использованием механизма **reduction.** Сравнить результаты и время вычисления последовательной и параллельной программ.

**Программа:**

#include <stdio.h>

#include <omp.h>

*const* int N = 550000000;

int main()

{

    double sum = 0;

    double start\_time = omp\_get\_wtime();

#pragma omp parallel for reduction(+ : sum)

    for (int n = 0; n < N; ++n)

    {

        double result = 3.0 / ((10 \* n \* n) - (2 \* n) - 3);

        sum += result;

    }

    printf("Result: %.20f; Time: %.20f seconds;\n",

           sum,

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

    return 0;

}

**Результат выполнения программы:**

****

Вывод: параллельная программа работает корректно. Время выполнения составляет ~0.208 секунды, против 3 секунд у последовательной программы. Ускорение составило ~14 раз. Результаты вычисления различаются начиная с восьмого знака после запятой, вероятно из-за различной очерёдности суммирования и погрешностей типа данных double.

**Пункт 5:**

Запустить параллельную программу без опции **reduction *(+:<имя переменной суммирования членов ряда>)***. Оценить результат. Убедиться, что он некорректен. Убедиться, что от запуска к запуску результат меняется.

**Программа:**

#include <stdio.h>

#include <stdio.h>

#include <omp.h>

*const* int N = 550000000;

int main()

{

    double sum = 0;

    double start\_time = omp\_get\_wtime();

#pragma omp parallel for

    for (int n = 0; n < N; ++n)

    {

        double result = 3.0 / ((10 \* n \* n) - (2 \* n) - 3);

        sum += result;

    }

    printf("Result: %.20f; Time: %.20f seconds;\n",

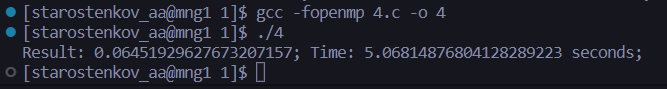
           sum,

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

    return 0;

}

**Результат выполнения программы**



Вывод: без опции reduction программа действительно работает некорректно и непредсказуемо. Переменная sum стала глобальной, потоки стали беспорядочно её перезаписывать, в том числе, «вклиниваясь» во время записи этой же переменной другими потоками. Извлечь полезных результатов из такой программы нельзя

**Пункт 6.**

В программе из пункта 5 защитить от гонок общую переменную суммы с помощью директивы **#pragma omp atomic.** Оценить результат и время вычисления.

**Программа**

#include <stdio.h>

#include <omp.h>

*const* int N = 550000000;

int main()

{

    double sum = 0;

    double start\_time = omp\_get\_wtime();

#pragma omp parallel for

    for (int n = 0; n < N; ++n)

    {

        double result = 3.0 / ((10 \* n \* n) - (2 \* n) - 3);

#pragma omp atomic

        sum += result;

    }

    printf("Result: %.20f; Time: %.20f seconds;\n",

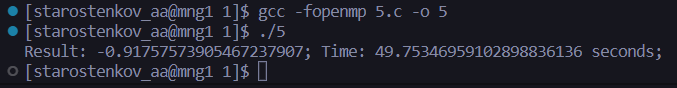
           sum,

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

    return 0;

}

**Результат выполнения программы**

****

Вывод: программа работает корректно, но стала значительно медленнее: ~49.7 секунд против ~0.208 секунд у программы с опцией reduction и 3 секунд у последовательной программы. Получившаяся программа работает даже медленнее последовательного варианта. Это происходит из-за больших накладных расходов на блокировки.

**Пункт 7:**

В программе из пункта 5 защитить от гонок общую переменную суммы с помощью директивы **#pragma omp critical.** Оценить результат и время вычисления.

**Программа:**

#include <stdio.h>

#include <omp.h>

*const* int N = 550000000;

int main()

{

    double sum = 0;

    double start\_time = omp\_get\_wtime();

#pragma omp parallel for

    for (int n = 0; n < N; ++n)

    {

        double result = 3.0 / ((10 \* n \* n) - (2 \* n) - 3);

#pragma omp critical

        sum += result;

    }

    printf("Result: %.20f; Time: %.20f seconds;\n",

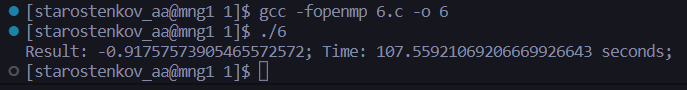
           sum,

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

    return 0;

}

**Результат выполнения программы:**

****

Вывод: Результат верный, но время выполнения 107 секунд – слишком большое.

**Пункт 8:**

В программе из пункта 5 защитить от гонок общую переменную суммы с помощью механизма замков (**lock**)**.** Оценить результат и время вычисления.

**Программа :**

#include <stdio.h>

#include <omp.h>

*const* int N = 550000000;

int main()

{

    double sum = 0;

    double start\_time = omp\_get\_wtime();

    omp\_lock\_t lock;

    omp\_init\_lock(&lock);

#pragma omp parallel for

    for (int n = 1; n < N; ++n)

    {

        double result = 3.0 / ((10 \* n \* n) - (2 \* n) - 3);

        omp\_set\_lock(&lock);

        sum += result;

        omp\_unset\_lock(&lock);

    }

    omp\_destroy\_lock(&lock);

    printf("Result: %.20f; Time: %.20f seconds;\n",

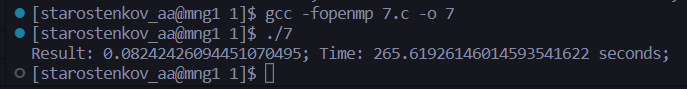
           sum,

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

    return 0;

}

**Результат работы программы:**

****

Вывод: результат некорректный, и программа с использованием замков выполняется почти 265 секунд.

Заключение: механизмы блокировки следует использовать аккуратно, иначе программа может стать невероятно медленной и работать медленнее последовательного варианта.