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: $\sum_{n=1}^{N} a_n$, где a_n - 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 for reduction *(+:<имя переменной суммирования членов ряда>)* private (<*имя переменной вычисления общего члена ряда*>).** При этом 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:

$$\frac{3}{10n^2 - 2n - 3}$$

Пункт 1 -5:

Текст программы:

```c
#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:

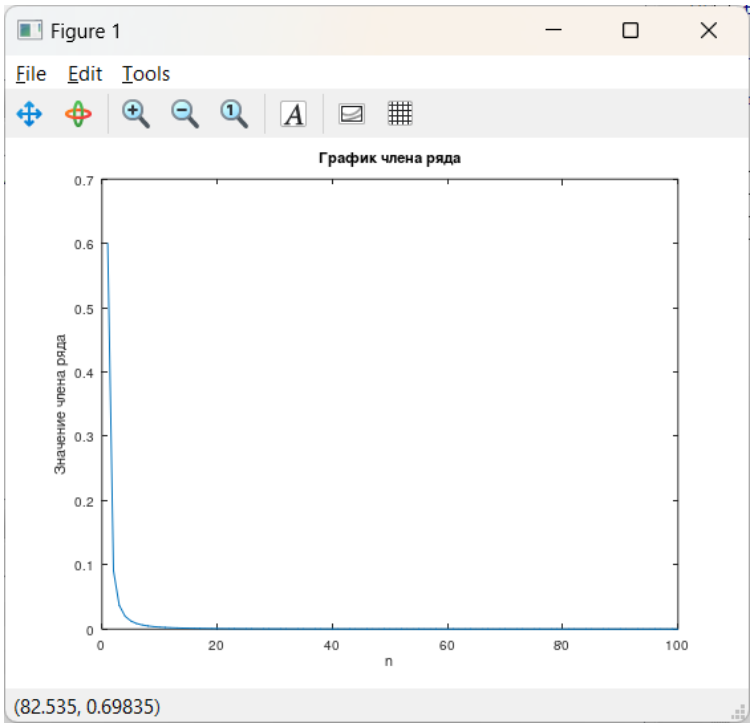
```
[starostenkov_aa@mng1 1]$ gcc 1.c -o 1
[starostenkov_aa@mng1 1]$ ./1
Result: 0.81331391950300557792
[starostenkov_aa@mng1 1]$
```

Проверка достоверности с помощью мат пакета (Octave 8.2.0):

```
% Функция для вычисления общего члена ряда
term = @(n) 3.0 ./ (10 * n .* n - 2 * n - 3);

% Вычисление значений члена ряда
n = 1:100;
terms = term(n);

% Вычисление суммы ряда
sum_value = sum(terms);
disp(['Сумма ряда: ', num2str(sum_value)]);

% Проверка сходимости ряда
if all(isfinite(terms))
disp('Ряд сходится.');
else
disp('Ряд расходится.');
end

% Построение графика
plot(n, terms);
title('График члена ряда');
xlabel('n');
ylabel('Значение члена ряда');
```

```
>> lb1_1

Сумма ряда: 0.81334
Ряд сходится.
>>
```

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 :

```c
#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:

```
[starostenkov_aa@mng1 1]$ gcc -fopenmp 2.c -o 2
[starostenkov_aa@mng1 1]$ ./2
Result: -0.91757573905597067387; Time: 2.06518925703130662441 seconds;
[starostenkov_aa@mng1 1]$
```

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 for reduction *(+:<имя переменной суммирования членов ряда>)* private (*<имя переменной вычисления общего члена ряда*>).** При этом 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:

```c
#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:

```
[starostenkov_aa@mng1 1]$ gcc -fopenmp 3.c -o 3
[starostenkov_aa@mng1 1]$ ./3
Result: -0.91757573905431200068; Time: 0.20897455816157162189 seconds;
[starostenkov_aa@mng1 1]$
```

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:

```c
#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

```
[starostenkov_aa@mng1 1]$ gcc -fopenmp 4.c -o 4
[starostenkov_aa@mng1 1]$ ./4
Result: 0.06451929627673207157; Time: 5.06814876804128289223 seconds;
[starostenkov_aa@mng1 1]$
```

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

```
[starostenkov_aa@mng1 1]$ gcc -fopenmp 5.c -o 5
[starostenkov_aa@mng1 1]$ ./5
Result: -0.91757573905467237907; Time: 49.753469591028988361 36 seconds;
[starostenkov_aa@mng1 1]$
```

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:

```
[starostenkov_aa@mng1 1]$ gcc -fopenmp 6.c -o 6
[starostenkov_aa@mng1 1]$ ./6
Result: -0.91757573905465572572; Time: 107.55921069206669926643 seconds;
[starostenkov_aa@mng1 1]$
```

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 :

```c
#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:

```
[starostenkov_aa@mng1 1]$ gcc -fopenmp 7.c -o 7
[starostenkov_aa@mng1 1]$ ./7
Result: 0.08242426094451070495; Time: 265.61926146014593541622 seconds;
[starostenkov_aa@mng1 1]$
```

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.