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 №2

«Параллельное выполнение циклов с использованием OpenMP»

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, реализующую циклическое вычисление общего члена ряда a_n (**без суммирования результатов!**), где n рассматривать как счетчик цикла. Варианты общего члена ряда выбрать из таблицы (как в лабораторной работе №1).
2. Предусмотреть замер времени выполнения цикла с использованием функции **omp_get_wtime ().**
3. Определить текущее значение переменной окружения **OMP_SCHEDULE.**
4. Отладить параллельную реализацию цикла с использованием директивы **#pragma omp for schedule (тип[, размер блока])** для следующих параметров, определяющих алгоритм распределения итераций между нитями: **static[, размер блока]; dynamic[, размер блока]; guided[, размер блока]; auto; runtime.**
5. Для анализа балансировки нитей при распараллеливании циклов использовать число нитей в диапазоне 5- 10. Число итераций цикла – в диапазоне 100- 200. Размер блока выбрать в диапазоне 1-4. Результаты представить в виде таблицы, указывающей количество итераций, выполненных каждой нитью в каждом алгоритме.
6. Сделать выводы о сбалансированности алгоритмов. Определить наиболее сбалансированные.

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

1. Предусмотреть замер времени выполнения цикла с использованием функции **omp_get_wtime ().**
2. Определить текущее значение переменной окружения **OMP_SCHEDULE.**

Код:

```c
#include <stdio.h>
#include <omp.h>
int main()
{
    omp_sched_t type;
    int chunk;
    omp_get_schedule(&type, &chunk);
    printf("schedule type: %i\n", type);
    printf("schedule chunk: %i\n", chunk);
    return 0;
}
```

Программа выводит тип переменной окружения и размер блока. Для получения более детальной информации о переменной окружения, была использована команда OMP_DISPLAY_ENV = true.

```
[starostenkov_aa@mng1 2]$ module load GCC
[starostenkov_aa@mng1 2]$ export OMP_DISPLAY_ENV=true
[starostenkov_aa@mng1 2]$ ./1

OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP = '201511'
  OMP_DYNAMIC = 'FALSE'
  OMP_NESTED = 'FALSE'
  OMP_NUM_THREADS = '21'
  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
schedule type: 2
schedule chunk: 1
[starostenkov_aa@mng1 2]$
```

Рисунок 1 – значение переменной окружения

3. Отладить параллельную реализацию цикла с использованием директивы **#pragma omp for schedule (тип[, размер блока])** для следующих параметров, определяющих алгоритм распределения итераций между нитями: **static[, размер блока]; dynamic[, размер блока]; guided[, размер блока]; auto;**

Листинг программ в приложении № 1.

4. Для анализа балансировки нитей при распараллеливании циклов использовать число нитей в диапазоне 5- 10. Число итераций цикла – в диапазоне 100- 200. Размер блока выбрать в диапазоне 1-4. Результаты представить в виде таблицы, указывающей количество итераций, выполненных каждой нитью в каждом алгоритме.

```
Введите количество значений для вычисления: 100
Введите количество потоков: 5
a1 = 0.600000
Tread 0 does iteration 1 on core 0
a3 = 0.037037
Tread 1 does iteration 3 on core 1
a4 = 0.020134
Tread 1 does iteration 4 on core 1
a13 = 0.001806
Tread 1 does iteration 13 on core 1
a14 = 0.001555
Tread 1 does iteration 14 on core 1
a23 = 0.000572
Tread 1 does iteration 23 on core 1
a24 = 0.000525
Tread 1 does iteration 24 on core 1
a33 = 0.000277
Tread 1 does iteration 33 on core 1
a34 = 0.000261
Tread 1 does iteration 34 on core 1
a43 = 0.000163
Tread 1 does iteration 43 on core 1
a44 = 0.000156
Tread 1 does iteration 44 on core 1
a53 = 0.000107
Tread 1 does iteration 53 on core 1
a54 = 0.000103
Tread 1 does iteration 54 on core 1
a63 = 0.000076
Tread 1 does iteration 63 on core 1
a64 = 0.000073
Tread 1 does iteration 64 on core 1
a73 = 0.000056
Tread 1 does iteration 73 on core 1
a74 = 0.000055
Tread 1 does iteration 74 on core 1
a83 = 0.000044
Tread 1 does iteration 83 on core 1
a84 = 0.000043
Tread 1 does iteration 84 on core 1
a93 = 0.000035
Tread 1 does iteration 93 on core 1
a94 = 0.000034
Tread 1 does iteration 94 on core 1
a2 = 0.090909
Tread 0 does iteration 2 on core 0
a11 = 0.002532
Tread 0 does iteration 11 on core 0
```

Рисунок 2 – Результат работы программы с типом Static и размером блока, равным 2

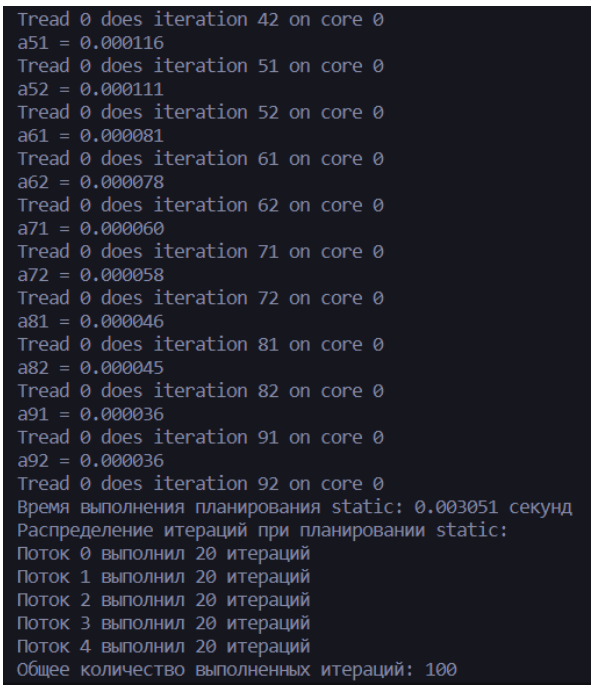

Рисунок 3 – Результат работы программы с типом Static и размером блока, равным 2 (продолжение рис. 2)

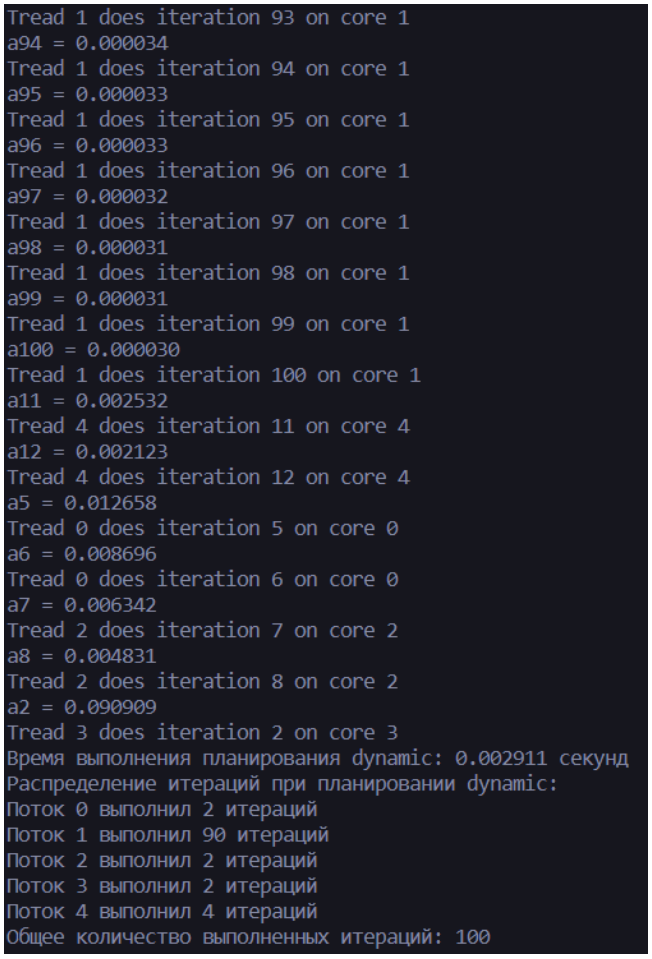

Рисунок 4 – Результат работы программы с типом Dynamic и размером блока, равным 2

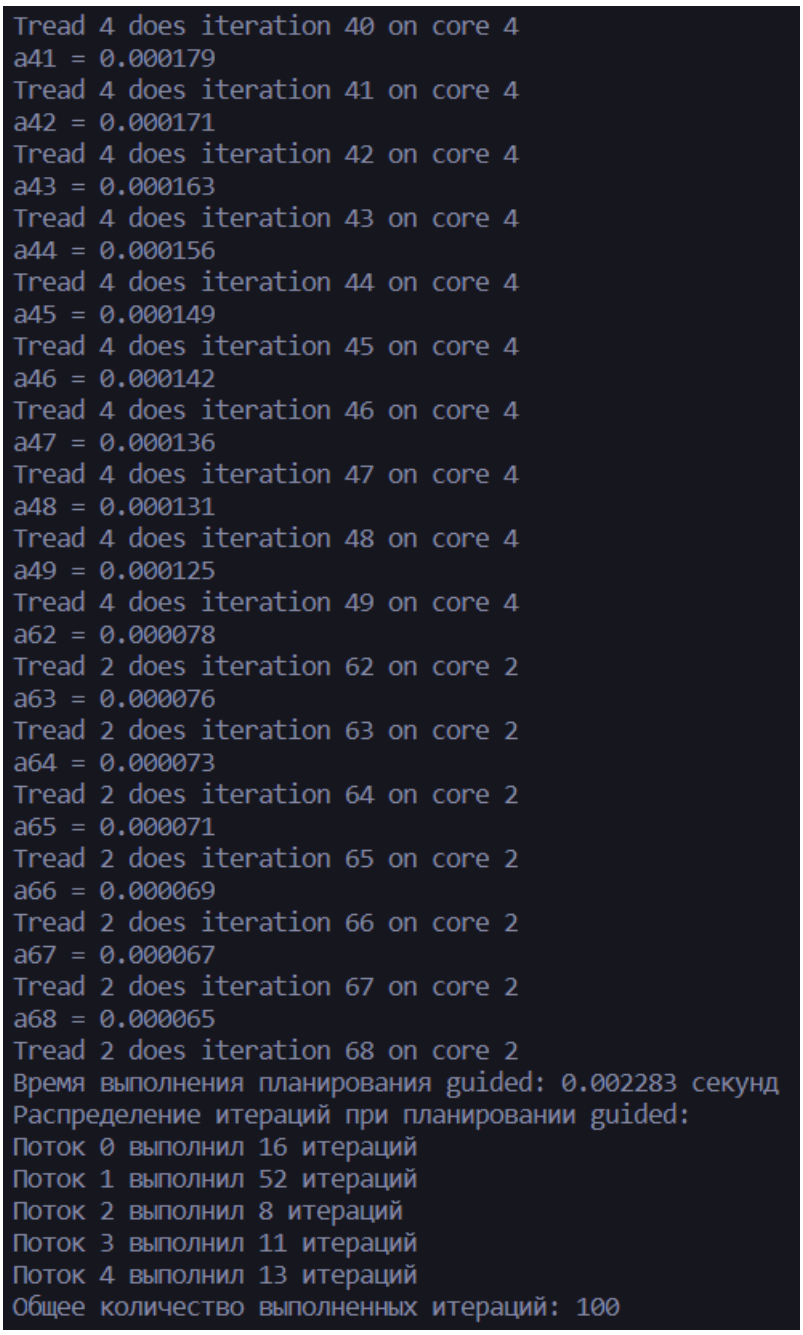

Рисунок 5 – Результат работы программы с типом Guided и размером блока, равным 2

Рисунок 6 – Результат работы программы с типом Auto

В таблице 1 приведены данные о равномерности распределения итераций в зависимости от типа алгоритма. Необходимо найти такие алгоритмы, при которых итерации между потоками будут распределяться наиболее равномерно, то есть, что бы потоки выполняли приблизительно одинаковое количество итераций

Таблица 1 – Равномерность распределения итераций

	Static	Dynamic	Guided	Auto
Thread 0	20	2	16	20
Thread 1	20	90	52	20
Thread2	20	2	8	20
Thread 3	20	2	11	20
Thread 4	20	4	13	20

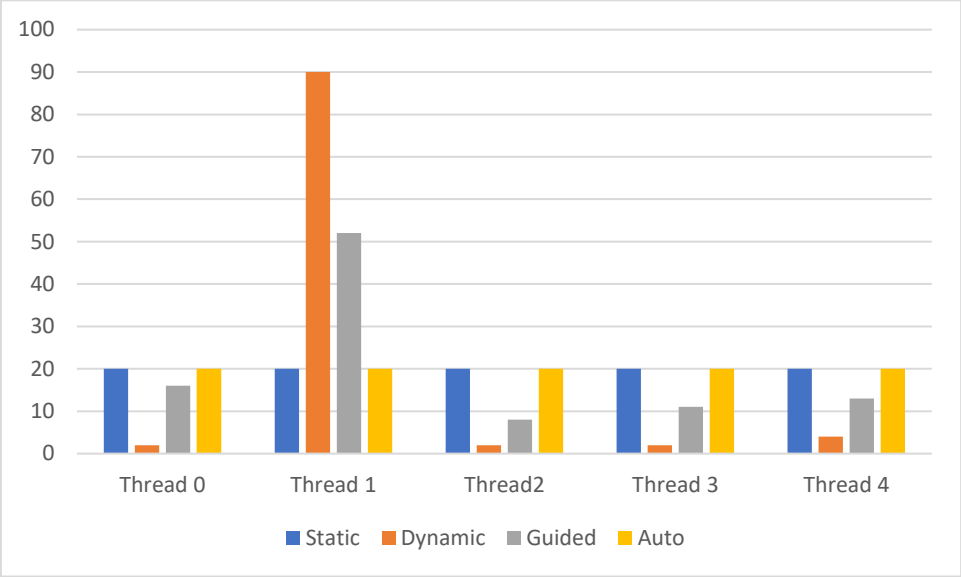

Рисунок 7 – График 1

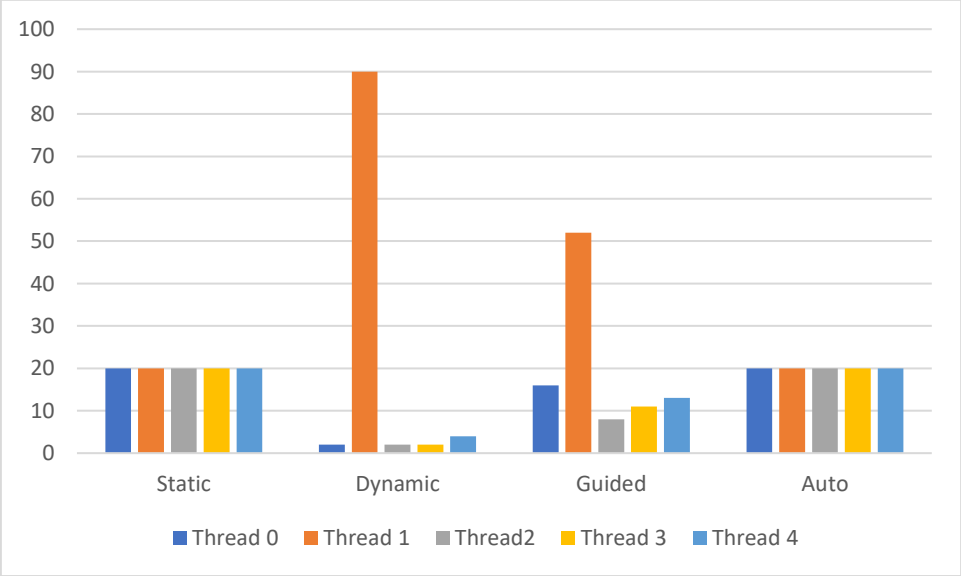

Рисунок 8 – График 2

Параметр static лучше всего распределяет итерации при размере блока = 2. При увеличении размера блока распределение несколько ухудшается. Параметры dynamic и guided при разных размерах блока так же плохо распределяют итерации по потокам.

Исходя из таблицы 1 и рисунков 7-8 можно сделать вывод о том, что директива schedule с параметром auto и static наиболее равномерно распределяет итерации по потокам и являются сбалансированными.

Приложение № 1. Код программы

```
// PROGRAM 5

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

float calculateSeriesTerm(int n)
{
    float result;
    int i;

    i = n;                                          // Используем n в качестве
значения счетчика i
    result = 3.0 / ((10 * n * n) - (2 * n) - 3); // Вычисляем общий член ряда

    return result;
}

int main()
{
    int n, i;
    int num_threads;
    int total_iterations = 0;
    int thread_iterations[32] = {0}; // Assuming up to 32 threads, adjust
accordingly
    double start_time, end_time;
    double static_time, dynamic_time, guided_time, auto_time;

    printf("Введите количество значений для вычисления: ");
    scanf("%d", &n);

    printf("Введите количество потоков: ");
    scanf("%d", &num_threads);

    start_time = omp_get_wtime(); // Замеряем время начала выполнения цикла

#pragma omp parallel num_threads(num_threads)
    {
        int thread_num = omp_get_thread_num();
#pragma omp for schedule(static, 2) private(i)
        for (i = 1; i <= n; i++)
        {
            float term = calculateSeriesTerm(i);
            printf("a%d = %f\n", i, term);
            printf("Tread %d does iteration %d on core %d\n", thread_num, i,
thread_num % num_threads);

// Use atomic to update thread_iterations
#pragma omp atomic
```

```
            thread_iterations[thread_num]++;

// Use a critical section to update total_iterations
#pragma omp critical
            total_iterations++;
        }
    }
    end_time = omp_get_wtime(); // Замеряем время окончания выполнения цикла
    static_time = end_time - start_time;
    printf("Время выполнения планирования static: %f секунд\n", static_time);
    printf("Распределение итераций при планировании static:\n");
    for (i = 0; i < num_threads; i++)
    {
        printf("Поток %d выполнил %d итераций\n", i, thread_iterations[i]);
    }

    printf("Общее количество выполненных итераций: %d\n", total_iterations);

    start_time = omp_get_wtime();
    total_iterations = 0;
    for (i = 0; i < num_threads; i++)
    {
        thread_iterations[i] = 0;
    }

#pragma omp parallel num_threads(num_threads)
    {
        int thread_num = omp_get_thread_num();
#pragma omp for schedule(dynamic, 2) private(i)
        for (i = 1; i <= n; i++)
        {
            float term = calculateSeriesTerm(i);
            printf("a%d = %f\n", i, term);
            printf("Tread %d does iteration %d on core %d\n", thread_num, i,
thread_num % num_threads);

// Use atomic to update thread_iterations
#pragma omp atomic
            thread_iterations[thread_num]++;

// Use a critical section to update total_iterations
#pragma omp critical
            total_iterations++;
        }
    }
    end_time = omp_get_wtime(); // Замеряем время окончания выполнения цикла
    dynamic_time = end_time - start_time;
    printf("Время выполнения планирования dynamic: %f секунд\n", dynamic_time);
    printf("Распределение итераций при планировании dynamic:\n");
    for (i = 0; i < num_threads; i++)
    {
```

```
        printf("Поток %d выполнил %d итераций\n", i, thread_iterations[i]);
    }

    printf("Общее количество выполненных итераций: %d\n", total_iterations);

    start_time = omp_get_wtime();
    total_iterations = 0;
    for (i = 0; i < num_threads; i++)
    {
        thread_iterations[i] = 0;
    }

#pragma omp parallel num_threads(num_threads)
    {
        int thread_num = omp_get_thread_num();
#pragma omp for schedule(guided, 2) private(i)
        for (i = 1; i <= n; i++)
        {
            float term = calculateSeriesTerm(i);
            printf("a%d = %f\n", i, term);
            printf("Tread %d does iteration %d on core %d\n", thread_num, i,
thread_num % num_threads);

// Use atomic to update thread_iterations
#pragma omp atomic
            thread_iterations[thread_num]++;

// Use a critical section to update total_iterations
#pragma omp critical
            total_iterations++;
        }
    }
    end_time = omp_get_wtime(); // Замеряем время окончания выполнения цикла
    guided_time = end_time - start_time;
    printf("Время выполнения планирования guided: %f секунд\n", guided_time);
    printf("Распределение итераций при планировании guided:\n");
    for (i = 0; i < num_threads; i++)
    {
        printf("Поток %d выполнил %d итераций\n", i, thread_iterations[i]);
    }

    printf("Общее количество выполненных итераций: %d\n", total_iterations);

    start_time = omp_get_wtime();
    total_iterations = 0;
    for (i = 0; i < num_threads; i++)
    {
        thread_iterations[i] = 0;
    }

#pragma omp parallel num_threads(num_threads)
```

```
    {
        int thread_num = omp_get_thread_num();
#pragma omp for schedule(auto) private(i)
        for (i = 1; i <= n; i++)
        {
            float term = calculateSeriesTerm(i);
            printf("a%d = %f\n", i, term);
            printf("Tread %d does iteration %d on core %d\n", thread_num, i,
thread_num % num_threads);

// Use atomic to update thread_iterations
#pragma omp atomic
            thread_iterations[thread_num]++;

// Use a critical section to update total_iterations
#pragma omp critical
            total_iterations++;
        }
    }
    end_time = omp_get_wtime(); // Замеряем время окончания выполнения цикла
    auto_time = end_time - start_time;
    printf("Время выполнения планирования auto: %f секунд\n", auto_time);
    printf("Распределение итераций при планировании auto:\n");
    for (i = 0; i < num_threads; i++)
    {
        printf("Поток %d выполнил %d итераций\n", i, thread_iterations[i]);
    }
    total_iterations = 0;
    for (i = 0; i < num_threads; i++)
    {
        thread_iterations[i] = 0;
    }

    printf("Общее количество выполненных итераций: %d\n", total_iterations);

    return 0;
}
```