**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**
2. Написать, отладить, скомпилировать и запустить на гибридном вычислительном кластере СФМЭИ программу, реализующую циклическое вычисление общего члена ряда  (**без суммирования результатов!**), где *n* рассматривать как счетчик цикла. Варианты общего члена ряда выбрать из таблицы (как в лабораторной работе №1).
3. Предусмотреть замер времени выполнения цикла с использованием функции **omp\_get\_wtime ().**
4. Определить текущее значение переменной окружения **OMP\_SCHEDULE.**
5. 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.**
6. 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.
7. Сделать выводы о сбалансированности алгоритмов. Определить наиболее сбалансированные.



1. Предусмотреть замер времени выполнения цикла с использованием функции **omp\_get\_wtime ().**
2. Определить текущее значение переменной окружения **OMP\_SCHEDULE.**

**Код:**

#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.

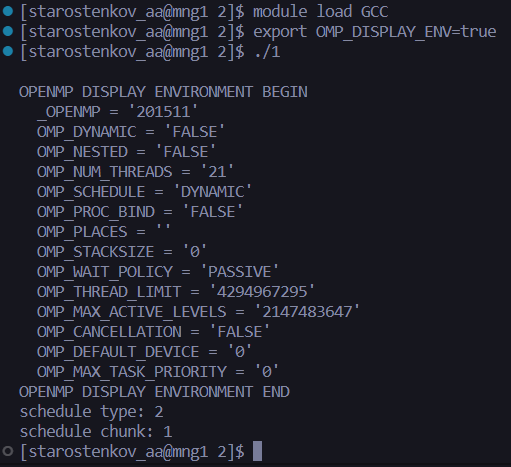


Рисунок 1 – значение переменной окружения

1. 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.*

1. 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.

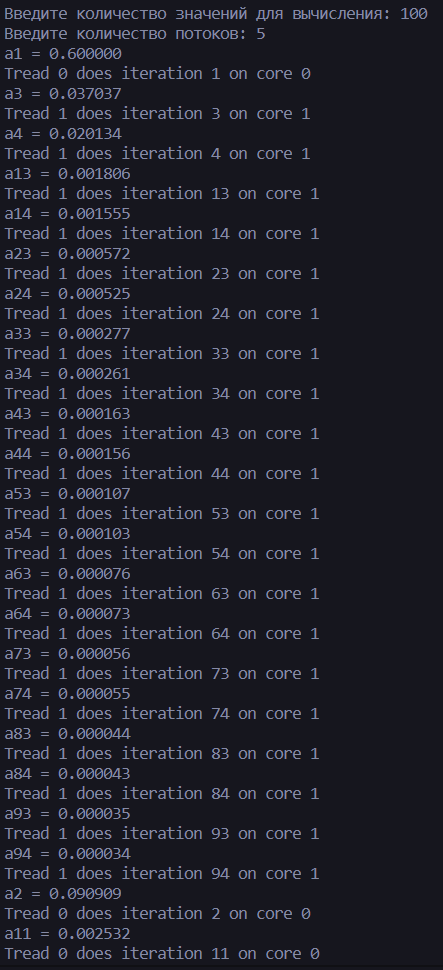


Рисунок 2 – Результат работы программы с типом Static и размером блока, равным 2

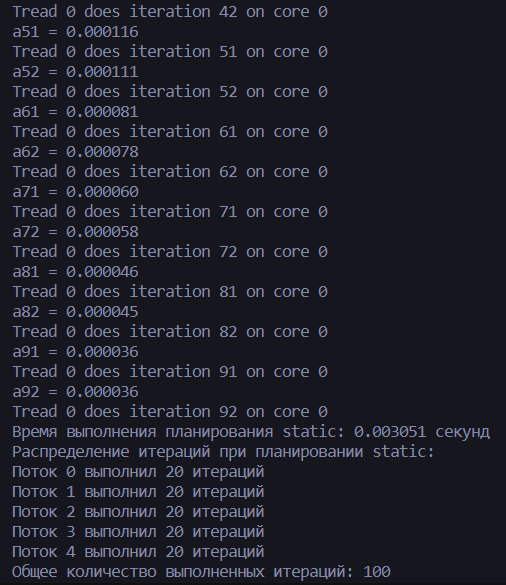


Рисунок 3 – Результат работы программы с типом Static и размером блока, равным 2 (продолжение рис. 2)

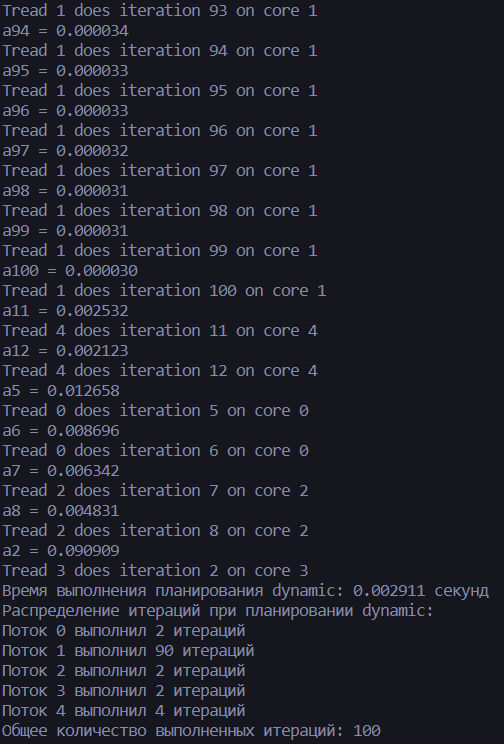


Рисунок 4 – Результат работы программы с типом Dynamic и размером блока, равным 2

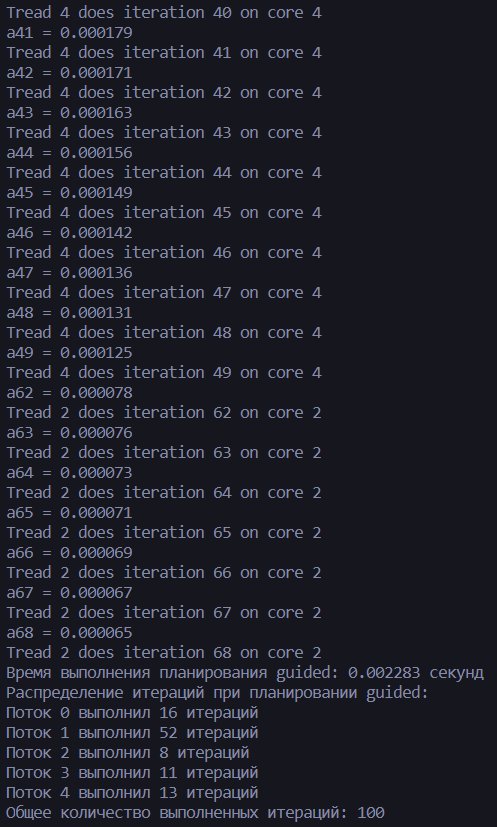
****

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;

}