МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

СЕРГИЕНКО ЛЕВ ЭДУАРДОВИЧ

Отчет по
Лабораторная работа 10

Технология OpenMP: поддержка шаблонов проектирования

Преподаватель

Кондратьева О.М.

2025

Шаблон SPMD

ТЕКСТ ПРОГРАММЫ

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

#define PAD 8 // Для устранения ложного разделения

void calculate_pi(long num_steps, int num_threads)
{
    double step;
    int i, nthreads;
    double pi = 0.0;
    double start_time, end_time;

    double (*sum)[PAD] = (double (*)[PAD])malloc(num_threads *
sizeof(double[PAD]));

    step = 1.0 / (double)num_steps;

    omp_set_num_threads(num_threads);

    start_time = omp_get_wtime();

#pragma omp parallel
    {
        int i, id, nthrds;
        double x;
        id = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0)
            nthreads = nthrds;

        for (i = id, sum[id][0] = 0.0; i < num_steps; i = i + nthrds)
        {
            x = (i + 0.5) * step;
            sum[id][0] += 4.0 / (1.0 + x * x);
        }
    }

    for (i = 0, pi = 0.0; i < nthreads; i++)
    {
        pi += sum[i][0] * step;
    }

    end_time = omp_get_wtime();

    printf("Threads: %d | Steps: %ld | Pi: %.10f | Time: %.6f seconds\n",
```

```
            num_threads, num_steps, pi, end_time - start_time);

    free(sum);
}

int main()
{
    long steps_array[] = {10000000, 100000000, 1000000000};
    int threads_array[] = {1, 2, 4, 8};
    int num_steps_count = sizeof(steps_array) / sizeof(steps_array[0]);
    int num_threads_count = sizeof(threads_array) / sizeof(threads_array[0]);

    for (int i = 0; i < num_steps_count; i++)
    {
        for (int j = 0; j < num_threads_count; j++)
        {
            calculate_pi(steps_array[i], threads_array[j]);
        }
        printf("-------------------------------------------------\n");
    }

    return 0;
}
```

Вычислительные эксперименты.

Размерность задачи	Время выполнения последовательной, с	2 потока			4 потока			8 потока		
		Время выполнения	Ускорение	Эффективность	Время выполнения	Ускорение	Эффективность	Время выполнения	Ускорение	Эффективность
10 000 000	0,04912	0,02475	1,98456	0,99228	0,01274	3,85451	0,96362	0,01999	2,45659	0,30707
100 000 000	0,48159	0,27006	1,78321	0,89160	0,12874	3,74075	0,93518	0,07307	6,59019	0,82377
1 000 000 000	3,45256	2,06442	1,67241	0,83620	1,06932	3,22872	0,80718	0,68759	5,02123	0,62765

Параллелизм циклов

ТЕКСТ ПРОГРАММЫ

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

```
#define PAD 8

void calculate_pi(long num_steps, int num_threads, int schedule_type)
{
    double step;
    double pi = 0.0;
    double start_time, end_time;

    double (*sum)[PAD] = (double (*)[PAD])malloc(num_threads *
sizeof(double[PAD]));
    for (int i = 0; i < num_threads; i++)
    {
        sum[i][0] = 0.0;
    }

    step = 1.0 / (double)num_steps;

    omp_set_num_threads(num_threads);

    start_time = omp_get_wtime();

    if (schedule_type == 0)
    {
// static scheduling
#pragma omp parallel for reduction(+ : pi) schedule(static)
        for (int i = 0; i < num_steps; i++)
        {
            double x = (i + 0.5) * step;
            pi += 4.0 / (1.0 + x * x);
        }
    }
    else if (schedule_type == 1)
    {
// dynamic scheduling
#pragma omp parallel for reduction(+ : pi) schedule(dynamic, 1000)
        for (int i = 0; i < num_steps; i++)
        {
            double x = (i + 0.5) * step;
            pi += 4.0 / (1.0 + x * x);
        }
    }
    else if (schedule_type == 2)
    {
// guided scheduling
#pragma omp parallel for reduction(+ : pi) schedule(guided, 1000)
        for (int i = 0; i < num_steps; i++)
```

```
        {
            double x = (i + 0.5) * step;
            pi += 4.0 / (1.0 + x * x);
        }
    }

    pi *= step;

    end_time = omp_get_wtime();

    const char *schedule_name;
    switch (schedule_type)
    {
    case 0:
        schedule_name = "static";
        break;
    case 1:
        schedule_name = "dynamic";
        break;
    case 2:
        schedule_name = "guided";
        break;
    }

    printf("Threads: %d | Steps: %ld | Schedule: %s | Pi: %.10f | Time:
%.6f seconds\n",
           num_threads, num_steps, schedule_name, pi, end_time -
start_time);

    free(sum);
}

int main()
{
    long steps_array[] = {10000000, 100000000, 1000000000};
    int threads_array[] = {1, 2, 4, 8};
    int schedule_types[] = {0, 1, 2}; // 0-static, 1-dynamic, 2-guided

    int num_steps_count = sizeof(steps_array) / sizeof(steps_array[0]);
    int num_threads_count = sizeof(threads_array) /
sizeof(threads_array[0]);
    int num_schedules = sizeof(schedule_types) /
sizeof(schedule_types[0]);

    for (int i = 0; i < num_steps_count; i++)
    {
```

```
        for (int j = 0; j < num_threads_count; j++)
        {
            for (int k = 0; k < num_schedules; k++)
            {
                calculate_pi(steps_array[i], threads_array[j],
schedule_types[k]);
            }

printf("-------------------------------------------------\n");
        }
    }

    return 0;
}
```

Вычислительные эксперименты.

Время выполнения последовательной, с	2 потока			4 потока			8 потока		
	Время выполнения	Ускорение	Эффективность	Время выполнения	Ускорение	Эффективность	Время выполнения	Ускорение	Эффективность
0,045346	0,023619	1,919895	0,959947	0,01336	3,394161	0,848540	0,008943	5,070557	0,633819
0,423734	0,174325	2,430712	1,215356	0,089967	4,709882	1,177470	0,053453	7,927225	0,990903
3,355625	1,811057	1,852854	0,926427	0,917441	3,657592	0,914398	0,480792	6,979369	0,87242

Разделяй и властвуй

ТЕКСТ ПРОГРАММЫ

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// Структура для хранения результатов вычислений
typedef struct
{
    double sum;
    long start;
    long end;
    double step;
} PiCalcTask;
```

```
double calculate_interval(long start, long end, double step)
{
    double sum = 0.0;
    for (long i = start; i < end; i++)
    {
        double x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }
    return sum;
}

// Рекурсивная функция "разделяй и властвуй" для вычисления π
double pi_divide_conquer(long start, long end, double step, int depth,
int cutoff_depth)
{
    if (depth >= cutoff_depth || (end - start) <= 10000)
    {
        return calculate_interval(start, end, step);
    }

    // делим интервал на две части
    long mid = start + (end - start) / 2;
    double left_sum, right_sum;

// Создаем задачи для левой и правой частей интервала
#pragma omp task shared(left_sum)
    left_sum = pi_divide_conquer(start, mid, step, depth + 1,
cutoff_depth);

#pragma omp task shared(right_sum)
    right_sum = pi_divide_conquer(mid, end, step, depth + 1,
cutoff_depth);

// Ждем завершения обеих задач
#pragma omp taskwait

    return left_sum + right_sum;
}

// Функция для запуска вычисления π
void calculate_pi_task(long num_steps, int num_threads, int
cutoff_depth)
{
    double step = 1.0 / (double)num_steps;
    double pi = 0.0;
```

```
    double start_time, end_time;

    omp_set_num_threads(num_threads);

    start_time = omp_get_wtime();

#pragma omp parallel
    {
#pragma omp single
        {
            pi = pi_divide_conquer(0, num_steps, step, 0, cutoff_depth)
* step;
        }
    }

    end_time = omp_get_wtime();

    printf("| %10ld | %8d | %8d | %16.10f | %12.6f |\n",
           num_steps, num_threads, cutoff_depth, pi, end_time -
start_time);
}

int main()
{
    long steps_array[] = {10000000, 100000000, 1000000000};
    int threads_array[] = {1, 2, 4, 8};
    int cutoff_depths[] = {32, 512, 512 * 4};

    int num_steps_count = sizeof(steps_array) / sizeof(steps_array[0]);
    int num_threads_count = sizeof(threads_array) /
sizeof(threads_array[0]);
    int num_depths = sizeof(cutoff_depths) / sizeof(cutoff_depths[0]);


printf("+------------+----------+----------+------------------+---------
-----+\n");
    printf("| Шаги       | Потоки   | Глубина  | Значение π       |
Время (сек)  |\n");

printf("+------------+----------+----------+------------------+---------
-----+\n");

    for (int i = 0; i < num_steps_count; i++)
    {
        for (int j = 0; j < num_threads_count; j++)
        {
```

```
            for (int k = 0; k < num_depths; k++)
            {
                calculate_pi_task(steps_array[i], threads_array[j],
cutoff_depths[k]);
            }
        }

printf("+------------+----------+----------+-----------------+---------
-----+\n");
    }

    return 0;
}
```

Вычислительные эксперименты.

Размерность задачи	Время выполнения последовательной, с	2 потока			4 потока			8 потока		
		Время выполнения	Ускорение	Эффективность	Время выполнения	Ускорение	Эффективность	Время выполнения	Ускорение	Эффективность
10 000 000	0,04687	0,02388	1,96261	0,98130	0,01816	2,58122	0,64530	0,01224	3,82684	0,47835
100 000 000	0,35068	0,17194	2,03957	1,01978	0,12526	2,79957	0,69989	0,07075	4,95627	0,61953
1 000 000 000	3,42642	1,83084	1,8714	0,93574	1,23477	2,77493	0,69373	0,56087	6,10907	0,76363

Сравнение результатов.

Основные выводы

1. **Параллелизм циклов (статический)** демонстрирует **наилучшее время** и **самую высокую эффективность** почти во всех случаях, поскольку цикл расчёта π равномерно сбалансирован.
2. **Шаблон SPMD** простой в реализации, но имеет дополнительные накладные расходы на явное управление массивом сумм и циклическое распределение итераций. Эффективность ниже при малой нагрузке (10 млн), но растёт с увеличением числа шагов.
3. **Разделяй и властвуй** даёт хорошее ускорение, но при малых глубинах рекурсии и мелком пороге (threshold) задач слишком много, что снижает эффективность. При больших объёмах работы (1 000

млн) масштабирование уже более конкурентоспособно, но всё же уступает статическому распараллеливанию.