Практикум по курсу

«Суперкомпьютеры и параллельная обработка данных»

Отчет о выполненном задании

студента 321 учебной группы
факультета ВМК МГУ

Теслюка Никиты Сергеевича

Лектор: доцент, к.ф.-м.н. Бахтин Владимир Александрович

Вариант 43

Москва, 2024г.

Содержание

Постановка задачи

Цель работы — разработка параллельной программы для решения задачи численного моделирования давления в задаче на уравнение Пуассона с использованием метода Якоби.

В рамках работы необходимо:

1. Реализовать параллельные программы с использованием OpenMP на языке программирования `C`:
    - с директивой `for` для распределения витков циклов;
    - с директивой `task` для механизма задач;
2. Реализовать параллельную программу с использованием MPI.
3. Исследовать эффективность программ.
    - Исследовать влияние различных опций оптимизации, которые поддерживаются компиляторами (`-O2`, `-O3`).
4. Исследовать масштабируемость программ: построить графики зависимости времени от числа ядер и объёма данных для разных версий программы.
5. Проанализировать причины ограниченной масштабируемости при максимальном числе ядер.

Описание последовательного алгоритма

Программа реализует итерационную схему для решения задачи давления методом Джакоби, используемого в решателе линейных уравнений для уравнения Пуассона давления в задаче о некомпрессируемом течении жидкости на языке C. Алгоритм работает на трехмерной сетке с разреженными коэффициентами.

Используемые функции

- `main()` — инициализация матриц, запуск итераций метода Джакоби и измерение производительности (время выполнения и операции с плавающей запятой).
- `initmt()` — инициализация матриц, задание начальных значений для элементов массива сетки и коэффициентов для вычисления давления и граничных условий.
- `jacobi(int nn)` — обновление значений давления и расчет сходимости на каждом шаге, вычисление ошибки (`gosa`), показателя сходимости.
- `second()` — измерение времени выполнения итераций. В будущем, при распараллеливании, будет использована функция `omp_get_wtime()`.

Используемые переменные

Программа использует следующие массивы:

- `imax, jmax, kmax` — переменные для хранения размеров решаемой области (по осям x, y, z).
- `omega` — параметр релаксации для метода Якоби (находится в диапазоне от 0 до 1).
- `NN` — количество итераций для выполнения в цикле метода Якоби.
- `cpu0, cpu1` — переменные для измерения времени выполнения программы.
- `gosa` — переменная для хранения значения ошибки (остаточного) в процессе вычислений.
- `p[MIMAX][MJMAX][MKMAX]` — трехмерный массив для хранения давления в сетке.
- `a[MIMAX][MJMAX][MKMAX][4]` — массив коэффициентов для уравнения Пуассона (размерность 4 для каждого элемента).
- `b[MIMAX][MJMAX][MKMAX][3]` — массив коэффициентов для уравнения Пуассона (размерность 3 для каждого элемента).
- `c[MIMAX][MJMAX][MKMAX][3]` — массив коэффициентов для уравнения Пуассона (размерность 3 для каждого элемента).
- `wrk1[MIMAX][MJMAX][MKMAX]` — рабочий массив для хранения промежуточных значений (источник члена уравнения Пуассона).
- `wrk2[MIMAX][MJMAX][MKMAX]` — рабочий массив для хранения результатов промежуточных вычислений.
- `bnd[MIMAX][MJMAX][MKMAX]` — массив для граничных условий (значения 0 или 1, определяющие, является ли точка на границе или нет).

	SMALL	MIDDLE	LARGE	EXTLARGE
MIMAX	65	129	257	513
MJMAX	33	65	129	257
MKMAX	33	65	129	257

Таблица 0. Размеры входных данных.

OpenMP

Директива `for`

Внесенные изменения

Для преобразования последовательной программы в параллельную с использованием директивы for OpenMP были внесены следующие изменения:

- Для параллелизации циклов использована директива `#pragma omp parallel for`, которая делегирует выполнение итераций цикла параллельным потокам.
- Директива `#pragma omp parallel` используется для создания параллельной области, позволяя нескольким потокам работать одновременно.
- Директива `collapse(3)` объединяет три вложенных цикла, увеличивая эффективность распределения работы между потоками.
- Для переменной `gosa` применена директива `reduction(+:gosa)`, которая создает локальные копии переменной для каждого потока. По завершении параллельных вычислений локальные значения складываются в глобальную переменную.
- Директива `schedule(static)` задает равномерное распределение работы между потоками до начала выполнения.
- `private(i, j, k)` гарантирует, что переменные индексов цикла `i`, `j`, `k` будут локальными для каждого потока, исключая конфликты между потоками.

Код программы

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

#define MIDDLE

#ifdef SMALL
#define MIMAX 65
#define MJMAX 33
#define MKMAX 33
#endif

#ifdef MIDDLE
#define MIMAX 129
#define MJMAX 65
#define MKMAX 65
#endif

#ifdef LARGE
#define MIMAX 257
#define MJMAX 129
#define MKMAX 129
#endif

#ifdef EXTLARGE
#define MIMAX 513
#define MJMAX 257
#define MKMAX 257
#endif

#define NN 200

static float p[MIMAX][MJMAX][MKMAX];
static float a[MIMAX][MJMAX][MKMAX][4],
             b[MIMAX][MJMAX][MKMAX][3], c[MIMAX][MJMAX][MKMAX][3];
static float bnd[MIMAX][MJMAX][MKMAX];
static float wrk1[MIMAX][MJMAX][MKMAX], wrk2[MIMAX][MJMAX][MKMAX];

static int imax, jmax, kmax;
static float omega;

void
initmt()
{
    int i, j, k;

#pragma omp parallel for collapse(3) private(i, j, k)
    for (i = 0; i < imax; ++i) {
        for (j = 0; j < jmax; ++j) {
            for (k = 0; k < kmax; ++k) {
                a[i][j][k][0] = 0.0;
                a[i][j][k][1] = 0.0;
                a[i][j][k][2] = 0.0;
                a[i][j][k][3] = 0.0;
                b[i][j][k][0] = 0.0;
                b[i][j][k][1] = 0.0;
                b[i][j][k][2] = 0.0;
                c[i][j][k][0] = 0.0;
                c[i][j][k][1] = 0.0;
                c[i][j][k][2] = 0.0;
                p[i][j][k] = 0.0;
```

```
                wrk1[i][j][k] = 0.0;
                bnd[i][j][k] = 0.0;
            }
        }
    }
#pragma omp parallel for collapse(3) private(i, j, k)
    for (i = 0; i < imax; ++i) {
        for (j = 0; j < jmax; ++j) {
            for (k = 0; k < kmax; ++k) {
                a[i][j][k][0] = 1.0;
                a[i][j][k][1] = 1.0;
                a[i][j][k][2] = 1.0;
                a[i][j][k][3] = 1.0 / 6.0;
                b[i][j][k][0] = 0.0;
                b[i][j][k][1] = 0.0;
                b[i][j][k][2] = 0.0;
                c[i][j][k][0] = 1.0;
                c[i][j][k][1] = 1.0;
                c[i][j][k][2] = 1.0;
                p[i][j][k] = (float) (k * k) /
                             (float) ((kmax - 1) * (kmax - 1));
                wrk1[i][j][k] = 0.0;
                bnd[i][j][k] = 1.0;
            }
        }
    }
}

float
jacobi(int nn)
{
    int i, j, k, n;
    float gosa, s0, ss;

    for (n = 0; n < nn; ++n) {
        gosa = 0.0;

#pragma omp parallel shared(a, b, c, p, wrk1, wrk2, bnd, gosa)
private(i, j, k, s0, ss)
        {
#pragma omp for schedule(static) collapse(3) reduction(+ : gosa)
            for (i = 1; i < imax - 1; ++i) {
                for (j = 1; j < jmax - 1; ++j) {
                    for (k = 1; k < kmax - 1; ++k) {
                        s0 = a[i][j][k][0] * p[i + 1][j][k] +
                             a[i][j][k][1] * p[i][j + 1][k] +
                             a[i][j][k][2] * p[i][j][k + 1] +
                             b[i][j][k][0] * (p[i + 1][j + 1][k] -
                                              p[i + 1][j - 1][k] -
                                              p[i - 1][j + 1][k] +
                                              p[i - 1][j - 1][k]) +
                             b[i][j][k][1] * (p[i][j + 1][k + 1] -
                                              p[i][j - 1][k + 1] -
                                              p[i][j + 1][k - 1] +
                                              p[i][j - 1][k - 1]) +
                             b[i][j][k][2] * (p[i + 1][j][k + 1] -
                                              p[i - 1][j][k + 1] -
                                              p[i + 1][j][k - 1] +
                                              p[i - 1][j][k - 1]) +
                             c[i][j][k][0] * p[i - 1][j][k] +
                             c[i][j][k][1] * p[i][j - 1][k] +
                             c[i][j][k][2] * p[i][j][k - 1] +
```

```
                                wrk1[i][j][k];


                            ss = (s0 * a[i][j][k][3] - p[i][j][k]) *
                                  bnd[i][j][k];

                            gosa = gosa + ss * ss;

                            wrk2[i][j][k] = p[i][j][k] + omega * ss;
                        }
                    }
                }

#pragma omp for schedule(static) collapse(3)
                for (i = 1; i < imax - 1; ++i) {
                    for (j = 1; j < jmax - 1; ++j) {
                        for (k = 1; k < kmax - 1; ++k) {
                            p[i][j][k] = wrk2[i][j][k];
                        }
                    }
                }
            }
        }

        return gosa;
}


int
main()
{
        int i, j, k;
        float gosa;
        double cpu0, cpu1, nflop, xmflops2, score;

        omega = 0.8;
        imax = MIMAX - 1;
        jmax = MJMAX - 1;
        kmax = MKMAX - 1;

        initmt();

        printf("mimax = %d mjmax = %d mkmax = %d\n", MIMAX, MJMAX, MKMAX);
        printf("imax = %d jmax = %d kmax =%d\n", imax, jmax, kmax);

        cpu0 = omp_get_wtime();

        gosa = jacobi(NN);

        cpu1 = omp_get_wtime();

        nflop = (kmax - 2) * (jmax - 2) * (imax - 2) * 34;

        if (cpu1 != 0.0) {
            xmflops2 = nflop / cpu1 * 1.0e-6 * (float) NN;
        }

        score = xmflops2 / 32.27;

        printf("cpu : %f sec.\n", cpu1 - cpu0);
        printf("Loop executed for %d times\n", NN);
        printf("Gosa : %e \n", gosa);
```

```
185     printf("MFLOPS measured : %f\n", xmflops2);
186     printf("Score based on MMX Pentium 200MHz : %f\n", score);
187 
188     return (0);
189 }
```

Тестирование программы на Polus

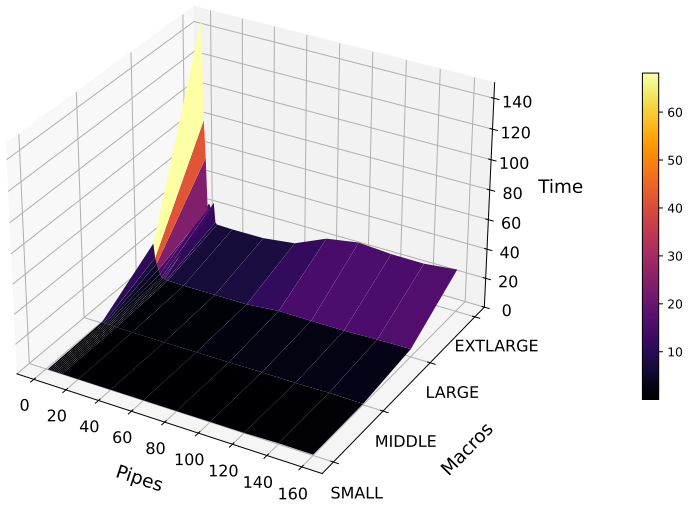

График 1.1

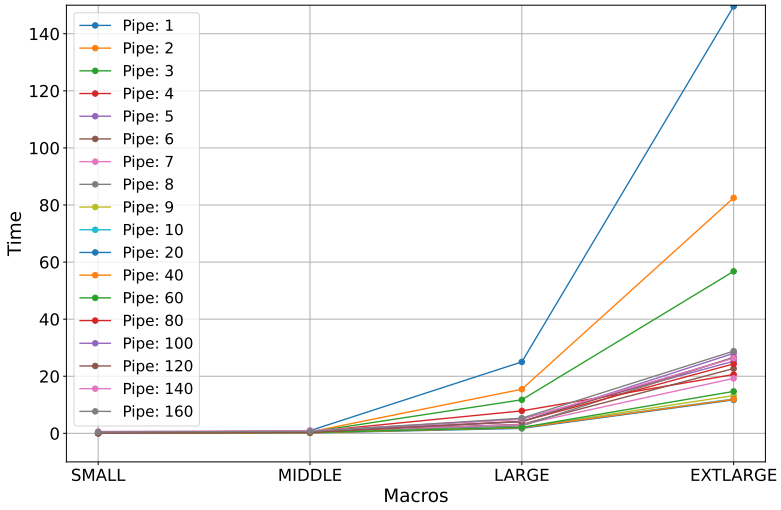

График 1.2

Графики 1.1, 1.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации директивой `for`.

	1	2	3	4	5	6	7	8	9
SMALL	0.11	0.07	0.06	0.06	0.08	0.08	0.05	0.06	0.05
MIDDLE	0.93	0.50	0.44	0.54	0.57	0.67	0.57	0.34	0.36
LARGE	25.03	15.44	11.76	7.87	5.08	3.15	3.04	2.59	2.01
EXTLARGE	149.6	82.5	56.76	20.61	25.27	22.69	19.25	26.63	13.21

	10	20	40	60	80	100	120	140	160
SMALL	0.07	0.03	0.05	0.07	0.09	0.14	0.12	0.65	0.50
MIDDLE	0.30	0.14	0.20	0.29	0.38	0.62	0.43	0.94	0.68
LARGE	1.80	1.74	1.95	2.05	4.18	4.06	4.15	4.91	5.31
EXTLARGE	12.02	11.72	11.90	14.69	24.32	28.01	26.45	26.32	28.81

Таблица 1. Результаты оптимизации директивой `for`. *Значения указаны в секундах и округлены до сотых.*

Тестирование программы на Polus. Флаг оптимизации `-O2`

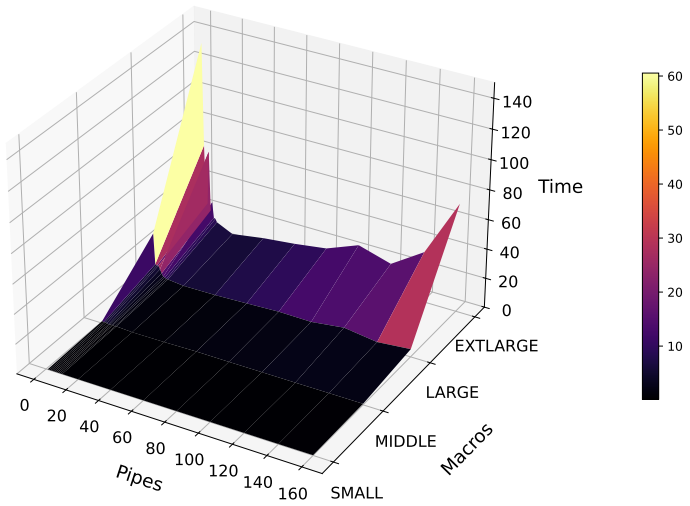

График 2.1

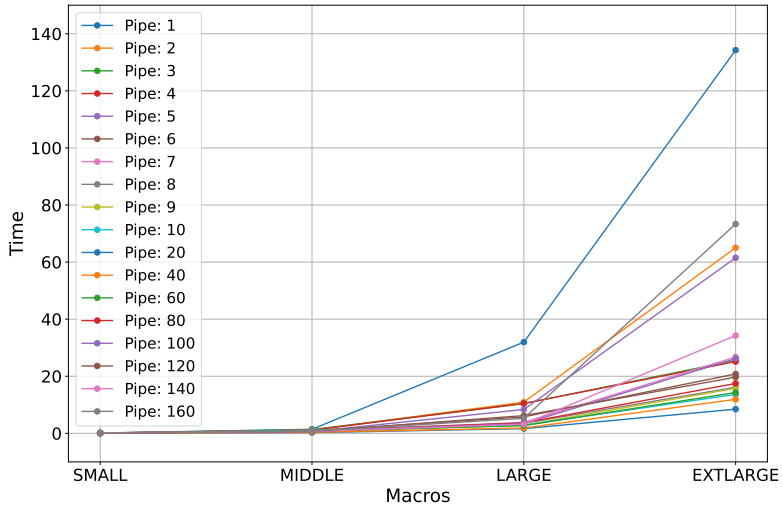

График 2.2

Графики 2.1, 2.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации директивой `for` с флагом оптимизации `-O2`.

	1	2	3	4	5	6	7	8	9
SMALL	0.11	0.10	0.08	0.09	0.09	0.08	0.09	0.05	0.06
MIDDLE	1.41	1.15	1.31	1.12	0.59	0.87	0.77	1.16	0.44
LARGE	31.95	10.93	10.43	10.45	8.37	5.77	3.82	3.53	2.84
EXTLARGE	134.27	65.04	25.57	25.14	61.51	20.80	26.74	16.14	15.77

	10	20	40	60	80	100	120	140	160
SMALL	0.06	0.03	0.04	0.07	0.09	0.12	0.13	0.13	0.18
MIDDLE	0.89	0.26	0.36	0.80	0.68	0.68	0.66	0.70	0.72
LARGE	1.64	1.86	2.71	3.67	3.38	6.21	3.19	5.26	2.85
EXTLARGE	8.46	11.91	14.30	17.48	26.18	19.69	34.25	73.35	16.14

Таблица 2. Результаты оптимизации директивой `for` с флагом оптимизации `-O2`. Значения указаны в секундах и округлены до сотых.

Тестирование программы на Polus. Флаг оптимизации `-O3`

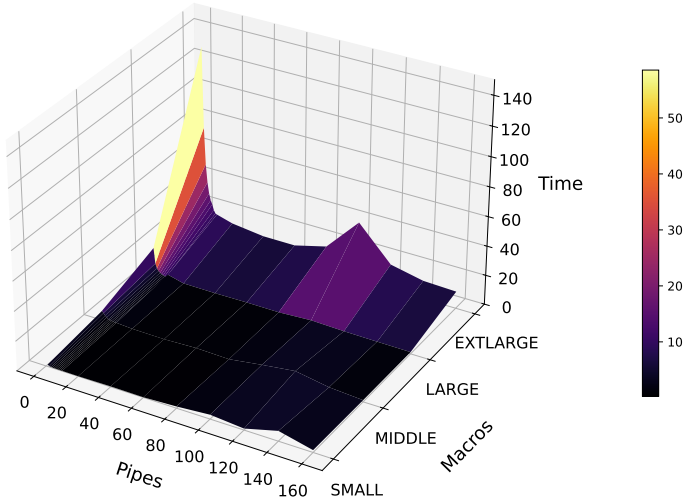

График 3.1

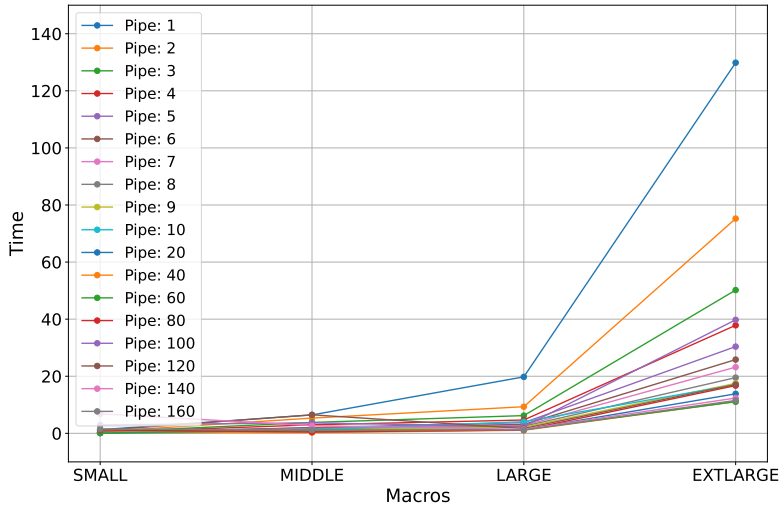

График 3.2

Графики 3.1,3.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации директивой `for` с флагом оптимизации `-O3`.

	1	2	3	4	5	6	7	8	9
SMALL	1.28	0.56	0.42	0.32	0.23	0.25	0.20	0.19	0.17
MIDDLE	6.44	5.34	3.82	2.99	2.00	1.93	1.65	1.54	1.24
LARGE	19.78	9.32	6.21	4.67	3.74	3.12	2.69	2.45	2.16
EXTLARGE	129.85	75.23	50.19	37.85	30.39	25.83	23.19	19.52	17.49

	10	20	40	60	80	100	120	140	160
SMALL	0.16	0.08	0.31	0.14	1.17	2.94	1.02	6.85	1.54
MIDDLE	1.31	0.55	0.26	1.09	0.46	3.66	6.49	2.67	1.14
LARGE	4.07	1.26	1.07	1.27	1.17	2.39	1.70	1.41	1.19
EXTLARGE	17.00	13.84	11.13	11.07	16.72	39.77	17.10	12.37	11.52

Таблица 3. Результаты оптимизации директивой `for` с флагом оптимизации `-O3`. Значения указаны в секундах и округлены до сотых.

Директива `task`

Внесенные изменения

Для распараллеливания программы с использованием директивы `task` в программу были внесены следующие изменения:

- Для параллелизации блоков кода используется директива `#pragma omp parallel`, которая создает параллельную область, позволяя нескольким потокам работать одновременно. В частности, параллелизация применяется к циклам, выполняющим вычисления.
- Внутри параллельных областей используются директивы `#pragma omp task` для параллельного выполнения отдельных итераций вложенных циклов. Это позволяет создать асинхронные задачи, которые выполняются одновременно в разных потоках.
- Для подсчета переменной `gosa` используется директива `atomic` для предотвращения гонок данных при добавлении локального значения к глобальной переменной. Это гарантирует корректную агрегацию результатов из разных потоков.
- Директива `firstprivate(...)` применяется к переменным индексов `i`, `j`, `k`, чтобы обеспечить их локальные копии для каждого потока. Это предотвращает возможные проблемы с синхронизацией при параллельной обработке этих переменных.
- Директива `#pragma omp taskwait` используется, чтобы гарантировать завершение всех асинхронных задач перед продолжением вычислений.
- Используется директива `shared` для массивов, которые должны быть доступны для всех потоков, и директива `private` для переменных, которые должны быть локальными для каждого потока.

Код программы

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

#define LARGE

#ifdef SMALL
#define MIMAX 65
#define MJMAX 33
#define MKMAX 33
#endif

#ifdef MIDDLE
#define MIMAX 129
#define MJMAX 65
#define MKMAX 65
#endif

#ifdef LARGE
#define MIMAX 257
#define MJMAX 129
#define MKMAX 129
#endif

#ifdef EXTLARGE
#define MIMAX 513
#define MJMAX 257
#define MKMAX 257
#endif

#define NN 200

static float a[MIMAX][MJMAX][MKMAX][4],
             b[MIMAX][MJMAX][MKMAX][3], c[MIMAX][MJMAX][MKMAX][3];
static float p[MIMAX][MJMAX][MKMAX];
static float wrk1[MIMAX][MJMAX][MKMAX], wrk2[MIMAX][MJMAX][MKMAX];
static float bnd[MIMAX][MJMAX][MKMAX];

static int imax, jmax, kmax;
static float omega;


void
initmt()
{
    int i, j, k;

    for (i = 0; i < imax; ++i) {
#pragma omp task firstprivate(i) private(j, k) shared(a, b, c, p, wrk1, bnd)
        {
            for (j = 0; j < jmax; ++j) {
                for (k = 0; k < kmax; ++k) {
                    a[i][j][k][0] = 0.0;
                    a[i][j][k][1] = 0.0;
                    a[i][j][k][2] = 0.0;
                    a[i][j][k][3] = 0.0;
                    b[i][j][k][0] = 0.0;
                    b[i][j][k][1] = 0.0;
                    b[i][j][k][2] = 0.0;
                    c[i][j][k][0] = 0.0;
                    c[i][j][k][1] = 0.0;
```

```
                    c[i][j][k][2] = 0.0;
                    p[i][j][k] = 0.0;
                    wrk1[i][j][k] = 0.0;
                    bnd[i][j][k] = 0.0;
                }
            }
        }
    }

    for (i = 0; i < imax; ++i) {
#pragma omp task firstprivate(i) private(j, k) shared(a, b, c, p, wrk1, bnd)
        {
            for (j = 0; j < jmax; ++j) {
                for (k = 0; k < kmax; ++k) {
                    a[i][j][k][0] = 1.0;
                    a[i][j][k][1] = 1.0;
                    a[i][j][k][2] = 1.0;
                    a[i][j][k][3] = 1.0 / 6.0;
                    b[i][j][k][0] = 0.0;
                    b[i][j][k][1] = 0.0;
                    b[i][j][k][2] = 0.0;
                    c[i][j][k][0] = 1.0;
                    c[i][j][k][1] = 1.0;
                    c[i][j][k][2] = 1.0;
                    p[i][j][k] = (float) (k * k) /
                                 (float) ((kmax - 1) * (kmax - 1));
                    wrk1[i][j][k] = 0.0;
                    bnd[i][j][k] = 1.0;
                }
            }
        }
    }
}

float
jacobi(int nn)
{
    int i, j, k, n;
    float gosa;
    float s0, ss;
    float local_gosa = 0.0;

#pragma omp parallel shared(a, b, c, p, wrk1, wrk2, bnd, omega, gosa)
    {
#pragma omp single
        {
            for (n = 0; n < nn; ++n) {
                gosa = 0.0;
                for (i = 1; i < imax - 1; ++i) {
#pragma omp task firstprivate(i) private(s0, ss, local_gosa)
                    {
                        local_gosa = 0.0;

                        for (j = 1; j < jmax - 1; ++j) {
                            for (k = 1; k < kmax - 1; ++k) {
                                s0 = a[i][j][k][0] * p[i + 1][j][k] +
                                     a[i][j][k][1] * p[i][j + 1][k] +
                                     a[i][j][k][2] * p[i][j][k + 1] +
                                     b[i][j][k][0] * (p[i + 1][j + 1][k] -
                                                      p[i + 1][j - 1][k] -
                                                      p[i - 1][j + 1][k] +
                                                      p[i - 1][j - 1][k]) +
```

```
                                        b[i][j][k][1] * (p[i][j + 1][k + 1] -
                                                         p[i][j - 1][k + 1] -
                                                         p[i][j + 1][k - 1] +
                                                         p[i][j - 1][k - 1]) +
                                        b[i][j][k][2] * (p[i + 1][j][k + 1] -
                                                         p[i - 1][j][k + 1] -
                                                         p[i + 1][j][k - 1] +
                                                         p[i - 1][j][k - 1]) +
                                        c[i][j][k][0] * p[i - 1][j][k] +
                                        c[i][j][k][1] * p[i][j - 1][k] +
                                        c[i][j][k][2] * p[i][j][k - 1] +
                                        wrk1[i][j][k];


                                    ss = (s0 * a[i][j][k][3] - p[i][j][k]) *
                                          bnd[i][j][k];
                                    local_gosa += ss * ss;

                                    wrk2[i][j][k] = p[i][j][k] + omega * ss;
                                }
                            }
#pragma omp atomic
                        gosa += local_gosa;
#pragma omp taskwait
                        }
                    }

                     for (i = 1; i < imax - 1; ++i) {
#pragma omp task firstprivate(i)
                        for (j = 1; j < jmax - 1; ++j) {
                            for (k = 1; k < kmax - 1; ++k)
                                p[i][j][k] = wrk2[i][j][k];
                        }
#pragma omp taskwait
                     }
                }
            }
        }

        return gosa;
}

int
main()
{
        int i, j, k;
        float gosa;
        double cpu0, cpu1, cpu2, cpu3, nflop, xmflops2, score;

        omega = 0.8;
        imax = MIMAX - 1;
        jmax = MJMAX - 1;
        kmax = MKMAX - 1;

        cpu2 = omp_get_wtime();

        initmt();

        cpu3 = omp_get_wtime();

        printf("mimax = %d mjmax = %d mkmax = %d\n", MIMAX, MJMAX, MKMAX);
        printf("imax = %d jmax = %d kmax =%d\n", imax, jmax, kmax);
```

```
185
186
187     cpu0 = omp_get_wtime();
188
189     gosa = jacobi(NN);
190
191     cpu1 = omp_get_wtime();
192
193     nflop = (kmax - 2) * (jmax - 2) * (imax - 2) * 34;
194
195     if (cpu1 != 0.0) {
196         xmflops2 = nflop / cpu1 * 1.0e-6 * (float) NN;
197     }
198
199     score = xmflops2 / 32.27;
200
201     printf("cpu : %f sec.\n", cpu1 - cpu0);
202     printf("init : %f sec.\n", cpu3 - cpu2);
203     printf("Loop executed for %d times\n", NN);
204     printf("Gosa : %e \n", gosa);
205     printf("MFLOPS measured : %f\n", xmflops2);
206     printf("Score based on MMX Pentium 200MHz : %f\n", score);
207     printf("\n");
208
209     return (0);
210 }
```

Тестирование программы на Polus

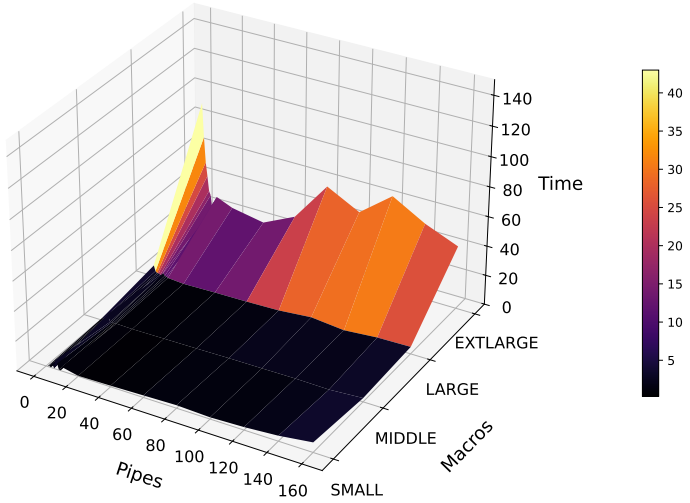

График 4.1

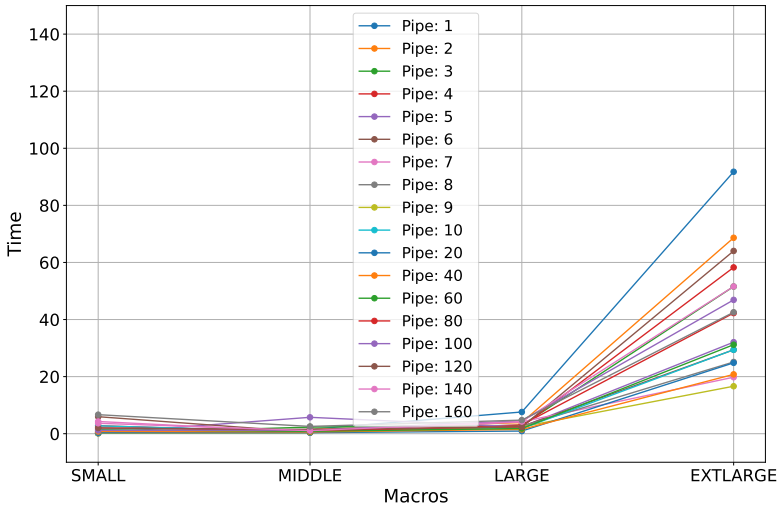

График 4.2

Графики 4.1, 4.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации директивой `task`.

	1	2	3	4	5	6	7	8	9
SMALL	0.21	0.79	0.14	2.8	0.18	5.96	4.28	0.24	0.2
MIDDLE	1.55	1.5	2.31	0.43	5.74	0.49	0.58	0.41	0.36
LARGE	7.61	3.91	2.9	3.1	2.37	2.51	4.16	2.59	2.42
EXTLARGE	91.77	68.64	51.56	42.26	32.07	29.4	19.79	25.13	16.63

	10	20	40	60	80	100	120	140	160
SMALL	2.76	0.34	0.78	1.4	2.02	1.18	2.01	3.66	6.67
MIDDLE	0.69	0.37	0.61	0.83	1.16	1.25	1.43	1.07	2.55
LARGE	0.93	1.5	1.82	2.45	4.5	2.36	4.27	4.76	3.91
EXTLARGE	24.83	20.79	31.18	58.26	46.89	64.03	51.58	42.55	42.26

Таблица 4. Результаты оптимизации директивой `task`. *Значения указаны в секундах и округлены до сотых.*

Тестирование программы на Polus. Флаг оптимизации `-O2`

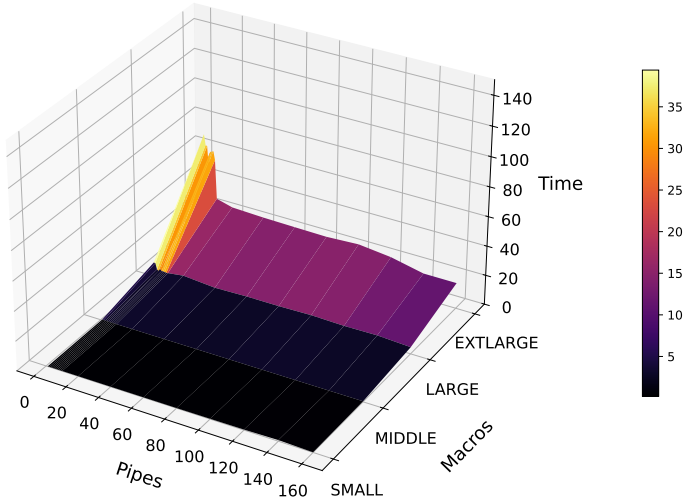

График 5.1

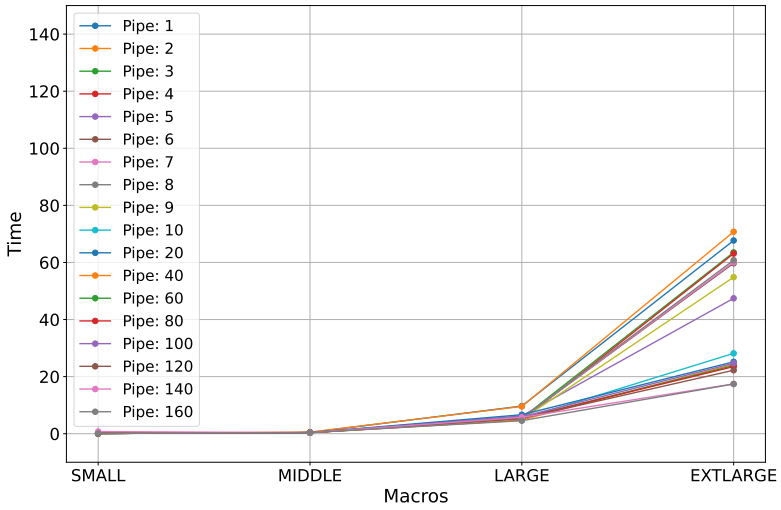

График 5.2

Графики 5.1, 5.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации директивой `task` с флагом оптимизации `-O2`.

	1	2	3	4	5	6	7	8	9
SMALL	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09	0.09
MIDDLE	0.43	0.55	0.37	0.37	0.37	0.37	0.38	0.37	0.37
LARGE	9.68	9.55	5.45	4.75	6.22	5.35	5.33	5.32	4.99
EXTLARGE	67.69	70.74	63.52	63.11	47.43	59.73	60.09	60.78	54.84

	10	20	40	60	80	100	120	140	160
SMALL	0.09	0.09	0.09	0.09	0.09	0.19	0.32	0.75	0.09
MIDDLE	0.37	0.37	0.37	0.38	0.37	0.37	0.37	0.37	0.38
LARGE	5.18	6.65	4.84	5.43	5.45	5.94	5.39	5.72	4.56
EXTLARGE	28.15	25.18	24.30	23.87	23.51	24.68	22.24	17.44	17.41

Таблица 5. Результаты оптимизации директивой `task` с флагом оптимизации `-O2`. Значения указаны в секундах и округлены до сотых.

Тестирование программы на Polus. Флаг оптимизации -O3

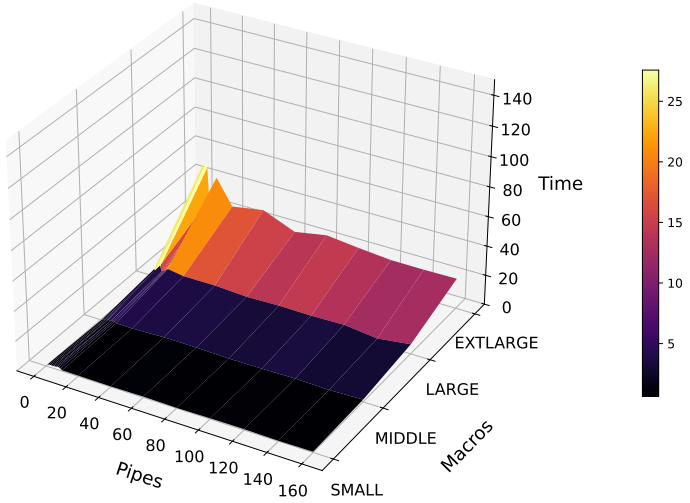

График 6.1

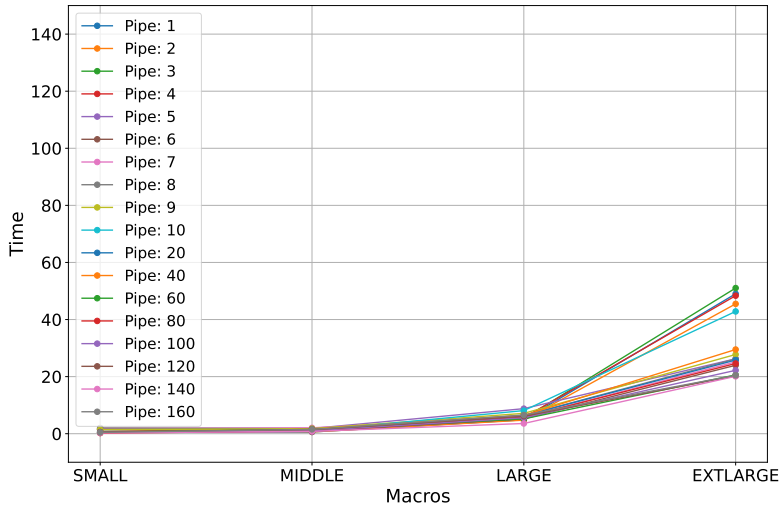

График 6.2

Графики 6.1, 6.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации директивой `task` с флагом оптимизации `-O3`.

	1	2	3	4	5	6	7	8	9
SMALL	1.43	1.56	1.66	1.49	1.73	1.7	1.95	2.0	1.41
MIDDLE	0.6	0.62	0.96	1.08	1.84	0.92	2.06	1.74	1.76
LARGE	4.86	4.75	5.3	5.61	8.77	5.2	5.81	6.38	7.13
EXTLARGE	49.02	45.48	51.03	48.35	26.24	23.97	25.17	26.19	27.75

	10	20	40	60	80	100	120	140	160
SMALL	0.53	0.47	0.3	0.55	0.23	0.52	0.61	0.28	0.45
MIDDLE	1.03	0.74	1.57	1.5	0.87	1.03	1.39	0.81	1.51
LARGE	6.33	6.41	5.2	5.8	5.58	6.13	3.58	6.55	8.1
EXTLARGE	25.8	29.49	20.59	24.59	22.2	20.24	20.11	20.45	42.83

Таблица 6. Результаты оптимизации директивой `task` с флагом оптимизации `-O3`. Значения указаны в секундах и округлены до сотых.

Выводы

После реализации программ и проведения тестов можно сделать следующе выводы:

- **Малые (`SMALL`) и средние (`MIDDLE`) наборы данных:** Для малых и средних наборов данных директива `task` показывает лучшую производительность. Например, для набора данных `MIDDLE` при одном потоке директива `task` работает быстрее, чем `for`. Это указывает на то, что `task` более эффективно распределяет задачи между потоками, особенно когда задачи относительно малы.
- **Большие (`LARGE`) и экстремально большие (`EXTLARGE`) наборы данных:** Для больших данных различия между `for` и `task` становятся менее выраженными. Для набора "`LARGE`"при большем числе потоков директива `for` работает быстрее, чем `task`. Это может свидетельствовать о том, что для больших данных дополнительная нагрузка на создание и синхронизацию задач в `task` становится более значимой.

Кроме того, выполнение программы не всегда ускоряется с увеличением числа потоков. Например, при использовании директивы `for`, для экстремально больших наборов данных оптимальное число потоков находится в диапазоне от 60 до 80. При увеличении числа потоков время выполнения перестает уменьшаться из-за накладных расходов на создание и синхронизацию дополнительных потоков. Особенно для небольших данных использование слишком большого числа потоков может замедлить выполнение программы, поскольку накладные расходы на создание потоков становятся значительными.

Влияние флагов компиляции

- `-O2`: Применение флага `-O2` значительно уменьшает время выполнения программы по сравнению с базовой версией. Это связано с улучшением работы с кэшированием данных и оптимизацией циклов.
- `-O3`: Флаг `-O3` ещё сильнее оптимизирует программу, что особенно заметно при использовании директивы `for` для большого числа потоков на больших наборах данных. Однако для некоторых случаев чрезмерная агрессивность оптимизаций может привести к ухудшению производительности из-за дополнительных вычислений, которые не всегда оправданы.

Приложения

- Прилагаемые файлы OpenMP:
  1. Файл `for.c` — реазация параллелизма при помощи директивы `for`
  2. Файл `task.c` — реазация параллелизма при помощи директивы `task`