

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	2
Описание исходного алгоритма	3
Используемые функции	3
Используемые переменные	3
OpenMP	4
Параллелизация при помощи директивы for	4
Внесенные изменения	4
Код программы	5
Тестирование программы на Polus	8
Тестирование программы на Polus. Флаг оптимизации -O2	9
Тестирование программы на Polus. Флаг оптимизации -O3	10
Параллелизация при помощи директивы task	11
Внесенные изменения	11
Код программы	12
Тестирование программы на Polus	16
Тестирование программы на Polus. Флаг оптимизации -O2	17
Тестирование программы на Polus. Флаг оптимизации -O3	18
Выводы об OpenMP	19
Влияние флагов компиляции	19
MPI	20
Параллелизация при помощи MPI	20
Внесенные изменения	20
Код программы	21
Тестирование программы на Polus	25
Тестирование программы на Polus. Флаг оптимизации -O2	26
Тестирование программы на Polus. Флаг оптимизации -O3	27
Выводы о MPI	28
Влияние флагов компиляции	28
Выводы	29
Приложения	30

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 Якоби, используемого в решателе линейных уравнений для уравнения Пуассона давления в задаче о некомпresseруемом течении жидкости на языке C. Алгоритм работает на трехмерной сетке с разреженными коэффициентами.

Используемые функции

- `main()` — инициализация матриц, запуск итераций метода Якоби 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()` в случае использования OMP и `MPI_Wtime()` в случае использования MPI.

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

Параллелизация при помощи директивы `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

```

1  #include <stdio.h>
2  #include <omp.h>
3
4  #define MIDDLE
5
6  #ifdef SMALL
7  #define MIMAX 65
8  #define MJMAX 33
9  #define MKMAX 33
10 #endif
11
12 #ifdef MIDDLE
13 #define MIMAX 129
14 #define MJMAX 65
15 #define MKMAX 65
16 #endif
17
18 #ifdef LARGE
19 #define MIMAX 257
20 #define MJMAX 129
21 #define MKMAX 129
22 #endif
23
24 #ifdef EXTLARGE
25 #define MIMAX 513
26 #define MJMAX 257
27 #define MKMAX 257
28 #endif
29
30 #define NN 200
31
32 static float p[MIMAX][MJMAX][MKMAX];
33 static float a[MIMAX][MJMAX][MKMAX][4],
34             b[MIMAX][MJMAX][MKMAX][3], c[MIMAX][MJMAX][MKMAX][3];
35 static float bnd[MIMAX][MJMAX][MKMAX];
36 static float wrk1[MIMAX][MJMAX][MKMAX], wrk2[MIMAX][MJMAX][MKMAX];
37
38 static int imax, jmax, kmax;
39 static float omega;
40
41 void
42 initmt()
43 {
44     int i, j, k;
45
46 #pragma omp parallel for collapse(3) private(i, j, k)
47     for (i = 0; i < imax; ++i) {
48         for (j = 0; j < jmax; ++j) {
49             for (k = 0; k < kmax; ++k) {
50                 a[i][j][k][0] = 0.0;
51                 a[i][j][k][1] = 0.0;
52                 a[i][j][k][2] = 0.0;
53                 a[i][j][k][3] = 0.0;
54                 b[i][j][k][0] = 0.0;
55                 b[i][j][k][1] = 0.0;
56                 b[i][j][k][2] = 0.0;
57                 c[i][j][k][0] = 0.0;
58                 c[i][j][k][1] = 0.0;
59                 c[i][j][k][2] = 0.0;
60                 p[i][j][k] = 0.0;

```

```

61         wrk1[i][j][k] = 0.0;
62         bnd[i][j][k] = 0.0;
63     }
64 }
65 }
66 #pragma omp parallel for collapse(3) private(i, j, k)
67 for (i = 0; i < imax; ++i) {
68     for (j = 0; j < jmax; ++j) {
69         for (k = 0; k < kmax; ++k) {
70             a[i][j][k][0] = 1.0;
71             a[i][j][k][1] = 1.0;
72             a[i][j][k][2] = 1.0;
73             a[i][j][k][3] = 1.0 / 6.0;
74             c[i][j][k][0] = 1.0;
75             c[i][j][k][1] = 1.0;
76             c[i][j][k][2] = 1.0;
77             p[i][j][k] = (float) (k * k) /
78                 (float) ((kmax - 1) * (kmax - 1));
79             wrk1[i][j][k] = 0.0;
80             bnd[i][j][k] = 1.0;
81         }
82     }
83 }
84 }
85
86 float
87 jacobi(int nn)
88 {
89     int i, j, k, n;
90     float gosa, s0, ss;
91
92     for (n = 0; n < nn; ++n) {
93         gosa = 0.0;
94
95         #pragma omp parallel shared(a, b, c, p, wrk1, wrk2, bnd, gosa)
96         private(i, j, k, s0, ss)
97         {
98             #pragma omp for schedule(static) collapse(3) reduction(+ : gosa)
99             for (i = 1; i < imax - 1; ++i) {
100                 for (j = 1; j < jmax - 1; ++j) {
101                     for (k = 1; k < kmax - 1; ++k) {
102                         s0 = a[i][j][k][0] * p[i + 1][j][k] +
103                             a[i][j][k][1] * p[i][j + 1][k] +
104                             a[i][j][k][2] * p[i][j][k + 1] +
105                             b[i][j][k][0] * (p[i + 1][j + 1][k] -
106                                 p[i + 1][j - 1][k] -
107                                 p[i - 1][j + 1][k] +
108                                 p[i - 1][j - 1][k]) +
109                             b[i][j][k][1] * (p[i][j + 1][k + 1] -
110                                 p[i][j - 1][k + 1] -
111                                 p[i][j + 1][k - 1] +
112                                 p[i][j - 1][k - 1]) +
113                             b[i][j][k][2] * (p[i + 1][j][k + 1] -
114                                 p[i - 1][j][k + 1] -
115                                 p[i + 1][j][k - 1] +
116                                 p[i - 1][j][k - 1]) +
117                             c[i][j][k][0] * p[i - 1][j][k] +
118                             c[i][j][k][1] * p[i][j - 1][k] +
119                             c[i][j][k][2] * p[i][j][k - 1] +
120                             wrk1[i][j][k];
121                     }
122                 }

```

```

123         ss = (s0 * a[i][j][k][3] - p[i][j][k]) *
124             bnd[i][j][k];
125
126         gosa = gosa + ss * ss;
127
128         wrk2[i][j][k] = p[i][j][k] + omega * ss;
129     }
130 }
131 }
132
133 #pragma omp for schedule(static) collapse(3)
134 for (i = 1; i < imax - 1; ++i) {
135     for (j = 1; j < jmax - 1; ++j) {
136         for (k = 1; k < kmax - 1; ++k) {
137             p[i][j][k] = wrk2[i][j][k];
138         }
139     }
140 }
141 }
142 }
143 return gosa;
144 }
145
146
147 int
148 main()
149 {
150     int i, j, k;
151     float gosa;
152     double cpu0, cpu1, nflop, xmflops2, score;
153     omega = 0.8;
154     imax = MIMAX - 1;
155     jmax = MJMAX - 1;
156     kmax = MKMAX - 1;
157
158     initmt();
159
160     printf("mimax = %d mjmax = %d mkmax = %d\n", MIMAX, MJMAX, MKMAX);
161     printf("imax = %d jmax = %d kmax = %d\n", imax, jmax, kmax);
162
163     cpu0 = omp_get_wtime();
164
165     gosa = jacobi(NN);
166
167     cpu1 = omp_get_wtime();
168
169     nflop = (kmax - 2) * (jmax - 2) * (imax - 2) * 34;
170
171     if (cpu1 != 0.0) {
172         xmflops2 = nflop / cpu1 * 1.0e-6 * (float) NN;
173     }
174
175     score = xmflops2 / 32.27;
176
177     printf("cpu : %f sec.\n", cpu1 - cpu0);
178     printf("Loop executed for %d times\n", NN);
179     printf("Gosa : %e \n", gosa);
180     printf("MFLOPS measured : %f\n", xmflops2);
181     printf("Score based on MMX Pentium 200MHz : %f\n", score);
182
183     return (0);
184 }

```


Тестирование программы на Polus

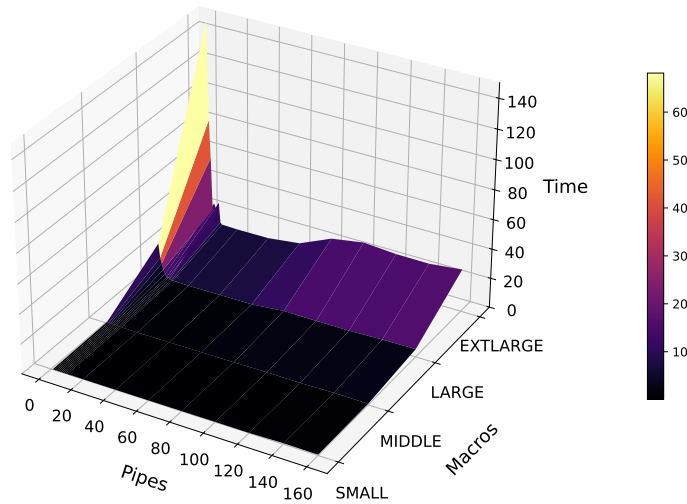


График 1.1

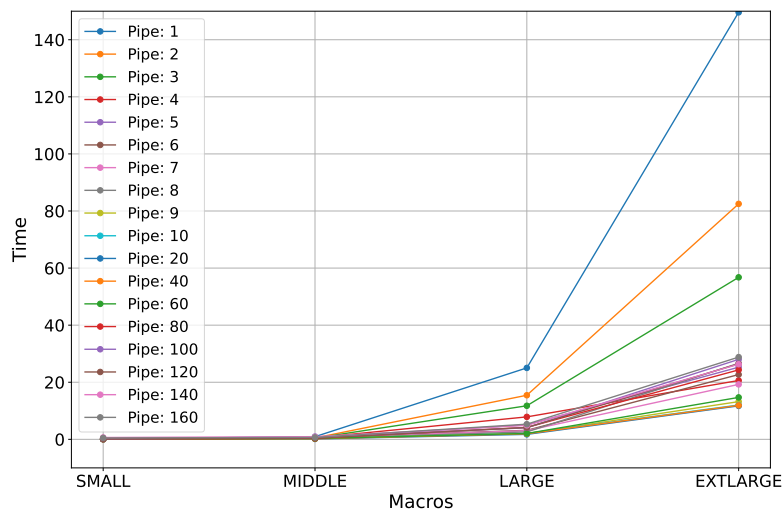


График 1.2

Графики 1.1, 1.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации с помощью директивы `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 с помощью директивы `for`. Значения указаны в секундах и округлены до сотых.

Тестирование программы на Polus. Флаг оптимизации -O2

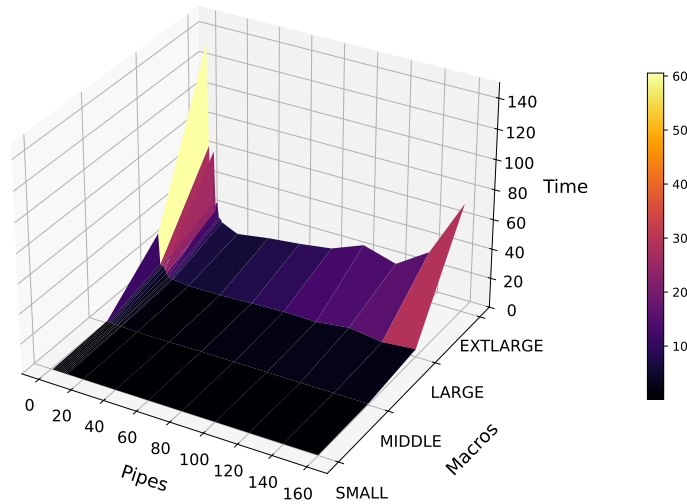


График 2.1

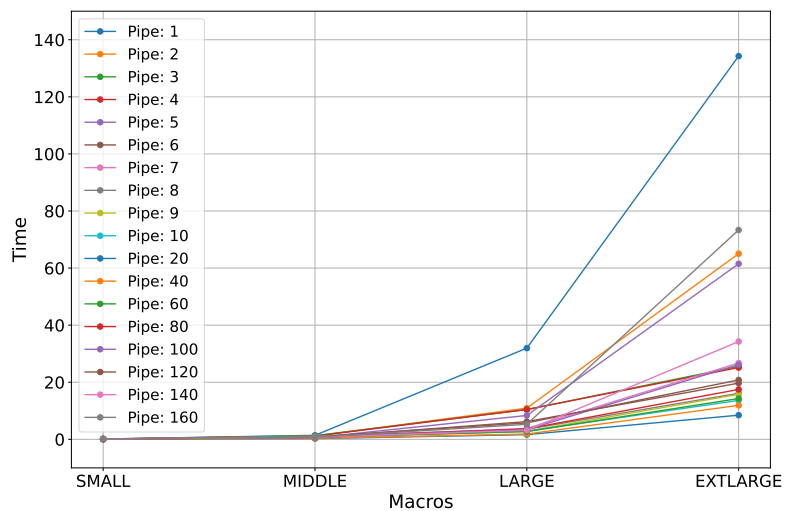


График 2.2

Графики 2.1, 2.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации с помощью директивы `for` и флага `-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 с помощью директивы `for` и флага `-O2`. Значения указаны в секундах и округлены до сотых.

Тестирование программы на Polus. Флаг оптимизации -O3

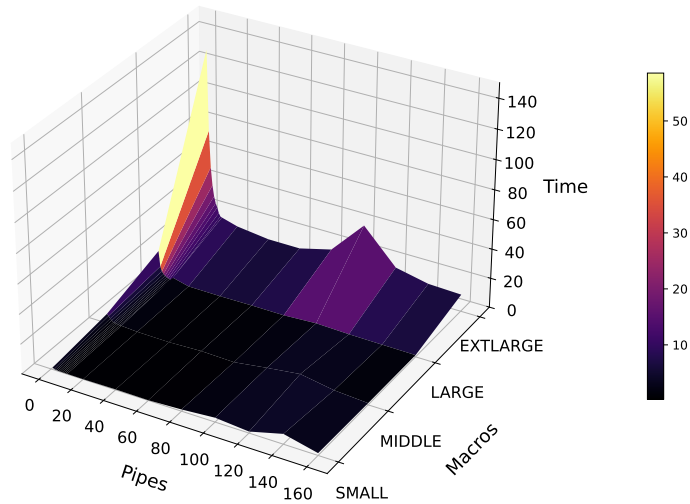


График 3.1

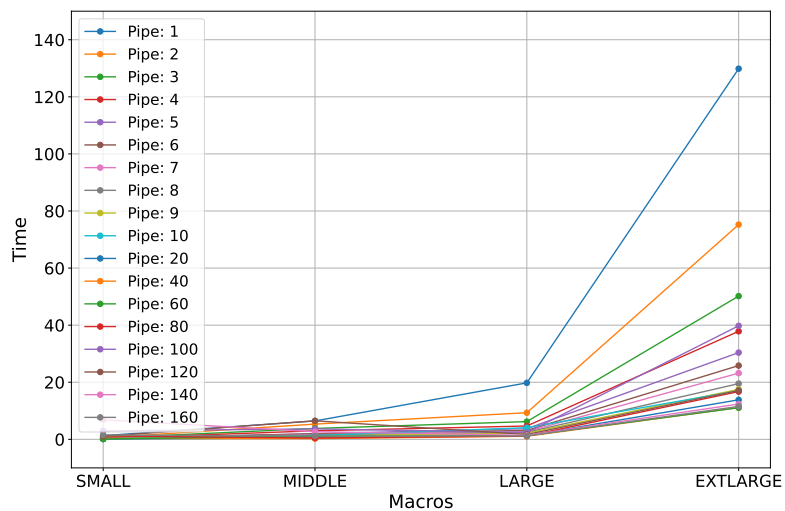


График 3.2

Графики 3.1, 3.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации с помощью директивы `for` и флага `-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 с помощью директивы `for` и флага `-O3`. Значения указаны в секундах и округлены до сотых.

Параллелизация при помощи директивы `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

```

1 #include <omp.h>
2 #include <stdio.h>
3
4 #define LARGE
5
6 #ifdef SMALL
7 #define MIMAX 65
8 #define MJMAX 33
9 #define MKMAX 33
10 #endif
11
12 #ifdef MIDDLE
13 #define MIMAX 129
14 #define MJMAX 65
15 #define MKMAX 65
16 #endif
17
18 #ifdef LARGE
19 #define MIMAX 257
20 #define MJMAX 129
21 #define MKMAX 129
22 #endif
23
24 #ifdef EXTLARGE
25 #define MIMAX 513
26 #define MJMAX 257
27 #define MKMAX 257
28 #endif
29
30 #define NN 200
31
32 static float a[MIMAX][MJMAX][MKMAX][4],
33             b[MIMAX][MJMAX][MKMAX][3], c[MIMAX][MJMAX][MKMAX][3];
34 static float p[MIMAX][MJMAX][MKMAX];
35 static float wrk1[MIMAX][MJMAX][MKMAX], wrk2[MIMAX][MJMAX][MKMAX];
36 static float bnd[MIMAX][MJMAX][MKMAX];
37
38 static int imax, jmax, kmax;
39 static float omega;
40
41
42 void
43 initmt()
44 {
45     int i, j, k;
46
47     for (i = 0; i < imax; ++i) {
48 #pragma omp task firstprivate(i) private(j, k) shared(a, b, c, p, wrk1, bnd)
49         {
50             for (j = 0; j < jmax; ++j) {
51                 for (k = 0; k < kmax; ++k) {
52                     a[i][j][k][0] = 0.0;
53                     a[i][j][k][1] = 0.0;
54                     a[i][j][k][2] = 0.0;
55                     a[i][j][k][3] = 0.0;
56                     b[i][j][k][0] = 0.0;
57                     b[i][j][k][1] = 0.0;
58                     b[i][j][k][2] = 0.0;
59                     c[i][j][k][0] = 0.0;
60                     c[i][j][k][1] = 0.0;

```

```

61         c[i][j][k][2] = 0.0;
62         p[i][j][k] = 0.0;
63         wrk1[i][j][k] = 0.0;
64         bnd[i][j][k] = 0.0;
65     }
66 }
67 }
68 }
69
70 for (i = 0; i < imax; ++i) {
71 #pragma omp task firstprivate(i) private(j, k) shared(a, b, c, p, wrk1, bnd)
72 {
73     for (j = 0; j < jmax; ++j) {
74         for (k = 0; k < kmax; ++k) {
75             a[i][j][k][0] = 1.0;
76             a[i][j][k][1] = 1.0;
77             a[i][j][k][2] = 1.0;
78             a[i][j][k][3] = 1.0 / 6.0;
79             c[i][j][k][0] = 1.0;
80             c[i][j][k][1] = 1.0;
81             c[i][j][k][2] = 1.0;
82             p[i][j][k] = (float) (k * k) /
83                         (float) ((kmax - 1) * (kmax - 1));
84             wrk1[i][j][k] = 0.0;
85             bnd[i][j][k] = 1.0;
86         }
87     }
88 }
89 }
90 }
91
92 float
93 jacobi(int nn)
94 {
95     int i, j, k, n;
96     float gosa;
97     float s0, ss;
98     float local_gosa = 0.0;
99
100 #pragma omp parallel shared(a, b, c, p, wrk1, wrk2, bnd, omega, gosa)
101 {
102 #pragma omp single
103 {
104     for (n = 0; n < nn; ++n) {
105         gosa = 0.0;
106         for (i = 1; i < imax - 1; ++i) {
107 #pragma omp task firstprivate(i) private(s0, ss, local_gosa)
108 {
109             local_gosa = 0.0;
110
111             for (j = 1; j < jmax - 1; ++j) {
112                 for (k = 1; k < kmax - 1; ++k) {
113                     s0 = a[i][j][k][0] * p[i + 1][j][k] +
114                         a[i][j][k][1] * p[i][j + 1][k] +
115                         a[i][j][k][2] * p[i][j][k + 1] +
116                         b[i][j][k][0] * (p[i + 1][j + 1][k] -
117                                     p[i + 1][j - 1][k] -
118                                     p[i - 1][j + 1][k] +
119                                     p[i - 1][j - 1][k]) +
120                         b[i][j][k][1] * (p[i][j + 1][k + 1] -
121                                     p[i][j - 1][k + 1] -
122                                     p[i][j + 1][k - 1] +

```

```

123         p[i][j - 1][k - 1]) +
124         b[i][j][k][2] * (p[i + 1][j][k + 1] -
125         p[i - 1][j][k + 1] -
126         p[i + 1][j][k - 1] +
127         p[i - 1][j][k - 1]) +
128         c[i][j][k][0] * p[i - 1][j][k] +
129         c[i][j][k][1] * p[i][j - 1][k] +
130         c[i][j][k][2] * p[i][j][k - 1] +
131         wrk1[i][j][k];
132
133
134         ss = (s0 * a[i][j][k][3] - p[i][j][k]) *
135         bnd[i][j][k];
136         local_gosa += ss * ss;
137
138         wrk2[i][j][k] = p[i][j][k] + omega * ss;
139     }
140 }
141 #pragma omp atomic
142     gosa += local_gosa;
143 #pragma omp taskwait
144 }
145 }
146
147     for (i = 1; i < imax - 1; ++i) {
148 #pragma omp task firstprivate(i)
149         for (j = 1; j < jmax - 1; ++j) {
150             for (k = 1; k < kmax - 1; ++k)
151                 p[i][j][k] = wrk2[i][j][k];
152         }
153 #pragma omp taskwait
154     }
155 }
156 }
157 }
158
159     return gosa;
160 }
161
162 int
163 main()
164 {
165     int i, j, k;
166     float gosa;
167     double cpu0, cpu1, cpu2, cpu3, nflop, xmflops2, score;
168
169     omega = 0.8;
170     imax = MIMAX - 1;
171     jmax = MJMAX - 1;
172     kmax = MKMAX - 1;
173
174     cpu2 = omp_get_wtime();
175
176     initmt();
177
178     cpu3 = omp_get_wtime();
179
180     printf("mimax = %d mjmax = %d mkmax = %d\n", MIMAX, MJMAX, MKMAX);
181     printf("imax = %d jmax = %d kmax = %d\n", imax, jmax, kmax);
182
183
184     cpu0 = omp_get_wtime();

```

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


Тестирование программы на Polus

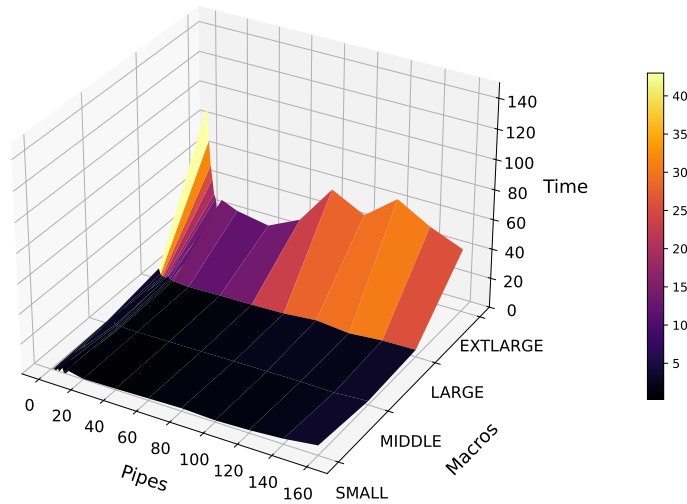


График 4.1

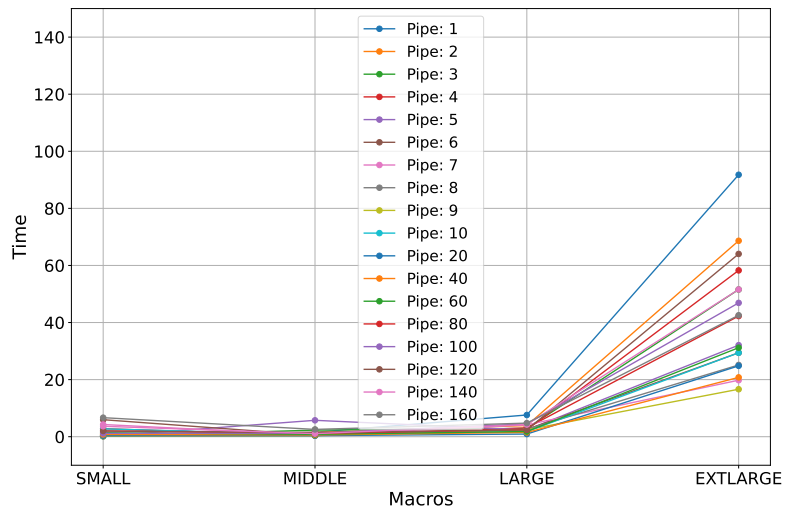


График 4.2

Графики 4.1, 4.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации с помощью директивы `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 с помощью директивы `task`. Значения указаны в секундах и округлены до сотых.

Тестирование программы на Polus. Флаг оптимизации -02

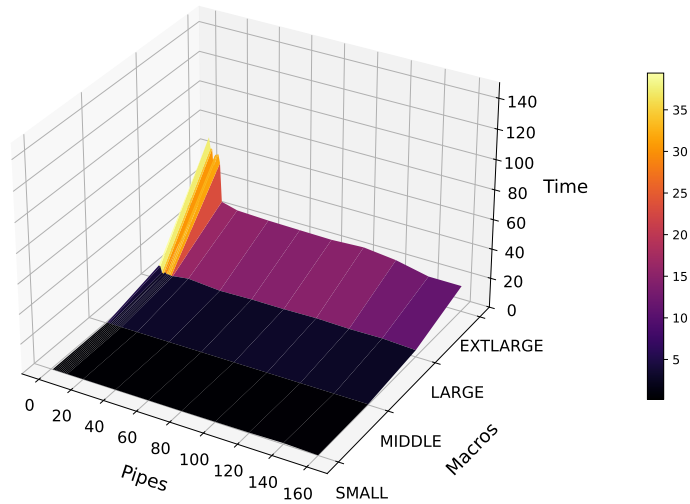


График 5.1

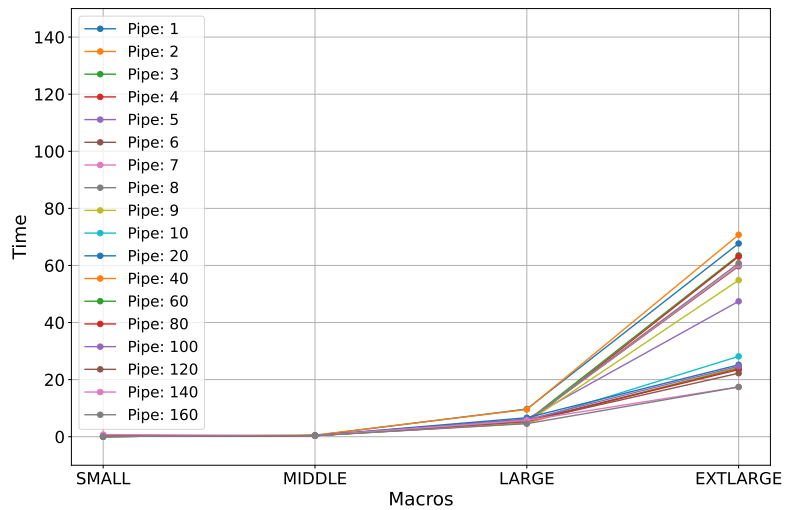


График 5.2

Графики 5.1, 5.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации с помощью директивы *task* и флага -02.

	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 с помощью директивы *task* и флага -02. Значения указаны в секундах и округлены до сотых.

Тестирование программы на Polus. Флаг оптимизации -O3

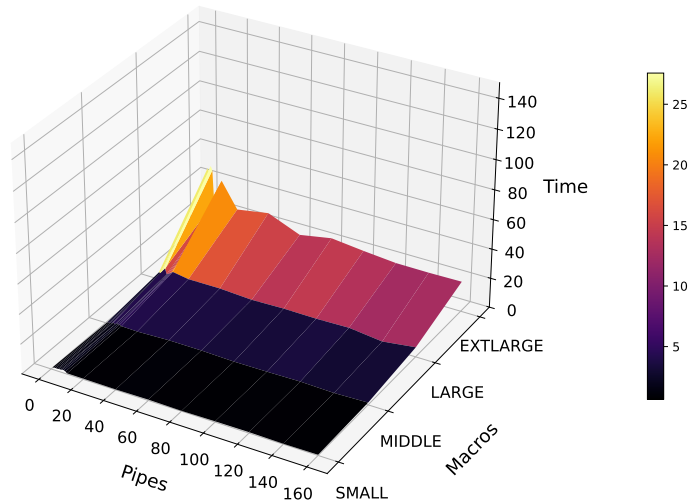


График 6.1

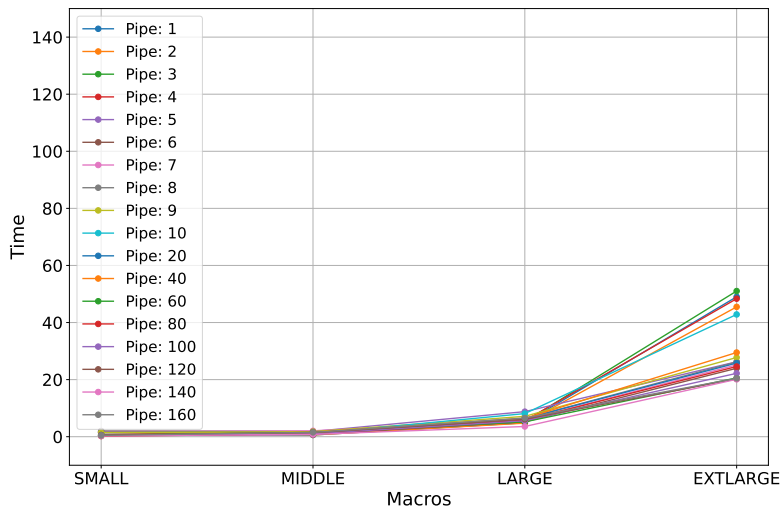


График 6.2

Графики 6.1, 6.2 — Зависимость времени работы программы от объема входных данных и количества потоков при оптимизации с помощью директивы `task` и флага `-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 с помощью директивы `task` и флага `-O3`. Значения указаны в секундах и округлены до сотых.

Выводы об OpenMP

После реализации программ и проведения тестов можно сделать следующие 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.

MPI

Параллелизация при помощи MPI

Внесенные изменения

Для преобразования последовательной программы в параллельную с использованием MPI были внесены следующие изменения:

- Добавлены вызовы инициализации и завершения работы библиотеки MPI: `MPI_Init`, `MPI_Finalize`, а также определение числа процессов посредством `MPI_Comm_size` и получение ранга каждого процесса с помощью `MPI_Comm_rank`.
- Массивы в программе распределены по измерению `i`, так что каждый процесс обрабатывает лишь подмассив, соответствующий определённому диапазону индексов по оси `i`. Для этого полный диапазон `i` (от 0 до `imax-1`) был разделён на примерно равные блоки по числу процессов, учитывая возможный остаток при неделимости длины массива на число процессов. Таким образом, если существует `size` процессов, то каждый процесс с рангом `rank` получает свой участок `[i_start, i_end]`, причём значения `i_start` и `i_end` вычисляются исходя из общего числа процессов и текущего ранга.
- Реализован обмен граничными слоями между соседями по оси `i` с помощью неблокирующих передач `MPI_Isend` и `MPI_Irecv`, что обеспечивает корректную обработку внутренних точек при вычислениях.
- Для сбора и агрегирования локальных результатов, таких как ошибка `gosa`, применена операция `MPI_Allreduce`, позволяющая получить глобальное значение ошибки.
- Функция измерения времени `second()` заменена на `MPI_Wtime()`, обеспечивающую синхронизированное измерение времени в параллельном режиме.

Код программы

```

1 #include <mpi.h>
2 #include <stdio.h>
3 #include <sys/time.h>
4
5 #define SMALL
6
7 #ifdef SMALL
8 #define MIMAX 65
9 #define MJMAX 33
10 #define MKMAX 33
11 #endif
12
13 #ifdef MIDDLE
14 #define MIMAX 129
15 #define MJMAX 65
16 #define MKMAX 65
17 #endif
18
19 #ifdef LARGE
20 #define MIMAX 257
21 #define MJMAX 129
22 #define MKMAX 129
23 #endif
24
25 #ifdef EXTLARGE
26 #define MIMAX 513
27 #define MJMAX 257
28 #define MKMAX 257
29 #endif
30
31 #define NN 200
32
33 float jacobi(int, int, int, int);
34 void initmt(int, int, int, int);
35
36 static float p[MIMAX][MJMAX][MKMAX];
37 static float a[MIMAX][MJMAX][MKMAX][4], b[MIMAX][MJMAX][MKMAX][3],
38 c[MIMAX][MJMAX][MKMAX][3];
39 static float bnd[MIMAX][MJMAX][MKMAX];
40 static float wrk1[MIMAX][MJMAX][MKMAX], wrk2[MIMAX][MJMAX][MKMAX];
41
42 static int imax, jmax, kmax;
43 static float omega;
44
45 void initmt(int i_start, int i_end, int jmax, int kmax) {
46     int i, j, k;
47     for (i = i_start; i <= i_end; i++)
48         for (j = 0; j < jmax; ++j)
49             for (k = 0; k < kmax; ++k) {
50                 a[i][j][k][0] = 1.0;
51                 a[i][j][k][1] = 1.0;
52                 a[i][j][k][2] = 1.0;
53                 a[i][j][k][3] = 1.0 / 6.0;
54                 b[i][j][k][0] = 0.0;
55                 b[i][j][k][1] = 0.0;
56                 b[i][j][k][2] = 0.0;
57                 c[i][j][k][0] = 1.0;
58                 c[i][j][k][1] = 1.0;
59                 c[i][j][k][2] = 1.0;
60                 p[i][j][k] =

```

```

61         (float)(k * k) / (float)((kmax - 1) * (kmax - 1));
62         wrk1[i][j][k] = 0.0;
63         bnd[i][j][k] = 1.0;
64         wrk2[i][j][k] = 0.0;
65     }
66 }
67
68 float jacobi(int nn, int i_start, int i_end, int size) {
69     int i, j, k, n;
70     float gosa_local, gosa_global;
71     int rank;
72     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
73
74     int left = (rank == 0) ? MPI_PROC_NULL : rank - 1;
75     int right = (rank == size - 1) ? MPI_PROC_NULL : rank + 1;
76
77     static float send_left[MJMAX][MKMAX], send_right[MJMAX][MKMAX];
78     static float recv_left[MJMAX][MKMAX], recv_right[MJMAX][MKMAX];
79
80     for (n = 0; n < nn; ++n) {
81         if (i_start > 0) {
82             for (j = 0; j < jmax; j++)
83                 for (k = 0; k < kmax; k++)
84                     send_left[j][k] = p[i_start][j][k];
85         }
86         if (i_end < imax - 1) {
87             for (j = 0; j < jmax; j++)
88                 for (k = 0; k < kmax; k++)
89                     send_right[j][k] = p[i_end][j][k];
90         }
91
92         MPI_Request reqs[4];
93         int req_count = 0;
94
95         if (left != MPI_PROC_NULL) {
96             MPI_Isend(&send_left[0][0], jmax * kmax, MPI_FLOAT, left, 0,
97                     MPI_COMM_WORLD, &reqs[req_count++]);
98             MPI_Irecv(&recv_left[0][0], jmax * kmax, MPI_FLOAT, left, 1,
99                     MPI_COMM_WORLD, &reqs[req_count++]);
100         }
101         if (right != MPI_PROC_NULL) {
102             MPI_Isend(&send_right[0][0], jmax * kmax, MPI_FLOAT, right, 1,
103                     MPI_COMM_WORLD, &reqs[req_count++]);
104             MPI_Irecv(&recv_right[0][0], jmax * kmax, MPI_FLOAT, right, 0,
105                     MPI_COMM_WORLD, &reqs[req_count++]);
106         }
107
108         MPI_Waitall(req_count, reqs, MPI_STATUSES_IGNORE);
109
110         if (left != MPI_PROC_NULL) {
111             for (j = 0; j < jmax; j++)
112                 for (k = 0; k < kmax; k++)
113                     p[i_start - 1][j][k] = recv_left[j][k];
114         }
115         if (right != MPI_PROC_NULL) {
116             for (j = 0; j < jmax; j++)
117                 for (k = 0; k < kmax; k++)
118                     p[i_end + 1][j][k] = recv_right[j][k];
119         }
120
121         gosa_local = 0.0f;
122     }

```

```

123     for (i = (i_start > 1 ? i_start : 1);
124           i <= (i_end < imax - 2 ? i_end : imax - 2); i++)
125         for (j = 1; j < jmax - 1; ++j)
126             for (k = 1; k < kmax - 1; ++k) {
127                 float s0 = a[i][j][k][0] * p[i + 1][j][k] +
128                           a[i][j][k][1] * p[i][j + 1][k] +
129                           a[i][j][k][2] * p[i][j][k + 1] +
130                           b[i][j][k][0] *
131                             (p[i + 1][j + 1][k] - p[i + 1][j - 1][k] -
132                              p[i - 1][j + 1][k] + p[i - 1][j - 1][k]) +
133                           b[i][j][k][1] *
134                             (p[i][j + 1][k + 1] - p[i][j - 1][k + 1] -
135                              p[i][j + 1][k - 1] + p[i][j - 1][k - 1]) +
136                           b[i][j][k][2] *
137                             (p[i + 1][j][k + 1] - p[i - 1][j][k + 1] -
138                              p[i + 1][j][k - 1] + p[i - 1][j][k - 1]) +
139                           c[i][j][k][0] * p[i - 1][j][k] +
140                           c[i][j][k][1] * p[i][j - 1][k] +
141                           c[i][j][k][2] * p[i][j][k - 1] + wrk1[i][j][k];
142
143                 float ss =
144                     (s0 * a[i][j][k][3] - p[i][j][k]) * bnd[i][j][k];
145                 gosa_local += ss * ss;
146                 wrk2[i][j][k] = p[i][j][k] + omega * ss;
147             }
148
149     for (i = (i_start > 1 ? i_start : 1);
150           i <= (i_end < imax - 2 ? i_end : imax - 2); i++)
151         for (j = 1; j < jmax - 1; ++j)
152             for (k = 1; k < kmax - 1; ++k)
153                 p[i][j][k] = wrk2[i][j][k];
154
155     MPI_Allreduce(&gosa_local, &gosa_global, 1, MPI_FLOAT, MPI_SUM,
156                  MPI_COMM_WORLD);
157 }
158
159 return gosa_global;
160 }
161
162 int main(int argc, char* argv[]) {
163     int i, j, k;
164     int rank, size;
165     double cpu0, cpu1, nflop, xmflops2, score;
166     float gosa_local, gosa;
167
168     MPI_Init(&argc, &argv);
169     MPI_Comm_size(MPI_COMM_WORLD, &size);
170     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
171
172     omega = 0.8;
173     imax = MIMAX - 1;
174     jmax = MJMAX - 1;
175     kmax = MKMAX - 1;
176
177     int i_block = imax / size;
178     int rest = imax % size;
179     int i_start, i_end;
180     if (rank < rest) {
181         i_block = imax / size + 1;
182         i_start = rank * i_block;
183         i_end = i_start + i_block - 1;
184     } else {

```



```

185     i_block = imax / size;
186     i_start = rest * ((imax / size) + 1) + (rank - rest) * i_block;
187     i_end = i_start + i_block - 1;
188 }
189
190 initmt(i_start, i_end, jmax, kmax);
191
192 if (rank == 0) {
193     printf("mimax = %d mjmax = %d mkmax = %d\n", MIMAX, MJMAX, MKMAX);
194     printf("imax = %d jmax = %d kmax = %d\n", imax, jmax, kmax);
195 }
196
197 MPI_Barrier(MPI_COMM_WORLD);
198 cpu0 = MPI_Wtime();
199
200 gosa = jacobi(NN, i_start, i_end, size);
201
202 MPI_Barrier(MPI_COMM_WORLD);
203 cpu1 = MPI_Wtime();
204
205 nflop = (kmax - 2) * (jmax - 2) * (imax - 2) * 34;
206
207 if (rank == 0) {
208     if (cpu1 != 0.0)
209         xmflops2 = nflop / cpu1 * 1.0e-6 * (float)NN;
210     else
211         xmflops2 = 0.0;
212
213     score = xmflops2 / 32.27;
214
215     printf("%f,\n", cpu1 - cpu0);
216     printf("cpu : %f sec.\n", cpu1 - cpu0);
217     printf("Loop executed for %d times\n", NN);
218     printf("Gosa : %e \n", gosa);
219     printf("MFLOPS measured : %f\n", xmflops2);
220     printf("Score based on MMX Pentium 200MHz : %f\n", score);
221 }
222
223 MPI_Finalize();
224 return (0);
225 }

```

Тестирование программы на Polus

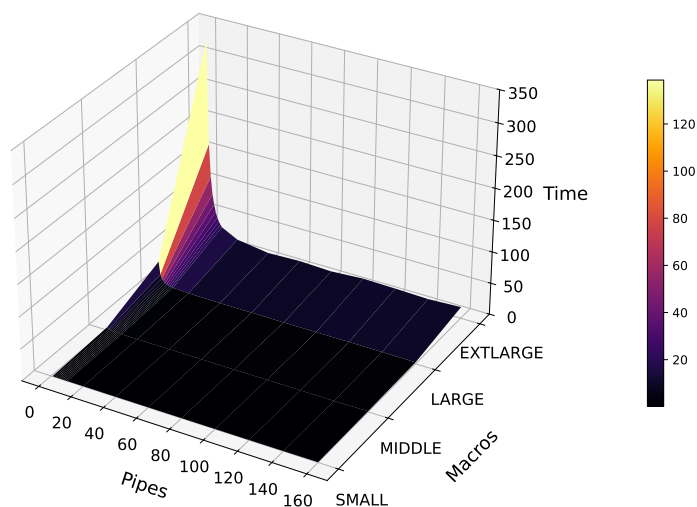


График 7.1

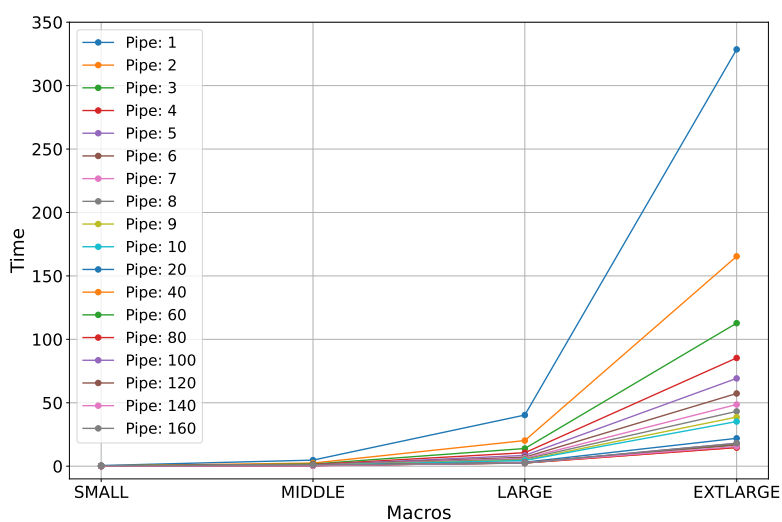


График 7.2

Графики 7.1, 7.2 — Зависимость времени работы программы от объема входных данных и количества потоков с помощью *mpi*.

	1	2	3	4	5	6	7	8	9
SMALL	0.54	0.28	0.24	0.16	0.12	0.22	0.11	0.11	0.15
MIDDLE	4.78	2.47	1.71	1.31	1.05	0.98	0.77	0.89	0.66
LARGE	40.32	20.21	13.93	10.62	8.51	7.10	6.20	5.54	4.85
EXTLARGE	328.54	165.42	112.75	85.33	69.20	57.30	48.63	43.20	38.70

	10	20	40	60	80	100	120	140	160
SMALL	0.15	0.11	0.10	0.06	0.06	0.07	0.07	0.19	0.36
MIDDLE	0.66	0.54	0.51	0.55	0.53	0.49	0.53	0.31	0.98
LARGE	4.51	3.14	2.54	2.60	2.49	2.57	2.80	2.70	2.43
EXTLARGE	35.19	21.96	14.78	16.61	14.68	18.08	17.06	15.52	18.06

Таблица 7. Результаты оптимизации с помощью *mpi*. Значения указаны в секундах и округлены до сотых.

Тестирование программы на Polus. Флаг оптимизации -02

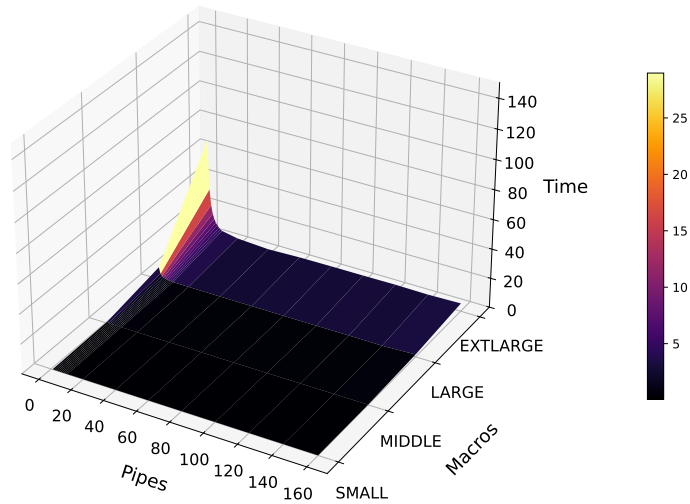


График 8.1

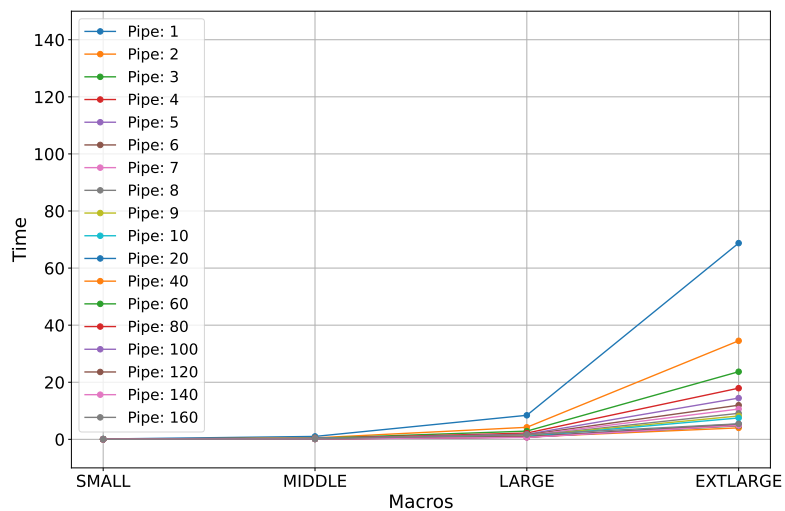


График 8.2

Графики 8.1, 8.2 — Зависимость времени работы программы от объема входных данных и количества потоков с помощью `mpi` и флага `-02`.

	1	2	3	4	5	6	7	8	9
SMALL	0.12	0.06	0.07	0.05	0.04	0.06	0.03	0.04	0.04
MIDDLE	1.02	0.52	0.36	0.29	0.33	0.25	0.23	0.23	0.19
LARGE	8.42	4.21	2.88	2.20	1.78	1.57	1.35	1.18	1.14
EXTLARGE	68.74	34.52	23.69	17.92	14.45	11.95	10.61	9.11	8.19

	10	20	40	60	80	100	120	140	160
SMALL	0.04	0.05	0.03	0.03	0.02	0.02	0.03	0.03	0.03
MIDDLE	0.29	0.21	0.20	0.21	0.14	0.15	0.15	0.15	0.13
LARGE	0.99	0.90	0.80	0.81	0.76	0.75	0.82	0.67	1.65
EXTLARGE	7.49	5.35	4.02	4.74	4.93	4.98	5.16	4.87	5.43

Таблица 8. Результаты оптимизации с помощью `mpi` и флага `-02`. Значения указаны в секундах и округлены до сотых.

Тестирование программы на Polus. Флаг оптимизации -O3

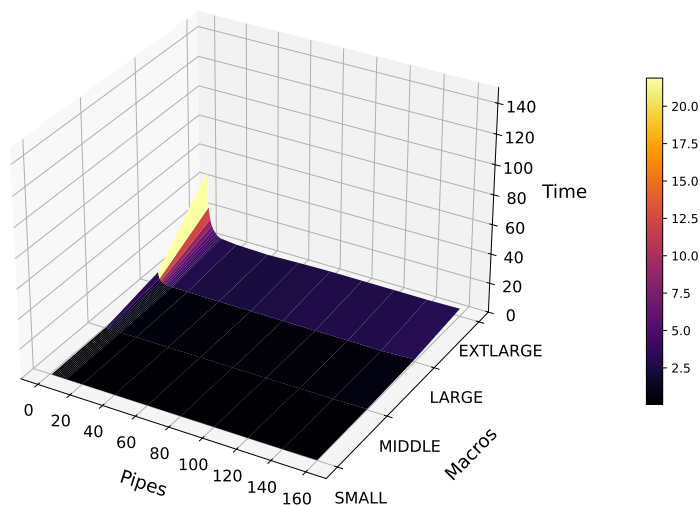


График 9.1

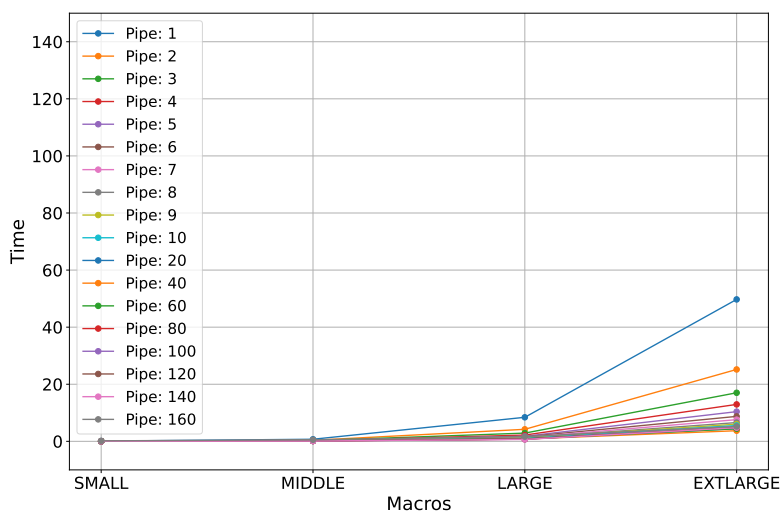


График 9.2

Графики 9.1, 9.2 — Зависимость времени работы программы от объема входных данных и количества потоков с помощью `mpi` и флага `-O3`.

	1	2	3	4	5	6	7	8	9
SMALL	0.08	0.04	0.03	0.04	0.03	0.04	0.02	0.04	0.03
MIDDLE	0.73	0.37	0.26	0.21	0.24	0.18	0.17	0.15	0.27
LARGE	8.42	4.21	2.88	2.20	1.78	1.57	1.35	1.18	1.14
EXTLARGE	49.72	25.18	17.01	12.94	10.40	8.74	7.60	6.57	6.01

	10	20	40	60	80	100	120	140	160
SMALL	0.01	0.04	0.03	0.03	0.02	0.02	0.02	0.02	0.17
MIDDLE	0.18	0.22	0.19	0.16	0.12	0.13	0.17	0.11	0.55
LARGE	0.99	0.90	0.80	0.81	0.76	0.75	0.82	0.67	1.65
EXTLARGE	5.55	4.42	3.72	4.42	4.85	4.82	4.84	4.79	5.22

Таблица 9. Результаты оптимизации с помощью `mpi` и флага `-O3`. Значения указаны в секундах и округлены до сотых.

Выводы о MPI

После проведения анализа результатов тестирования программы с использованием MPI можно сделать следующие выводы:

- **Малые (SMALL) наборы данных:** Для малых наборов данных увеличение числа потоков не всегда приводит к улучшению производительности. Например, в базовой версии (без флагов) время выполнения существенно колеблется при большем количестве потоков. С применением флагов -02 и -03 ситуация становится стабильнее, но наибольшее ускорение наблюдается только при ограниченном числе потоков (до 20). Накладные расходы на синхронизацию потоков приводят к деградации производительности при большем числе потоков.
- **Средние (MIDDLE) наборы данных:** Для средних наборов данных наблюдается значительное уменьшение времени выполнения с увеличением числа потоков, особенно при использовании флагов компиляции. Флаг -03 показывает наилучшую производительность для числа потоков от 10 до 80. Однако при увеличении потоков более 120 начинает проявляться эффект насыщения, и производительность стабилизируется либо ухудшается из-за накладных расходов.
- **Большие (LARGE) наборы данных:** Для больших данных оптимизация более заметна. Базовая версия программы демонстрирует монотонное улучшение производительности с увеличением числа потоков. Применение флагов -02 и -03 позволяет достичь значительно лучшего ускорения, особенно при числе потоков до 40. При этом флаг -03 показывает лучшее время выполнения, но производительность стабилизируется при большом числе потоков (от 80 и выше).
- **Экстремально большие (EXTLARGE) наборы данных:** Для экстремально больших данных наибольшее ускорение достигается при использовании флагов -02 и -03. Особенно заметен эффект флага -03, который обеспечивает лучшее распределение нагрузки между потоками. Однако при большом числе потоков (свыше 100) производительность стабилизируется, а в некоторых случаях даже деградирует из-за накладных расходов на синхронизацию.

Влияние флагов компиляции

- -02: Флаг -02 значительно улучшает производительность для всех наборов данных. Наибольший эффект наблюдается для средних (MIDDLE) и больших (LARGE) данных, где время выполнения сокращается более чем в два раза. Для экстремально больших (EXTLARGE) данных флаг обеспечивает стабильность при числе потоков от 80, снижая влияние синхронизации.
- -03: Флаг -03 показывает лучшие результаты, особенно на больших (LARGE) и экстремально больших (EXTLARGE) данных, с максимальным ускорением при 60–80 потоках. Однако на малых (SMALL) данных или при числе потоков выше 140 агрессивные оптимизации иногда приводят к нестабильности из-за накладных расходов.

Выводы

На основе результатов тестирования можно сделать следующие выводы о производительности реализаций параллелизма с использованием OpenMP (`for` и `task`) и MPI:

1. OpenMP (директива `for`):

- Хорошо работает с относительно мелкими объемами данных (`SMALL` и `MIDDLE`).
- Демонстрирует стабильную производительность при увеличении числа потоков.
- При больших объемах данных (`LARGE` и `EXTLARGE`) производительность начинает снижаться, особенно если число потоков превышает оптимальное значение.
- Для больших данных использование оптимизаций (`-O2`, `-O3`) значительно улучшает производительность.

2. OpenMP (директива `task`):

- Эффективна на большом числе потоков при условии оптимизации. Однако в некоторых случаях наблюдается рост времени выполнения из-за накладных расходов на управление задачами.
- Показывает преимущество при обработке крупных данных (`LARGE` и `EXTLARGE`), особенно с оптимизациями компилятора.
- Меньше подходит для мелких объемов данных, где накладные расходы на управление задачами оказываются значительными.

3. MPI:

- Наиболее эффективна для очень больших объемов данных (`EXTLARGE`).
- Масштабируемость выше, чем у OpenMP, при условии правильного разбиения данных.

Приложения

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