Отчёт по практическому заданию

«Суперкомпьютеры и параллельная обработка данных»

Вариант задания: №37

Студент: Мозжухин А.В.
Группа: 321

Преподаватель: Бахтин В.А.

Московский Государственный Университет имени М.В. Ломоносова

Содержание

1 Постановка задачи **2**

2 Описание исходного алгоритма **2**
2.1 Исходные данные 3
2.2 Код оригинальной программы 3

3 Анализ производительности оригинальной программы **5**

4 Использование OpenMP **6**
4.1 Директива for 6
4.1.1 Отличия от оригинальной 6
4.1.2 Код программы 6
4.1.3 Графическое представление результатов 9
4.2 Директива task 11
4.2.1 Отличия от оригинальной 11
4.2.2 Код программы 12
4.2.3 Графическое представление результатов 16
4.3 Выводы и сравнения OMP программ с оригинальной 18
4.3.1 Сравнение производительности на разном количестве потоков 18
4.3.2 Сравнение производительности на разных размерах данных . 18
4.3.3 Общие выводы 18

5 Использование MPI **19**
5.1 Отличия от оригинальной программы 19
5.2 Код программы 20
5.3 Графическое представление результатов 25
5.4 Выводы и сравнения MPI программы с оригинальной 27
5.4.1 Сравнение производительности на разном количестве процессов 27
5.4.2 Сравнение производительности на разных размерах данных . 27
5.4.3 Общие выводы 27

6 Финальные выводы **28**
6.1 Когда все программы работают хорошо 28
6.2 Когда все программы работают плохо 28
6.3 Когда одна из программ лучше 29
6.4 Общие выводы 29

7 Приложения **29**

1 Постановка задачи

Целью данной работы является модификация исходной последовательной программы для решения задачи численного моделирования с использованием параллельных технологий OpenMP и MPI.

Исходная программа решает трёхмерное уравнение Лапласа на кубической области, используя метод Якоби. Задача состоит в нахождении стационарного распределения значений внутри области с заданными краевыми условиями.

Функционал программы распределён следующим образом:

- `init()`: инициализация массива с краевыми значениями (равными нулю) и начальной аппроксимацией для внутренних узлов.
- `relax()`: итеративное обновление значений массива с вычислением ошибки eps, представляющей максимальную разницу между новым и старым значениями элементов.
- `verify()`: вычисление итоговой суммы для проверки корректности работы программы.

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.
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.

2 Описание исходного алгоритма

Алгоритм включает следующие этапы:

1. **Инициализация (функция `init()`)**: массив размером $N \times N \times N$ заполняется начальными значениями.
   - Границы области (краевые точки) устанавливаются в нулевые значения: $A[i][j][k] = 0$.
   - Внутренние точки получают значения $A[i][j][k] = 4 + i + j + k$, что задаёт начальное приближение для метода итераций.

2. **Итерации (функция `relax()`)**: Значения элементов массива обновляются по формуле:

$$u_{i,j,k}^{(n+1)} = \frac{1}{4}\left(u_{i+1,j,k}^{(n)} + u_{i-1,j,k}^{(n)} + u_{i,j+1,k}^{(n)} + u_{i,j-1,k}^{(n)}\right) + \text{дополнительные члены.}$$

   Также учитываются дальние соседи $(i \pm 2, j \pm 2, k \pm 2)$, что позволяет улучшить аппроксимацию.

3. **Условие сходимости**: Итерации продолжаются до тех пор, пока значение максимальной разницы между текущими и новыми значениями eps не станет меньше заданного предела maxeps.

4. **Верификация (функция `verify()`)**: После завершения итераций вычисляется итоговая сумма:

$$S = \sum_{i,j,k} A[i][j][k] \cdot \frac{(i+1)(j+1)(k+1)}{N^3}.$$

   Это служит для проверки корректности работы алгоритма.

2.1 Исходные данные

Для тестирования программы используются следующие параметры:

- Размерность массива:
  - $N = 66$.
  - $N = 130$.
  - $N = 258$.
- Параметры сходимости:
  - Максимально допустимая ошибка: `maxeps = 0.1 * e-7`.
  - Максимальное число итераций: `itmax = 100`.

2.2 Код оригинальной программы

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#define  Max(a,b) ((a)>(b)?(a):(b))

#define  N   (2*2*2*2*2*2+2)
float   maxeps = 0.1e-7;
int itmax = 100;
int i,j,k;

float eps;
float A [N][N][N];

```

```
14 void relax();
15 void init();
16 void verify();
17
18 int main(int an, char **as)
19 {
20     int it;
21
22     init();
23
24     for(it=1; it<=itmax; it++)
25     {
26         eps = 0.;
27         relax();
28         printf( "it=%4i   eps=%f\\n", it,eps);
29         if (eps < maxeps) break;
30     }
31
32     verify();
33
34     return 0;
35 }
36
37 void init()
38 {
39     for(k=0; k<=N-1; k++)
40     for(j=0; j<=N-1; j++)
41     for(i=0; i<=N-1; i++)
42     {
43         if(i==0 || i==N-1 || j==0 || j==N-1 || k==0 || k==N-1)
44         A[i][j][k]= 0.;
45         else A[i][j][k]= ( 4. + i + j + k) ;
46     }
47 }
48
49 void relax()
50 {
51     for(k=1; k<=N-2; k++)
52     for(j=1; j<=N-2; j++)
53     for(i=2; i<=N-3; i++)
54     {
55         A[i][j][k] = (A[i-1][j][k]+A[i+1][j][k]+
56                 A[i-2][j][k]+A[i+2][j][k])/4.;
57     }
58
59     for(k=1; k<=N-2; k++)
60     for(j=2; j<=N-3; j++)
61     for(i=1; i<=N-2; i++)
62     {
63         A[i][j][k] =(A[i][j-1][k]+A[i][j+1][k]+
64                 A[i][j-2][k]+A[i][j+2][k])/4.;
```

```
65      }
66
67      for(k=2; k<=N-3; k++)
68      for(j=1; j<=N-2; j++)
69      for(i=1; i<=N-2; i++)
70      {
71          float e;
72          e=A[i][j][k];
73          A[i][j][k] = (A[i][j][k-1]+A[i][j][k+1]+
74                  A[i][j][k-2]+A[i][j][k+2])/4.;
75          eps=Max(eps,fabs(e-A[i][j][k]));
76      }
77  }
78
79  void verify()
80  {
81      float s;
82
83      s=0.;
84      for(k=0; k<=N-1; k++)
85      for(j=0; j<=N-1; j++)
86      for(i=0; i<=N-1; i++)
87      {
88          s=s+A[i][j][k]*(i+1)*(j+1)*(k+1)/(N*N*N);
89      }
90      printf("  S = %f\\n",s);
91  }
```

Листинг 1: Код программы var37.c

3 Анализ производительности оригинальной программы

Для оценки производительности оригинальной программы были проведены эксперименты с различными размерами массива N. Результаты представлены в таблице:

Таблица 1: Время выполнения исходной программы

Размер данных N	**Среднее время выполнения (сек)**
66	0.4747
130	4.8033
258	87.6646

График зависимости времени выполнения от размера массива:

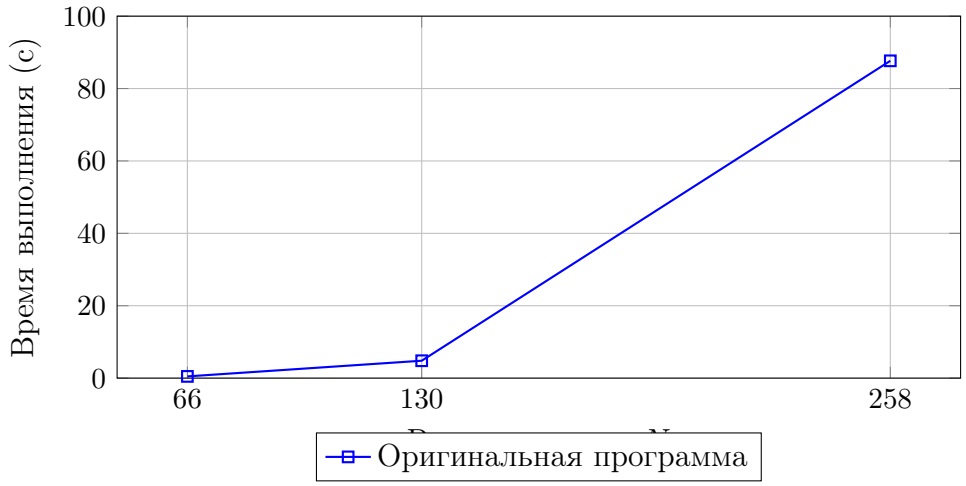

Рис. 1: Зависимость времени выполнения от размера массива N.

4 Использование OpenMP

4.1 Директива for

4.1.1 Отличия от оригинальной

В программе с использованием OpenMP реализованы следующие изменения:

- `#pragma omp parallel for collapse(3) shared(A) default(none)`: Параллелизация трёх вложенных циклов в функции `init`, с указанием переменной `A` как разделяемой, а остальных переменных как явно объявленных.
- `#pragma omp parallel for collapse(3) schedule(static)`: Используется для равномерного распределения итераций цикла между потоками в функции `relax`.
- `#pragma omp parallel for collapse(3) reduction(max:local_eps) schedule(static)`: Добавлен механизм `reduction` для параллельного вычисления максимальной разницы между старым и новым значением в функции `relax`.
- `#pragma omp critical`: Обеспечивает синхронизацию при обновлении глобальной переменной `eps`.
- `#pragma omp parallel for collapse(3) reduction(+:s) shared(A) default(none)`: Реализована параллелизация вычисления итоговой суммы в функции `verify`, с использованием `reduction` для суммирования.

4.1.2 Код программы

```
92 #include <math.h>
93 #include <stdlib.h>
94 #include <stdio.h>
95 #include <omp.h>
```

```
#define  Max(a,b) ((a)>(b)?(a):(b))

#define  N   (66)
float maxeps = 0.1e-7;
int itmax = 100;

float eps;
float A[N][N][N];

void relax();
void init();
void verify();

int main() {
    printf("OMP_THREADS count=%d\n", omp_get_max_threads());
    int it;
    double program_start_time, program_end_time;

    program_start_time = omp_get_wtime();

    init();
    for (it = 1; it <= itmax; it++) {
        eps = 0.0f;

        relax();

        if (eps < maxeps) break;
    }
    verify();
    program_end_time = omp_get_wtime() - program_start_time;
    printf("TOTAL_TIME: %f\n", program_end_time);
    return 0;
}

void init() {
#pragma omp parallel for collapse(3) shared(A) default(none)
    for (int k = 0; k < N; k++)
        for (int j = 0; j < N; j++)
            for (int i = 0; i < N; i++) {
                if (i == 0 || i == N - 1 || j == 0 ||
                        j == N - 1 || k == 0 || k == N - 1)
                    A[i][j][k] = 0.0f;
                else
                    A[i][j][k] = 4.0f + i + j + k;
            }
}

void relax() {
    float local_eps = 0.0f;

#pragma omp parallel for collapse(3) schedule(static)
```

```
147     for(int k = 1; k <= N - 2; k++)
148         for(int j = 1; j <= N - 2; j++)
149             for(int i = 2; i <= N - 3; i++) {
150                 A[i][j][k] = (A[i-1][j][k] + A[i+1][j][k] +
151                 A[i-2][j][k] + A[i+2][j][k]) / 4.0f;
152             }
153
154 #pragma omp parallel for collapse(3) schedule(static)
155     for(int k = 1; k <= N - 2; k++)
156         for(int j = 2; j <= N - 3; j++)
157             for(int i = 1; i <= N - 2; i++) {
158                 A[i][j][k] = (A[i][j-1][k] + A[i][j+1][k] +
159                 A[i][j-2][k] + A[i][j+2][k]) / 4.0f;
160             }
161
162 #pragma omp parallel for collapse(3)
163             reduction(max:local_eps) schedule(static)
164     for(int k = 2; k <= N - 3; k++)
165         for(int j = 1; j <= N - 2; j++)
166             for(int i = 1; i <= N - 2; i++) {
167                 float e = A[i][j][k];
168                 A[i][j][k] = (A[i][j][k-1] + A[i][j][k+1] +
169                 A[i][j][k-2] + A[i][j][k+2]) / 4.0f;
170                 float diff = fabsf(e - A[i][j][k]);
171                 if (diff > local_eps)
172                     local_eps = diff;
173             }
174
175 #pragma omp critical
176     {
177         eps = Max(eps, local_eps);
178     }
179 }
180
181 void verify() {
182     float s = 0.0f;
183
184 #pragma omp parallel for collapse(3) reduction(+:s)
185             shared(A) default(none)
186     for (int k = 0; k < N; k++)
187         for (int j = 0; j < N; j++)
188             for (int i = 0; i < N; i++) {
189                 s += A[i][j][k] * (i + 1) * (j + 1) * (k + 1) /
190                         (float)(N * N * N);
191             }
192     printf("  S = %f\n", s);
193 }
```

Листинг 2: Код программы var37_for.c

4.1.3 Графическое представление результатов

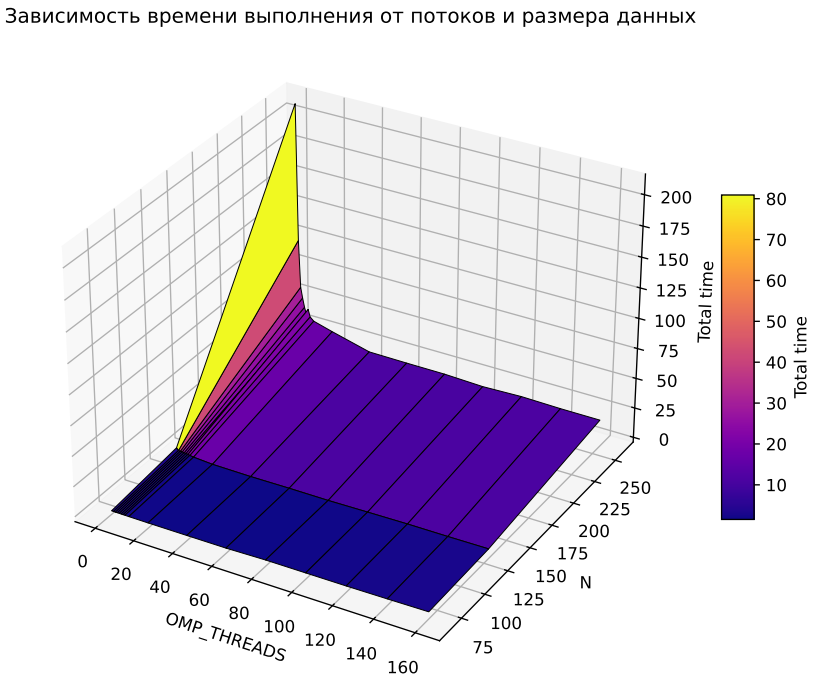

Рис. 2: Зависимость времени выполнения от потоков и размера данных

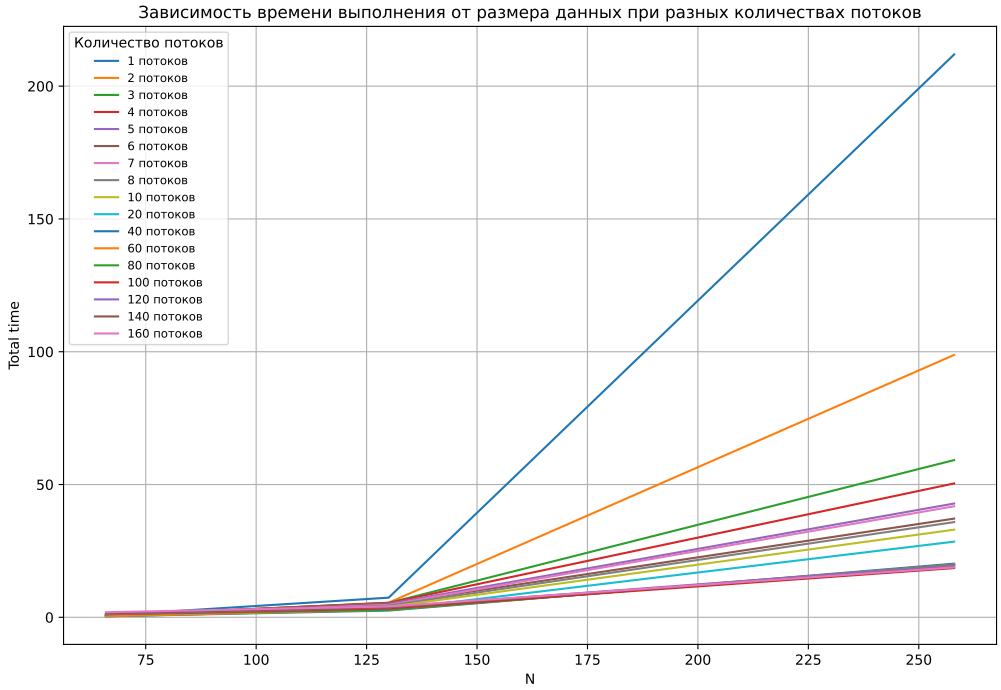

Рис. 3: Зависимость времени выполнения от размера данных при разных количествах потоков

Таблица 2:

Исходные данные для построения графика зависимости времени выполнения от размера данных

Количество потоков	**Размер данных** $N = 66$ **(сек)**	**Размер данных** $N = 130$ **(сек)**	**Размер данных** $N = 258$ **(сек)**
1	0.708305	7.396162	211.929427
2	0.587093	5.510503	98.795370
3	0.583170	5.426540	59.203935
4	0.618750	5.413721	50.407291
5	0.599654	5.160032	42.838277
6	0.591524	4.982432	37.162253
7	0.531227	4.726086	41.810504
8	0.552342	4.388342	35.847319
10	0.487627	3.935538	32.990397
20	0.395729	2.844497	28.488687
40	0.375469	2.539852	20.196651
60	0.399366	2.818270	19.855609
80	1.154532	2.792874	19.762702
100	1.185313	3.323867	18.531570
120	1.227761	3.772425	19.550416
140	1.509529	3.816676	18.887787
160	1.911405	4.088450	18.815091

4.2 Директива task

4.2.1 Отличия от оригинальной

В программе ‘var37_task.c‘ с использованием OpenMP реализованы следующие изменения:

- **Инициализация массива** (`init()`): Параллелизация трёх вложенных циклов выполнена с использованием `#pragma omp parallel for collapse(3)`, что позволяет равномерно распределить нагрузку по потокам.
- **Расслабление массива** (`relax()`):
  - Вместо циклов `for` используется `#pragma omp task` для создания независимых задач на каждом этапе.
  - После каждой группы задач выполняется синхронизация с помощью `#pragma omp taskwait`.
- **Обновление ошибки** `eps`:
  - Локальная переменная `local_eps` используется для локального вычисления изменения.
  - Обновление глобальной переменной `eps` синхронизируется с использованием `#pragma omp critical`.
- **Функция проверки** (`verify()`): Выполнена параллелизация с использованием `#pragma omp parallel for collapse(3)` и редукции (`reduction(+:s)`), что ускоряет подсчёт итоговой суммы.

Краткое описание директив OpenMP:

- `#pragma omp parallel for`: Создаёт параллельные потоки для выполнения цикла. Каждый поток выполняет определённую часть итераций.
- `collapse(3)`: Объединяет три вложенных цикла в один, чтобы равномерно распределить итерации между потоками.
- `#pragma omp task`: Определяет отдельную задачу, которую могут выполнять потоки независимо друг от друга.
- `#pragma omp taskwait`: Синхронизирует выполнение задач, заставляя поток ожидать завершения всех задач перед продолжением.
- `#pragma omp critical`: Гарантирует, что секция кода внутри этой директивы будет выполнена только одним потоком за раз, предотвращая гонки данных.
- `reduction(+:variable)`: Суммирует значения переменной `variable`, вычисленные в каждом потоке, и объединяет их в финальное значение.

4.2.2 Код программы

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

#define  Max(a, b) ((a)>(b)?(a):(b))

#define  N   (256)
float maxeps = 0.1e-7;
int itmax = 100;
int i, j, k;

float eps;
float A[N][N][N];

void relax();
void init();
void verify();

int main(int an, char **as) {
    int it;
    double start, end;
    printf("OMP_NUM_THREADS = %d\n: ", omp_get_max_threads());
    start=omp_get_wtime();
    init();

    for (it = 1; it <= itmax; it++) {
        eps = 0.;

        relax();

        if (eps < maxeps) break;
    }

    verify();
    end=omp_get_wtime();
    printf("TOTAL_TIME: %f\n", end - start);
    return 0;
}


void init() {
#pragma omp parallel for private(i, j, k) collapse(3)
    for (k = 0; k <= N - 1; k++)
        for (j = 0; j <= N - 1; j++)
            for (i = 0; i <= N - 1; i++) {
                if (i == 0 || i == N - 1 || j == 0 ||
                        j == N - 1 || k == 0 || k == N - 1)
                    A[i][j][k] = 0.;
```

```
                else
                    A[i][j][k] = (4. + i + j + k);
            }
}

void relax() {
    eps = 0.0;

#pragma omp parallel
    {
#pragma omp single
        {
            int num_threads = omp_get_num_threads();
            int i_len = (N-4);
            int i_chunk = i_len / num_threads;
            int i_remainder = i_len % num_threads;
            {
                int start_i = 2;
                for (int t = 0; t < num_threads; t++) {
                    int end_i = start_i + i_chunk - 1;
                    if (t < i_remainder) {
                        end_i += 1;
                    }
                    int current_start = start_i;
                    int current_end = end_i;
                    start_i = end_i + 1;

#pragma omp task firstprivate(current_start, current_end)
                    private(i,j,k) shared(A)
                    {
                        for (int k = 1; k <= N - 2; k++)
                            for (int j = 1; j <= N - 2; j++)
                                for (int i = current_start;
                                    i <= current_end; i++) {
                                    A[i][j][k] = (A[i-1][j][k] +
                                        A[i+1][j][k] + A[i-2][j][k]
                                        + A[i+2][j][k]) / 4.0;
                                }
                    }
                }
            }
#pragma omp taskwait
            {
                int j_len = (N-4);
                int j_chunk = j_len / num_threads;
                int j_remainder = j_len % num_threads;

                int start_j = 2;
                for (int t = 0; t < num_threads; t++) {
                    int end_j = start_j + j_chunk - 1;
                    if (t < j_remainder) {
```

```
                        end_j += 1;
                    }
                    int current_start = start_j;
                    int current_end = end_j;
                    start_j = end_j + 1;

#pragma omp task firstprivate(current_start, current_end)
        private(i,j,k) shared(A)
                    {
                        for (int k = 1; k <= N - 2; k++)
                            for (int j = current_start;
                                j <= current_end; j++)
                                for (int i = 1; i <= N - 2; i++) {
                                    A[i][j][k] = (A[i][j-1][k] +
                                    A[i][j+1][k] + A[i][j-2][k]
                                    + A[i][j+2][k]) / 4.0;
                                }
                    }
                }
            }
#pragma omp taskwait
            {
                int k_len = (N-4);
                int k_chunk = k_len / num_threads;
                int k_remainder = k_len % num_threads;

                int start_k = 2;
                for (int t = 0; t < num_threads; t++) {
                    int end_k = start_k + k_chunk - 1;
                    if (t < k_remainder) {
                        end_k += 1;
                    }
                    int current_start = start_k;
                    int current_end = end_k;
                    start_k = end_k + 1;

#pragma omp task firstprivate(current_start, current_end)
        private(i,j,k) shared(A, eps)
                    {
                        float local_eps = 0.0f;
                        for (int k = current_start;
                            k <= current_end; k++)
                            for (int j = 1; j <= N - 2; j++)
                                for (int i = 1; i <= N - 2; i++) {
                                    float e = A[i][j][k];
                                    A[i][j][k] = (A[i][j][k-1] +
                                    A[i][j][k+1] + A[i][j][k-2]
                                    + A[i][j][k+2]) / 4.0;
                                    float diff = fabs(e -
                                    A[i][j][k]);
                                    if (diff > local_eps)
```

```
                                        local_eps = diff;
                                    }
#pragma omp critical
                            {
                                eps = Max(eps, local_eps);
                            }
                        }
                    }
                }
#pragma omp taskwait
            }
        }
}

void verify() {
    float s;

    s = 0.;

#pragma omp parallel for private(i, j, k) collapse(3)
            reduction(+:s)
    for (k = 0; k <= N - 1; k++)
        for (j = 0; j <= N - 1; j++)
            for (i = 0; i <= N - 1; i++) {
                s = s + A[i][j][k] * (i + 1) * (j + 1) * (k + 1) /
                        (N * N * N);
            }
    printf("  S = %f\n", s);
}
```

Листинг 3: Код программы var37_task.c

4.2.3 Графическое представление результатов

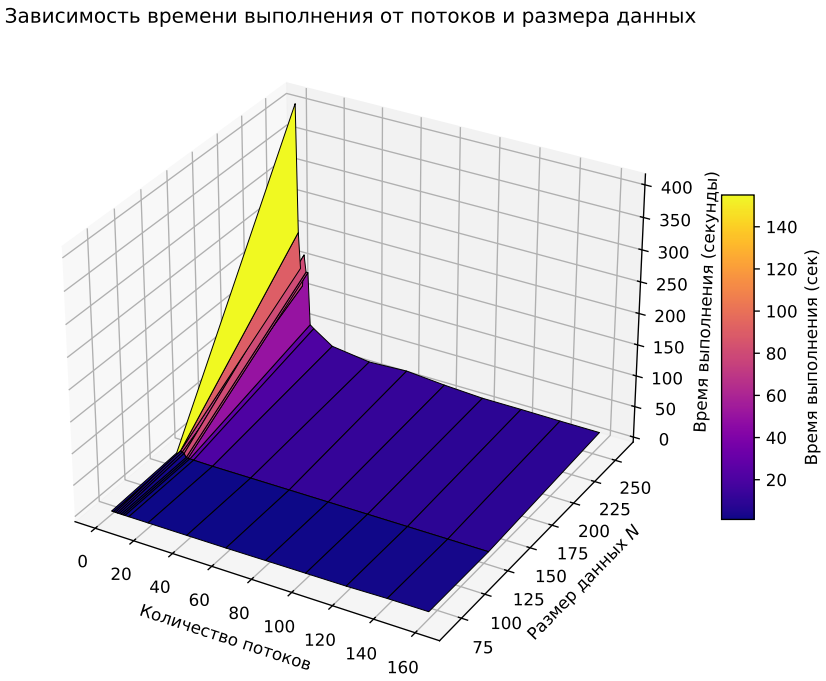

Рис. 4: Зависимость времени выполнения от количества потоков и размера данных (task)

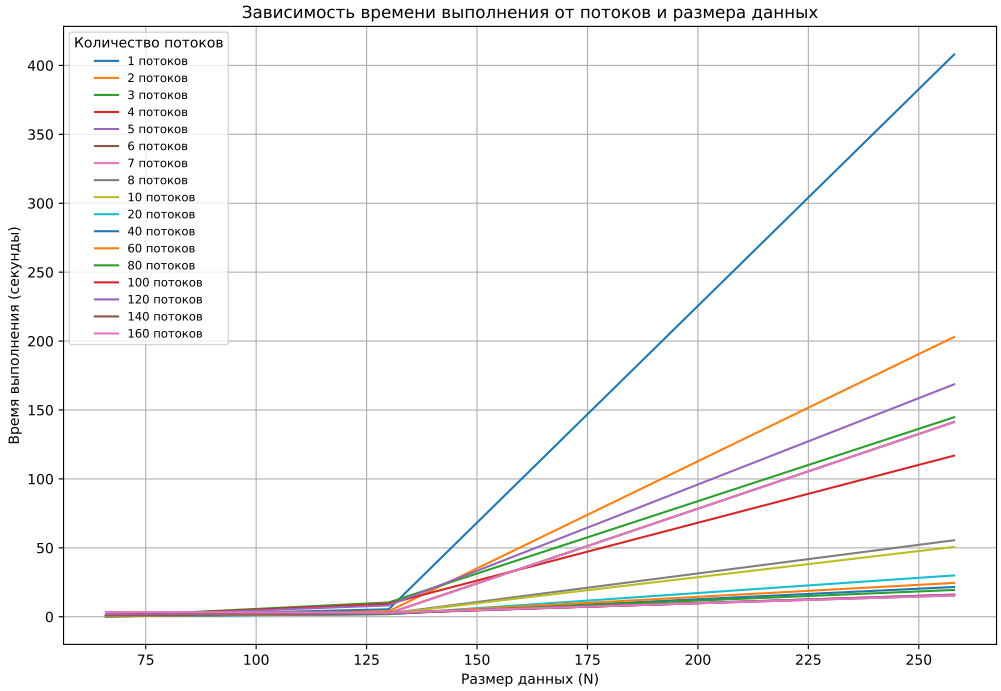

Рис. 5: Зависимость времени выполнения от размера данных для различных потоков (task)

Таблица 3: **Исходные данные для построения графика зависимости времени выполнения от размера данных (task)**

Количество потоков	**Размер данных** $N = 66$ **(сек)**	**Размер данных** $N = 130$ **(сек)**	**Размер данных** $N = 258$ **(сек)**
1	0.544581	5.405589	407.918048
2	0.402061	3.994932	202.985691
3	0.371196	10.367265	144.747997
4	0.345222	9.446811	116.894496
5	0.349610	8.280343	168.615217
6	0.358028	2.549414	141.292341
7	0.344900	2.466811	141.179076
8	0.330124	2.398253	55.499396
10	0.325617	2.235469	50.689523
20	0.348759	1.822643	29.997787
40	0.419808	2.033462	21.578118
60	0.492951	2.258384	24.542644
80	0.973725	2.420332	19.441177
100	1.349136	2.537475	15.994226
120	1.703085	2.715005	16.020607
140	3.167416	2.872127	15.517862
160	3.453633	2.953421	15.422939

4.3 Выводы и сравнения OMP программ с оригинальной

В данной работе были реализованы и протестированы две версии программы с использованием технологий OpenMP: `for` для параллельного выполнения итераций циклов и `task` для динамического распределения задач. На основе проведённых экспериментов можно сделать следующие выводы:

4.3.1 Сравнение производительности на разном количестве потоков

- При небольшом количестве потоков (1–4) обе версии демонстрируют схожее время выполнения, так как параллелизм на таком уровне не полностью используется. В этом режиме накладные расходы на управление задачами в версии `task` могут снижать её эффективность по сравнению с `for`.

- С увеличением количества потоков версия с `for` демонстрирует более стабильное поведение, так как равномерное распределение итераций минимизирует накладные расходы. Однако на больших данных ($N = 258$) задачи с `task` начинают показывать более высокую производительность за счёт гибкого управления нагрузкой.

- При большом количестве потоков (> 20) версия `for` начинает показывать меньшую эффективность на малых размерах данных ($N = 66, 130$), так как распределение задач становится недостаточно оптимальным. Версия с `task` в таких случаях лучше адаптируется, но также теряет производительность из-за накладных расходов.

4.3.2 Сравнение производительности на разных размерах данных

- На малых данных ($N = 66$) обе версии показывают примерно одинаковую производительность на небольшом количестве потоков. Однако при увеличении числа потоков версия `task` демонстрирует больше накладных расходов, что снижает её эффективность.

- На средних данных ($N = 130$) версии начинают различаться сильнее. `for` остаётся стабильным при увеличении потоков, тогда как `task` может столкнуться с уменьшением производительности из-за увеличения числа мелких задач, которые не всегда эффективно распределяются.

- На больших данных ($N = 258$) версия с `task` начинает превосходить `for`, так как задачи динамически распределяются по потокам, что особенно эффективно при неоднородной нагрузке.

4.3.3 Общие выводы

- Версия с `for` лучше подходит для небольших и средних размеров данных, особенно когда количество потоков не превышает 20. Она обеспечивает стабильное распределение итераций и минимальные накладные расходы.

- Версия с `task` проявляет себя лучше на больших данных ($N = 258$) и при большом количестве потоков (> 20), так как гибкость задач позволяет эффективно использовать ресурсы, несмотря на накладные расходы.

- Оптимальный выбор версии зависит от характера задачи и доступных вычислительных ресурсов. Для предсказуемых циклов и равномерной нагрузки предпочтительнее использовать `for`, тогда как для задач с непредсказуемой нагрузкой и большим количеством данных лучше подходит `task`.

5 Использование MPI

5.1 Отличия от оригинальной программы

В программе с использованием MPI были реализованы следующие изменения:

- Распределение данных между процессами:
  - Массив A разбивается на участки по размерности k, которые обрабатываются отдельными процессами.
  - Каждый процесс работает только с локальным буфером A_local, соответствующим его участку данных.
- Обмен граничными слоями:
  - Используются два гало-слоя (`halo_up` и `halo_down`) для обмена граничными данными между соседними процессами.
  - Обмен выполняется с помощью функции `MPI_Sendrecv()`.
- Вычисление сходимости:
  - Локальная ошибка eps рассчитывается каждым процессом, а затем объединяется в глобальную ошибку с использованием `MPI_Allreduce()`.
- Финальная проверка результатов:
  - Каждый процесс рассчитывает локальную сумму s_local, а затем результаты объединяются с помощью `MPI_Reduce()`.

Краткое описание функций MPI:

- `MPI_Init()` и `MPI_Finalize()`: Инициализация и завершение MPI.
- `MPI_Comm_rank()` и `MPI_Comm_size()`: Определяют номер процесса (`rank`) и общее количество процессов (`size`).
- `MPI_Sendrecv()`: Обеспечивает обмен гало-слоями между соседними процессами.
- `MPI_Allreduce()`: Вычисляет глобальное значение ошибки eps путём объединения локальных значений.
- `MPI_Reduce()`: Суммирует локальные значения итоговой суммы s и передаёт результат корневому процессу.

5.2 Код программы

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

#define  Max(a,b) ((a)>(b)?(a):(b))
#define  Min(a,b) ((a)<(b)?(a):(b))
#define  N   (2*2*2*2*2*2+2)
#define TAG_DOWN 0
#define TAG_UP   1

float maxeps = 0.1e-7;
int itmax = 100;

float eps;
float A_local[N][N][N];

float halo_down[2][N][N];
float halo_up[2][N][N];

void relax(int rank, int size, int k_start, int k_end);
void init(int rank, int size, int k_start, int k_end);
void verify(int rank, int size, int k_start, int k_end);
void exchange_halos(int rank, int size, int k_start, int k_end);

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int base = N / size;
    int rem = N % size;
    int local_K;
    int k_start, k_end;
    if (rank < rem) {
        local_K = base + 1;
        k_start = rank * local_K;
    } else {
        local_K = base;
        k_start = rem * (base + 1) + (rank - rem) * base;
    }
    k_end = k_start + local_K - 1;

    double program_end_time, program_start_time = omp_get_wtime();

    init(rank, size, k_start, k_end);
```

```
50
51      for(int it = 1; it <= itmax; it++)
52      {
53          eps = 0.0f;
54          relax(rank, size, k_start, k_end);
55
56          float global_eps;
57          MPI_Allreduce(&eps, &global_eps, 1, MPI_FLOAT,
58          MPI_MAX, MPI_COMM_WORLD);
59          eps = global_eps;
60          if (eps < maxeps) break;
61      }
62
63      verify(rank, size, k_start, k_end);
64
65      program_end_time = omp_get_wtime();
66      if(rank == 0) printf("TOTAL_TIME: %f\n",
67      program_end_time - program_start_time);
68
69      MPI_Finalize();
70      return 0;
71  }
72
73  void init(int rank, int size, int k_start, int k_end)
74  {
75      for(int kk = k_start; kk <= k_end; kk++) {
76          int k_local = kk - k_start;
77
78          if (k_local < 0 || k_local >= N) {
79              MPI_Abort(MPI_COMM_WORLD, 1);
80          }
81
82          for(int j = 0; j < N; j++) {
83              for(int i = 0; i < N; i++) {
84                  if(i == 0 || i == N-1 || j == 0 ||
85                  j == N-1 || kk == 0 || kk == N-1)
86                      A_local[i][j][k_local] = 0.0f;
87                  else
88                      A_local[i][j][k_local] = 4.0f + i + j + kk;
89              }
90          }
91      }
92
93  }
94
95  void exchange_halos(int rank, int size, int k_start, int k_end)
96  {
97      MPI_Status status;
98      int left = (rank == 0) ? MPI_PROC_NULL : rank - 1;
99      int right = (rank == size - 1) ? MPI_PROC_NULL : rank + 1;
100
```

```
    int local_K = k_end - k_start + 1;

    float send_down[2][N][N];
    float send_up[2][N][N];

    for(int j = 0; j < N; j++) {
        for(int i = 0; i < N; i++) {
            if(local_K >=2){
                send_down[0][i][j] = A_local[i][j][0];
                send_down[1][i][j] = A_local[i][j][1];

                send_up[0][i][j] = A_local[i][j][local_K - 2];
                send_up[1][i][j] = A_local[i][j][local_K - 1];
            }
            else{
                send_down[0][i][j] = 0.0f;
                send_down[1][i][j] = 0.0f;
                send_up[0][i][j] = 0.0f;
                send_up[1][i][j] = 0.0f;
            }
        }
    }

    MPI_Sendrecv(send_down, 2 * N * N, MPI_FLOAT, left, TAG_DOWN,
                 halo_up, 2 * N * N, MPI_FLOAT, left, TAG_UP,
                 MPI_COMM_WORLD, &status);

    MPI_Sendrecv(send_up, 2 * N * N, MPI_FLOAT, right, TAG_UP,
                 halo_down, 2 * N * N, MPI_FLOAT, right, TAG_DOWN,
                 MPI_COMM_WORLD, &status);

}

void relax(int rank, int size, int k_start, int k_end)
{
    int local_K = k_end - k_start +1;

    for(int kk = Max(k_start,1); kk <= Min(k_end, N-2); kk++) {
        if(local_K >=1){
            int k_local = kk - k_start;
            for(int j = 1; j <= N-2; j++) {
                for(int i = 2; i <= N-3; i++) {
                    A_local[i][j][k_local] =
                    (A_local[i-1][j][k_local] +
                    A_local[i+1][j][k_local] +
                    A_local[i-2][j][k_local] +
                    A_local[i+2][j][k_local]) /
                    4.0f;
                }
            }
        }
```

```
152     }
153     for(int kk = Max(k_start,1); kk <= Min(k_end, N-2); kk++) {
154         if(local_K >=1){
155             int k_local = kk - k_start;
156             for(int j = 2; j <= N-3; j++) {
157                 for(int i = 1; i <= N-2; i++) {
158                     A_local[i][j][k_local] =
159                     (A_local[i][j-1][k_local] +
160                     A_local[i][j+1][k_local] +
161                     A_local[i][j-2][k_local] +
162                     A_local[i][j+2][k_local])
163                     / 4.0f;
164                 }
165             }
166         }
167     }
168 
169     if(local_K >=5){
170         exchange_halos(rank, size, k_start, k_end);
171 
172         for(int kk = Max(k_start,2); kk <= Min(k_end, N-3); kk++) {
173             int k_local = kk - k_start;
174             if(k_local <2 || k_local > (local_K -3)){
175                 continue;
176             }
177 
178             for(int j =1; j <=N-2; j++) {
179                 for(int i=1; i <=N-2; i++) {
180                     float e = A_local[i][j][k_local];
181                     float a_km1 = (k_local -1 >=0) ?
182                     A_local[i][j][k_local -1] : halo_down[0][i][j];
183                     float a_km2 = (k_local -2 >=0) ?
184                     A_local[i][j][k_local -2] : halo_down[1][i][j];
185                     float a_kp1 = (k_local +1 < local_K) ?
186                     A_local[i][j][k_local +1] : halo_up[0][i][j];
187                     float a_kp2 = (k_local +2 < local_K) ?
188                     A_local[i][j][k_local +2] : halo_up[1][i][j];
189                     A_local[i][j][k_local] =
190                     (a_km1 + a_kp1 + a_km2 + a_kp2) / 4.0f;
191                     float diff = fabsf(e - A_local[i][j][k_local]);
192                     eps = Max(eps, diff);
193                 }
194             }
195         }
196     }
197 }
198 
199 void verify(int rank, int size, int k_start, int k_end)
200 {
201     float s_local = 0.0f;
202     int local_K = k_end - k_start +1;
```

```
203
204     for(int kk = k_start; kk <= k_end; kk++) {
205         if(local_K >=1){
206             int k_local = kk - k_start;
207             if (k_local <0 || k_local >= N){
208                 MPI_Abort(MPI_COMM_WORLD, 1);
209             }
210
211             for(int j = 0; j < N; j++) {
212                 for(int i = 0; i < N; i++) {
213                     if(i <0 || i >=N || j <0 || j >=N){
214                         MPI_Abort(MPI_COMM_WORLD, 1);
215                     }
216
217                     s_local += A_local[i][j][k_local] * (i + 1)
218                     * (j + 1) * (kk + 1) / (float)(N * N * N);
219                 }
220             }
221         }
222     }
223
224     float s;
225     MPI_Reduce(&s_local, &s, 1, MPI_FLOAT, MPI_SUM,
226             0, MPI_COMM_WORLD);
227
228     if(rank == 0) {
229         printf("  S = %f\n", s);
230     }
231 }
```

Листинг 4: Код программы var37_mpi.c

5.3 Графическое представление результатов

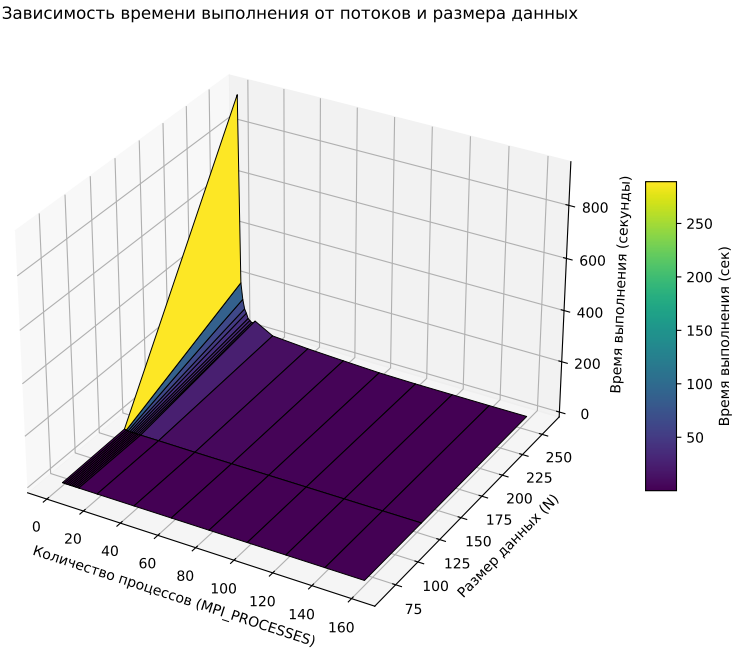

Рис. 6: Зависимость времени выполнения от количества процессов и размера данных (MPI)

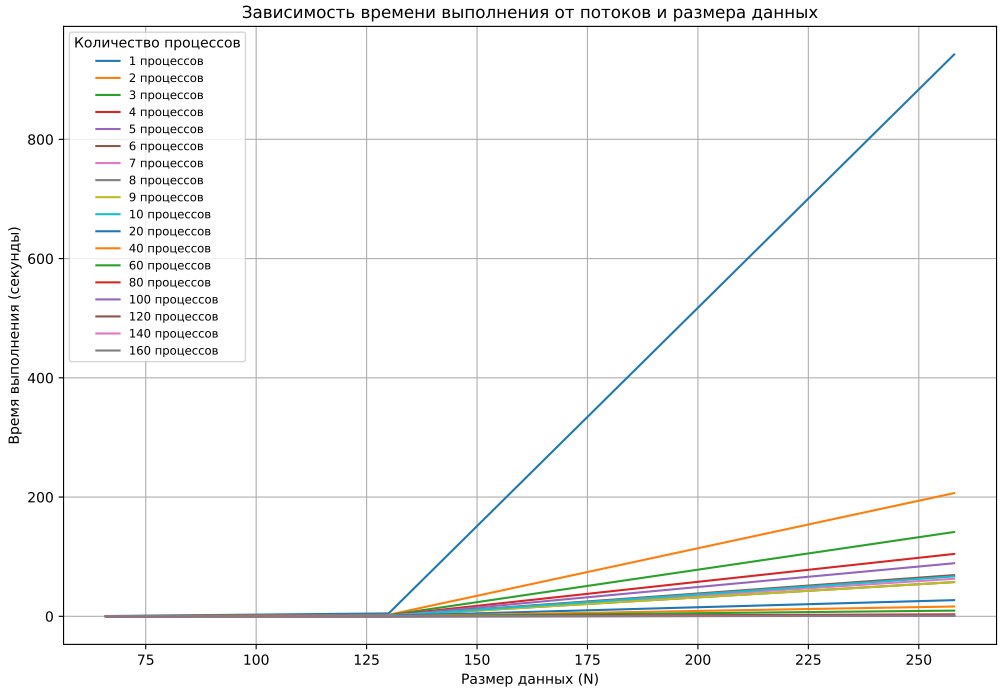

Рис. 7: Зависимость времени выполнения от размера данных для различных процессов (MPI)

Таблица 4: **Исходные данные для построения графиков времени выполнения (MPI)**

Количество процессов	**Размер данных** $N = 66$ **(сек)**	**Размер данных** $N = 130$ **(сек)**	**Размер данных** $N = 258$ **(сек)**
1	0.474961	4.860682	942.309633
2	0.237137	2.535721	206.636641
3	0.162464	1.947022	141.407818
4	0.168469	1.440570	104.602865
5	0.137883	1.022919	88.998342
6	0.157951	1.028949	69.100864
7	0.135964	1.018184	62.949256
8	0.145587	0.844532	57.393863
9	0.131674	0.8683310	57.312884
10	0.088982	0.706126	66.851185
20	0.004070	0.859691	27.099570
40	0.002642	0.023487	16.456826
60	0.001820	0.022115	9.548422
80	0.002042	0.020791	3.322462
100	0.001661	0.025835	1.235991
120	0.056755	0.030338	0.939643
140	0.025243	0.045007	0.576664
160	0.057330	0.045989	0.849541

5.4 Выводы и сравнения MPI программы с оригинальной

В данной работе была реализована версия программы с использованием технологии MPI для параллельной обработки данных. Сравнение производительности MPI программы и оригинальной последовательной версии позволяет сделать следующие выводы:

5.4.1 Сравнение производительности на разном количестве процессов

- При небольшом количестве процессов (1–4) MPI программа показывает улучшение производительности по сравнению с оригинальной программой за счёт частичного распараллеливания вычислений. Однако накладные расходы на обмен данными между процессами при $N = 66$ минимальны, что делает улучшение менее заметным.

- С увеличением числа процессов MPI программа демонстрирует значительное сокращение времени выполнения, особенно для больших данных ($N = 258$). Это связано с равномерным распределением нагрузки между процессами.

- При использовании большого числа процессов (> 40) для малых данных ($N = 66$) эффективность MPI программы начинает снижаться из-за роста накладных расходов на коммуникацию. На больших данных ($N = 258$) это влияние минимально.

5.4.2 Сравнение производительности на разных размерах данных

- На малых данных ($N = 66$) преимущества MPI программы заметны только при среднем количестве процессов ($10 \breve{} 20$). При большом числе процессов накладные расходы становятся значительными, и прирост производительности замедляется.

- На средних данных ($N = 130$) программа демонстрирует хорошую масштабируемость при увеличении количества процессов до $40 \breve{} 60$. Это указывает на эффективное использование ресурсов.

- На больших данных ($N = 258$) MPI программа показывает максимальную производительность при использовании большого числа процессов (> 100). Время выполнения сокращается более чем в 500 раз по сравнению с оригинальной программой.

5.4.3 Общие выводы

- MPI программа значительно превосходит оригинальную последовательную версию по производительности, особенно на больших данных ($N = 258$) и при использовании более чем 20 процессов.

- Основные преимущества MPI реализации:

  – Эффективное распределение данных между процессами.

  – Минимизация накладных расходов за счёт использования обмена граничными слоями (`halo_up` и `halo_down`).

  - Гибкость в обработке больших массивов данных.

- Основные ограничения MPI реализации:
  - Накладные расходы на коммуникацию между процессами становятся значительными при работе с малыми данными ($N = 66$) и большим числом процессов.
  - Эффективность MPI программы зависит от равномерного распределения задач между процессами.

- Таким образом, MPI программа является более предпочтительным решением для задач с большими объёмами данных или требующих высокой масштабируемости.

6 Финальные выводы

В данной работе были разработаны и протестированы три версии программы для решения задачи численного моделирования: оригинальная последовательная, параллельная с использованием OpenMP (`for` и `task`) и параллельная с использованием MPI. Проведённый анализ позволяет сделать следующие выводы:

6.1 Когда все программы работают хорошо

- **Малое количество потоков или процессов (1–4):** На небольшом количестве потоков (`OMP`) или процессов (`MPI`) все программы показывают хорошую производительность. Параллельные версии (`OMP for`, `OMP task`, `MPI`) имеют небольшие накладные расходы и начинают демонстрировать выигрыш по сравнению с оригинальной программой.

- **Средние данные ($N = 130$):** Для OpenMP (`for`) и MPI реализаций наблюдается значительное сокращение времени выполнения, особенно при 10–20 потоках или процессах.

6.2 Когда все программы работают плохо

- **Малые данные ($N = 66$) и большое число потоков или процессов (> 40):** Все программы теряют эффективность из-за увеличения накладных расходов.
  - В OpenMP (`task`) накладные расходы на создание мелких задач начинают доминировать, что снижает производительность.
  - В MPI программах коммуникация между процессами и обмен граничными слоями (`halo`) создают значительные задержки.

- **Очень большое количество потоков или процессов (> 100):** Для OpenMP (`for`) и MPI на малых данных накладные расходы становятся больше, чем выигрыш от параллелизации, особенно если ресурсы процессора не используются равномерно.

6.3 Когда одна из программ лучше

- **OpenMP (`for`):**
  - Лучше всего подходит для малых и средних данных ($N = 66$, $N = 130$), особенно при количестве потоков < 20. Она демонстрирует стабильное распределение итераций и минимальные накладные расходы.
  - Неэффективна для больших данных ($N = 258$), так как равномерное распределение нагрузки не всегда оптимально.
- **OpenMP (`task`):**
  - Лучше всего справляется с большими данными ($N = 258$) при среднем количестве потоков ($20\tilde{}40$). Гибкость задач позволяет эффективно использовать ресурсы.
  - Становится менее эффективной при большом количестве потоков (> 40) на малых данных ($N = 66$).
- **MPI:**
  - Значительно превосходит OpenMP версии на больших данных ($N = 258$) при > 40 процессах. Эффективное распределение данных и обмен граничными слоями позволяют достичь высокой производительности.
  - Теряет эффективность на малых данных ($N = 66$) при большом количестве процессов из-за значительных накладных расходов на коммуникацию.

6.4 Общие выводы

- OpenMP (`for`) и OpenMP (`task`) лучше подходят для задач с предсказуемой нагрузкой и средними объёмами данных ($N = 66$, $N = 130$). Эти версии проще в реализации и имеют меньшие накладные расходы при правильной настройке.
- MPI показывает наилучшую масштабируемость и эффективность на больших данных ($N = 258$), особенно при большом количестве процессов (> 40).
- Выбор подходящей технологии зависит от размера задачи и доступных вычислительных ресурсов:
  - Для задач с малыми и средними данными, требующих быстрого решения, предпочтительно использовать OpenMP (`for`).
  - Для сложных задач с большими данными, требующих высокой масштабируемости, рекомендуется использовать MPI.

7 Приложения

В рамках работы были разработаны следующие программы, которые прилагаются к отчёту:

- `var37.c`: Оригинальная последовательная версия программы для решения трёхмерного уравнения Лапласа с использованием метода Якоби.
- `var37_for.c`: Версия программы с использованием OpenMP и директивы `for` для параллельного выполнения вложенных циклов.
- `var37_task.c`: Версия программы с использованием OpenMP и директивы `task` для создания независимых задач и их динамического распределения между потоками.
- `var37_mpi.c`: Версия программы с использованием MPI для распределения данных между процессами, обмена граничными слоями и параллельного выполнения итераций.