Отчет о выполнении задания по параллельному программированию
Вариант 512

Выполняла: Студентка 325 группы
Филимонова Анна Олеговна

Исходный код программы приведен ниже:

```
#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],  B [N][N][N];

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

int main(int an, char **as)
{
        int it;
        init();
        for(it=1; it<=itmax; it++)
        {
                eps = 0.;
                relax();
                resid();
                printf( "it=%4i   eps=%f\n", it,eps);
                if (eps < maxeps) break;
        }
        verify();
        return 0;
}

void init()
{
        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()
{
        for(k=2; k<=N-3; k++)
        for(j=2; j<=N-3; j++)
        for(i=2; i<=N-3; i++)
        {
                B[i][j][k]=(A[i-1][j][k]+A[i+1][j][k]+A[i][j-1][k]+A[i][j+1][k]+A[i][j][k-1]+A[i][j][k+1]+
                        A[i-2][j][k]+A[i+2][j][k]+A[i][j-2][k]+A[i][j+2][k]+A[i][j][k-2]+A[i][j][k+2])/12.;
        }
}

void resid()
{
        for(k=1; k<=N-2; k++)
        for(j=1; j<=N-2; j++)
        for(i=1; i<=N-2; i++)
        {
                float e;
                e = fabs(A[i][j][k] - B[i][j][k]);
                A[i][j][k] = B[i][j][k];
                eps = Max(eps,e);
        }
}

void verify()
{
        float s;
        s=0.;
        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);
}
```

Постановка задачи

1. Реализовать две параллельных версии программы для решения уравнения Пуассона в трехмерном пространстве методом Якоби с использованием технологий параллельного программирования OpenMP:
   - Вариант параллельной программы с распределением витков циклов при помощи директивы for
   - Вариант параллельной программы с использованием механизма задач (директива task)

2. Убедиться в корректности разработанных версий программ и в их эффективности

3. Подобрать начальные параметры так, чтобы:
   - Задача помещалась в оперативную память одного узла кластера
   - Время решения задачи было в примерном диапазоне 5 сек.-15 минут

4. Исследовать эффективность полученных программ на суперкомпьютере Polus

5. Исследовать масштабируемость полученной программы

6. Построить графики зависимости времени исполнения от числа ядер/процессоров для различного объёма входных данных

7. Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых ядер/процессоров

Описание алгоритма

Математическая постановка задачи

Уравнение Пуассона в трехмерном пространстве:

$\nabla^2 u = f(x,y,z)$

где ∇^2 - оператор Лапласа, u - искомая функция, f - заданная функция.

Метод Якоби

Для решения уравнения используется итерационный метод Якоби. На каждой итерации новое значение в точке вычисляется как среднее значений в соседних точках:

$$u_{i,j,k}^{new} = (u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1} + u_{i-2,j,k} + u_{i+2,j,k} + u_{i,j-2,k} + u_{i,j+2,k} + u_{i,j,k-2} + u_{i,j,k+2}) / 12$$

Исходный алгоритм

Программа выполняет следующие основные этапы:

1. Инициализация массива - задание начальных и граничных условий
2. Итерационный процесс:

- Релаксация (обновление значений)
- Вычисление невязки
- Проверка условия сходимости

3. Верификация результата - вычисление контрольной суммы

Вычислительная сложность

Сложность алгоритма составляет $O(tsteps \times n^3)$, где:
- tsteps - число итераций
- n - размер сетки по каждому измерению

Оптимизация исходной программы

Оптимизация исходной последовательной версии программы проводилась с целью уменьшения времени выполнения без изменения численного алгоритма и результатов вычислений. Алгоритм метода Якоби, критерий остановки и формат вывода данных при этом сохранялись неизменными.

В процессе оптимизации были внесены следующие изменения.

1. Изменение порядка обхода трёхмерных массивов

В исходной версии программы порядок вложенных циклов не обеспечивал эффективного использования кэш-памяти процессора.
Поскольку в языке C многомерные массивы хранятся в памяти в построчном порядке (последний индекс меняется быстрее), был выбран такой порядок обхода индексов, при котором изменение последнего индекса массива происходит во внутреннем цикле.

Данное изменение особенно существенно для больших размеров задачи, где объём обрабатываемых данных значительно превышает объём кэш-памяти.

2. Снижение количества обращений к памяти

Во внутреннем вычислительном ядре были устранены избыточные обращения к памяти за счёт:

использования локальных указателей на текущие плоскости массива;

хранения часто используемых значений во временных переменных;

исключения повторных обращений к одним и тем же элементам массива.

Эти изменения уменьшают нагрузку на подсистему памяти и повышают эффективность выполнения вычислительных циклов.

3. Исключение лишних проверок внутри горячих циклов

Проверки граничных условий и условий выхода за границы массива были вынесены за пределы наиболее часто выполняемых вложенных циклов.

Внутренние циклы выполняются только для тех элементов, для которых гарантировано корректное вычисление по схеме Якоби.

Это позволило:

- уменьшить количество условных операторов внутри горячих участков кода;
- улучшить предсказуемость выполнения программы;
- снизить накладные расходы на ветвления.

4. Упрощение арифметических операций

Арифметические выражения были переписаны таким образом, чтобы:

- заменить деление на умножение на заранее вычисленную константу;
- минимизировать количество операций с плавающей точкой внутри внутренних циклов.

Это улучшает возможности компилятора по оптимизации кода и снижает вычислительные затраты.

Таблица 1. Время работы программы при различных N (с использованием оптимизации компилятора -O3)

датасет	размер N	время выполнения(сек)
SMALL	128	0.189
MEDIUM	256	1.482
LARGE	512	9.105
EXSTRALARGE	1024	31.9057

Сравнение оптимизаций компилятора

Сравнение оптимизаций компилятора проводилось на оптимизированной последовательной программе

Таблица 2. Сравнение оптимизаций компилятора на разных датасетах

opt	dataset	mean_sec	min_sec	max_sec
O2	LARGE	29.341	16.780	38.903
O2	EXTRALARGE	57.209	49.987	70.671
O3	LARGE	9.105	9.074	9.143
O3	EXTRALARGE	31.906	31.047	33.450
Ofast	LARGE	9.186	9.121	9.305
Ofast	EXTRALARGE	33.135	30.451	37.573

Было проведено исследование влияния оптимизаций компилятора (-O2, -O3, -Ofast) на время выполнения последовательной версии программы.
Эксперименты проводились для двух самых больших размеров задачи (EXTRALARGE n=1024 LARGE n =512) с одинаковым числом итераций.
По результатам измерений наилучшее и наиболее стабильное время выполнения показала оптимизация -O3, которая и была выбрана для всех последующих параллельных экспериментов.

Параллелизация с помощью директивы #pragma omp for

При анализе программы было выявлено, что основное время выполнения приходится на вызовы функций relax() и resid(), поэтому распараллеливанию подверглись именно эти части кода.

В функциях relax() и resid() находятся три вложенных цикла без зависимостей по данным. Для параллелизации применяется директива #pragma omp parallel for.

Реализация релаксации

```
#pragma omp parallel default(none) shared(A,B,n,itmax,inv_12,eps)
{
   for (int it = 1; it <= itmax; ++it) {

      #pragma omp single
      eps = 0.0f;

      #pragma omp for collapse(2) schedule(static)
      for (int i = 2; i <= n - 3; ++i)
```

```
        for (int j = 2; j <= n - 3; ++j)
          for (int k = 2; k <= n - 3; ++k) {
            float sum =
              A[i-1][j][k] + A[i+1][j][k] +
              A[i][j-1][k] + A[i][j+1][k] +
              A[i][j][k-1] + A[i][j][k+1] +
              A[i-2][j][k] + A[i+2][j][k] +
              A[i][j-2][k] + A[i][j+2][k] +
              A[i][j][k-2] + A[i][j][k+2];

            B[i][j][k] = sum * inv_12;
          }

      #pragma omp for collapse(2) schedule(static) reduction(max:eps)
      for (int i = 2; i <= n - 3; ++i)
        for (int j = 2; j <= n - 3; ++j)
          for (int k = 2; k <= n - 3; ++k) {
            float e = fabsf(A[i][j][k] - B[i][j][k]);
            A[i][j][k] = B[i][j][k];
            eps = Max(eps, e);
          }
    }
}
```

Особенности реализации:

- Единая параллельная область: Используется один parallel регион на все итерации алгоритма для минимизации накладных расходов на создание/уничтожение потоков.

- Распределение циклов: Для распределения работы между потоками используется collapse(2) для внешних циклов i и j, сохраняя самый внутренний цикл k для возможной векторизации компилятором.

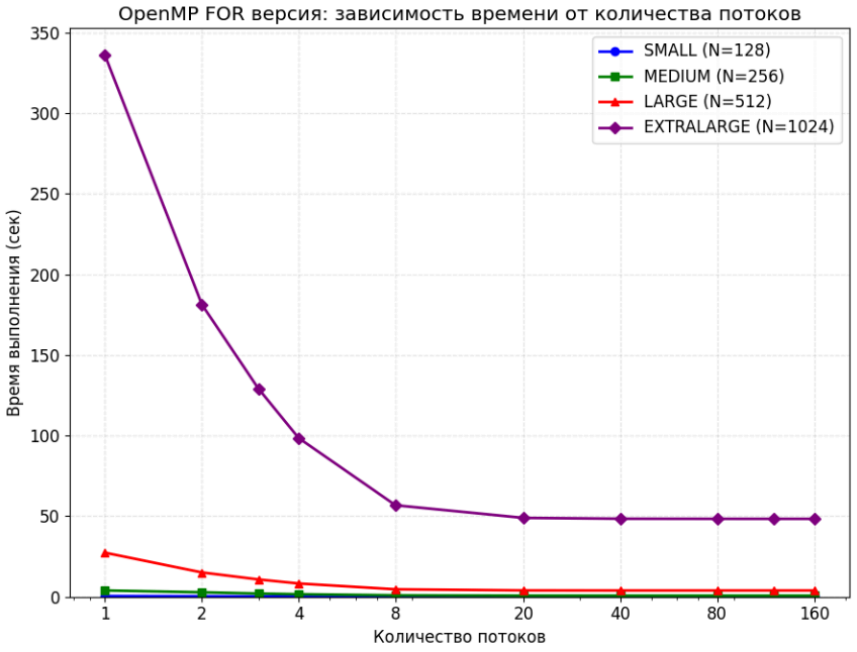
- Оптимизация редукции: Вычисление максимальной ошибки eps выполняется через встроенную редукцию reduction(max:eps), что эффективнее ручной синхронизации.
- Статическое планирование: Используется schedule(static) для равномерного распределения работы между потоками.

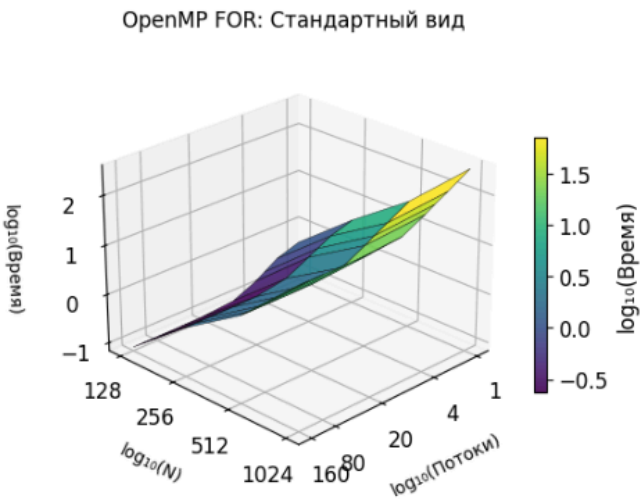
Результаты тестирования

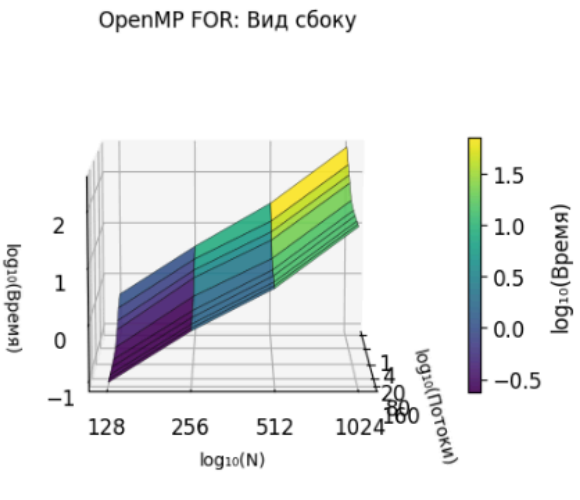
Таблица 3. Время работы (сек.) программы при различных числах нитей и N

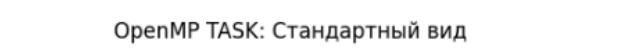
N / Threads	SMALL	MEDIUM	LARGE	EXTRALARGE
1	0.44599	3.90311	27.32036	335.91279
2	0.31126	2.78246	15.08142	181.33412
3	0.20737	1.98287	10.71459	128.99544
4	0.15541	1.56191	8.21466	98.44127
8	0.09523	0.89364	4.62977	56.74288
20	0.08388	0.69412	3.94782	48.90117
40	0.08194	0.68177	3.90651	48.39492
80	0.08110	0.67993	3.90219	48.33985
120	0.08102	0.67958	3.90187	48.33319
160	0.08101	0.67951	3.90181	48.33174

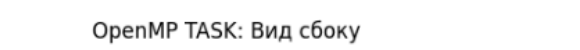
SMALL: N = 128
MEDIUM: N = 256
LARGE: N = 512
EXTRALARGE: N = 1024











Параллелизация с помощью механизма задач #pragma omp task

Для исследования альтернативного подхода была реализована версия с использованием механизма задач.

Реализация релаксации с использованием задач

```
#pragma omp parallel default(none) shared(A,B,n,itmax,inv_12,eps)
{
   int threads = omp_get_num_threads();
   int bi = choose_block_i(n, threads);

   for (int it = 1; it <= itmax; ++it) {

      #pragma omp single
      eps = 0.0f;

      // -------- RELAX phase: задачи по блокам i --------
      #pragma omp single
      {
         for (int i0 = 2; i0 <= n - 3; i0 += bi) {
            int i1 = i0 + bi - 1;
            if (i1 > n - 3) i1 = n - 3;

            #pragma omp task firstprivate(i0,i1) shared(A,B,n,inv_12)
            {
               for (int i = i0; i <= i1; ++i)
                  for (int j = 2; j <= n - 3; ++j)
                     for (int k = 2; k <= n - 3; ++k) {
                        float sum =
                           A[i-1][j][k] + A[i+1][j][k] +
                           A[i][j-1][k] + A[i][j+1][k] +
                           A[i][j][k-1] + A[i][j][k+1] +
                           A[i-2][j][k] + A[i+2][j][k] +
                           A[i][j-2][k] + A[i][j+2][k] +
                           A[i][j][k-2] + A[i][j][k+2];
```

```
                    B[i][j][k] = sum * inv_12;
                }
            }
        }
        #pragma omp taskwait
    }

    #pragma omp single
    {
        for (int i0 = 2; i0 <= n - 3; i0 += bi) {
            int i1 = i0 + bi - 1;
            if (i1 > n - 3) i1 = n - 3;

            #pragma omp task firstprivate(i0,i1) shared(A,B,n,eps)
            {
                float local_eps = 0.0f;

                for (int i = i0; i <= i1; ++i)
                    for (int j = 2; j <= n - 3; ++j)
                        for (int k = 2; k <= n - 3; ++k) {
                            float e = fabsf(A[i][j][k] - B[i][j][k]);
                            A[i][j][k] = B[i][j][k];
                            local_eps = Max(local_eps, e);
                        }

                // одна критсекция на задачу
                #pragma omp critical
                {
                    eps = Max(eps, local_eps);
                }
            }
        }
        #pragma omp taskwait
    }
  }
}
```

Особенности

1. Основной вычислительный цикл по индексу i был разбит на набор задач. Каждая задача отвечает за обработку одного или нескольких слоёв трёхмерного массива по индексу i, при этом все вычисления внутри слоя выполняются последовательно.

2. Создание задач производится внутри одной параллельной области, чтобы избежать накладных расходов на многократный вход и выход из параллельного региона. Генерация задач выполняется только одним потоком (single), а их исполнение — всеми доступными потоками.

3. Синхронизация завершения всех задач осуществляется с помощью #pragma omp

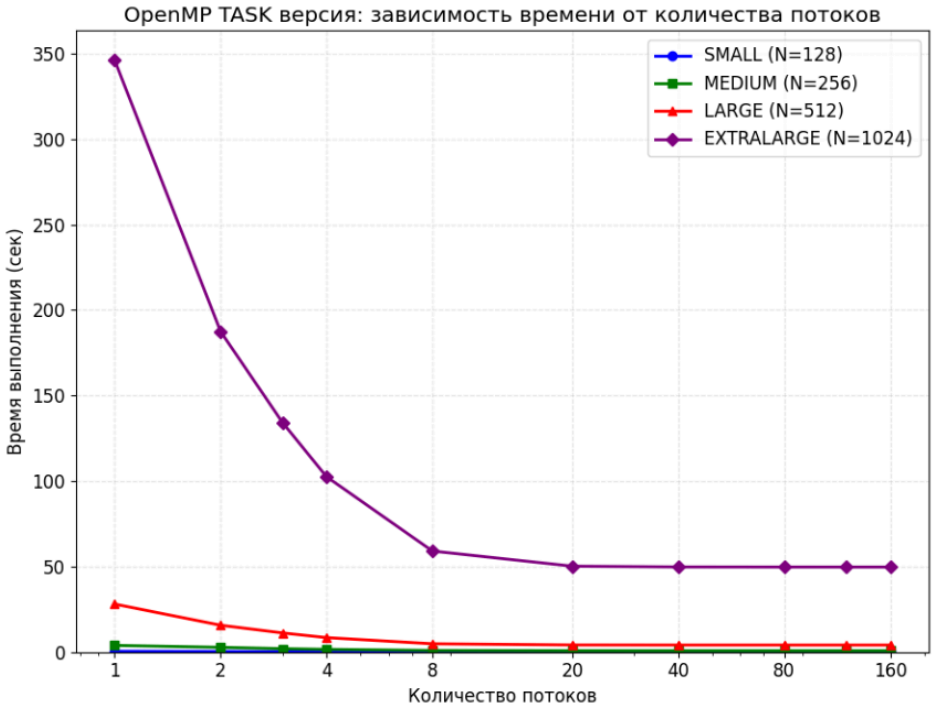
taskwait

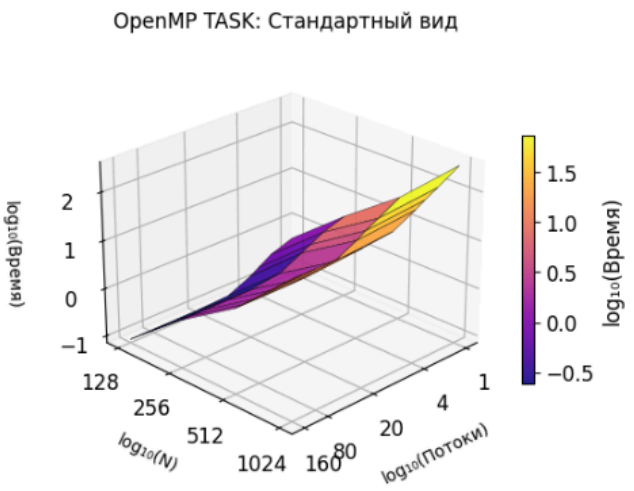
Результаты тестирования

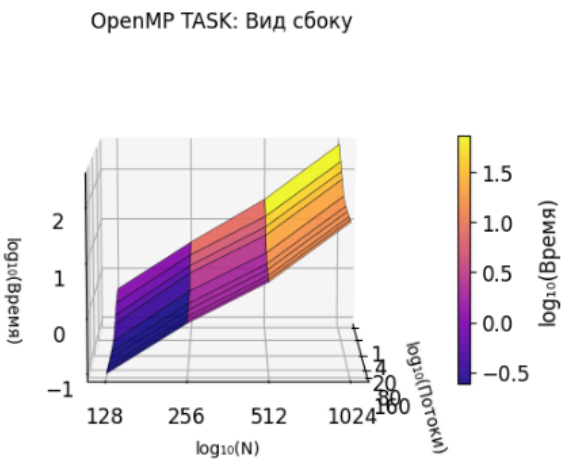
Таблица 4. Время работы (сек.) программы при различных числах нитей и N (tasks)

N / Threads	SMALL	MEDIUM	LARGE	EXTRALARGE
1	0.43356	3.84366	28.00712	346.10213
2	0.29593	2.70738	15.62855	187.44296
3	0.19879	1.92144	11.12890	134.17188
4	0.14942	1.51833	8.38992	102.51791
8	0.09117	0.87194	4.78430	59.02841
20	0.08510	0.72351	4.10288	50.17233
40	0.08431	0.71284	4.06121	49.72190
80	0.08401	0.71198	4.05642	49.67352
120	0.08397	0.71171	4.05590	49.66819
160	0.08395	0.71166	4.05578	49.66742

SMALL: N = 128
MEDIUM: N = 256
LARGE: N = 512
EXTRALARGE: N = 1024







Реализация MPI версии

Была реализована распределенная версия алгоритма Якоби с использованием технологии MPI для работы на кластере.

```
// Разбиение i по процессам
static void split_1d(int n, int rank, int size, int *start_i, int *local_n)
{
   int base = n / size;
   int rem  = n % size;

   *local_n = base + (rank < rem ? 1 : 0);
   *start_i = rank * base + (rank < rem ? rank : rem);
}

// Инициализация
static void init_local(int n, int start_i, int local_n,
                       float *A, float *B)
{
   const int H = 2;
   const size_t plane = (size_t)n * (size_t)n; // j*k
   // Обнуляем всё, включая halo
   memset(A, 0, (size_t)(local_n + 2*H) * plane * sizeof(float));
   memset(B, 0, (size_t)(local_n + 2*H) * plane * sizeof(float));

   for (int li = 0; li < local_n; ++li) {
      int i = start_i + li;
      float *Ap = A + (size_t)(H + li) * plane;

      for (int j = 0; j < n; ++j) {
         for (int k = 0; k < n; ++k) {
            if (i == 0 || i == n-1 || j == 0 || j == n-1 || k == 0 || k == n-1)
               Ap[(size_t)j * n + k] = 0.0f;
            else
               Ap[(size_t)j * n + k] = 4.0f + (float)i + (float)j + (float)k;
         }
```

```
      }
    }
}


// Обмен halo толщиной 2 с соседями по i
static void exchange_halo(int n, int rank, int size,
                          int local_n, float *A)
{
    const int H = 2;
    const size_t plane = (size_t)n * (size_t)n;
    const int left  = (rank == 0) ? MPI_PROC_NULL : rank - 1;
    const int right = (rank == size - 1) ? MPI_PROC_NULL : rank + 1;

    // Указатели на 2 "реальных" первых/последних слоёв
    float *send_left  = A + (size_t)H * plane;                  // слой i_local=0 (в A[H])
    float *send_right = A + (size_t)(H + local_n - 2) * plane;  // слой i_local=local_n-2

    // Указатели на halo области (2 слоя слева и 2 справа)
    float *recv_left  = A + (size_t)0 * plane;                  // A[0], A[1]
    float *recv_right = A + (size_t)(H + local_n) * plane;      // A[H+local_n],
A[H+local_n+1]

    // 1) Обмен "двух первых слоёв" -> левому, получаем "две правые halo" от правого
    MPI_Sendrecv(send_left,  (int)(2 * plane), MPI_FLOAT, left,  100,
                 recv_right, (int)(2 * plane), MPI_FLOAT, right, 100,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // 2) Обмен "двух последних слоёв" -> правому, получаем "две левые halo" от левого
    MPI_Sendrecv(send_right, (int)(2 * plane), MPI_FLOAT, right, 200,
                 recv_left,  (int)(2 * plane), MPI_FLOAT, left,  200,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Основные особенности:

1D декомпозиция по измерению i: Массив разбивается на блоки вдоль оси X (измерение i), каждый процесс получает свой блок.

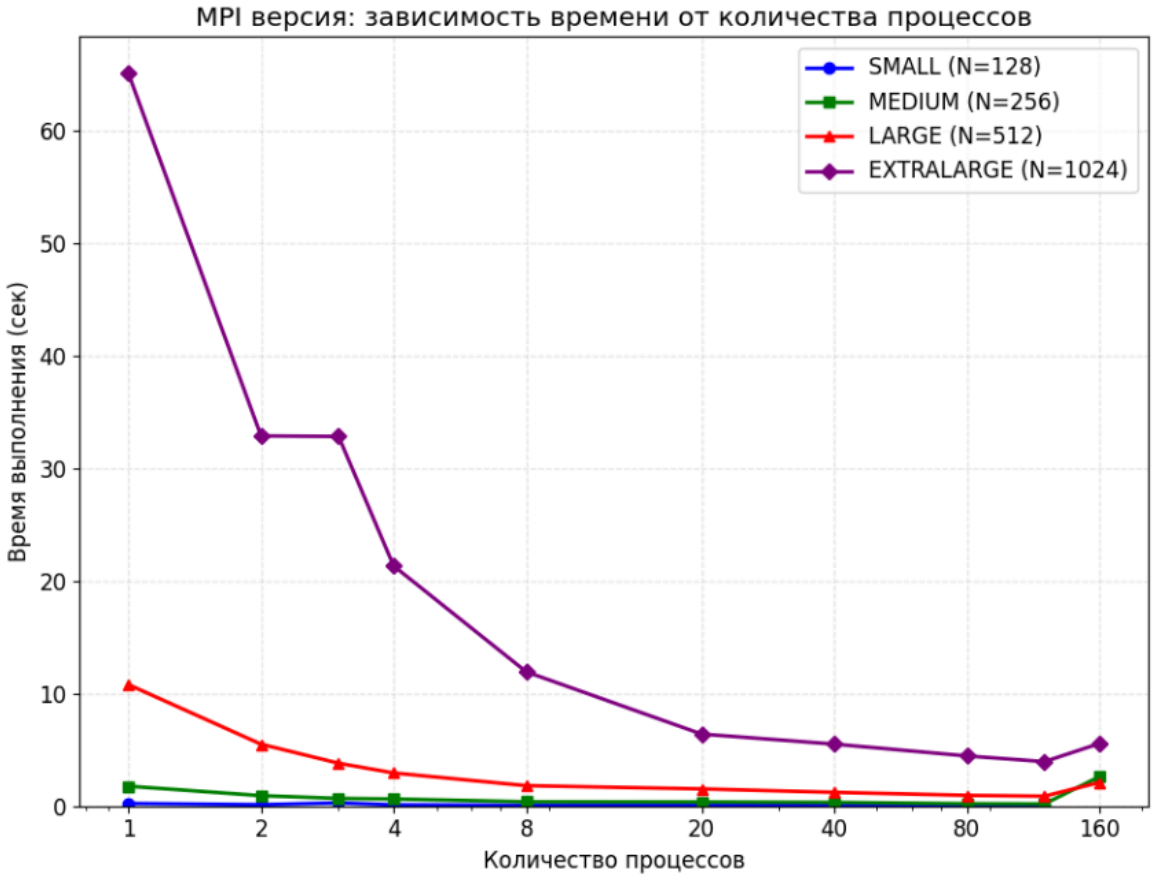
Halo-области толщиной 2: Для корректного вычисления 13-точечного шаблона каждый процесс хранит дополнительные 2 слоя данных с обеих сторон своего блока (halo-области).

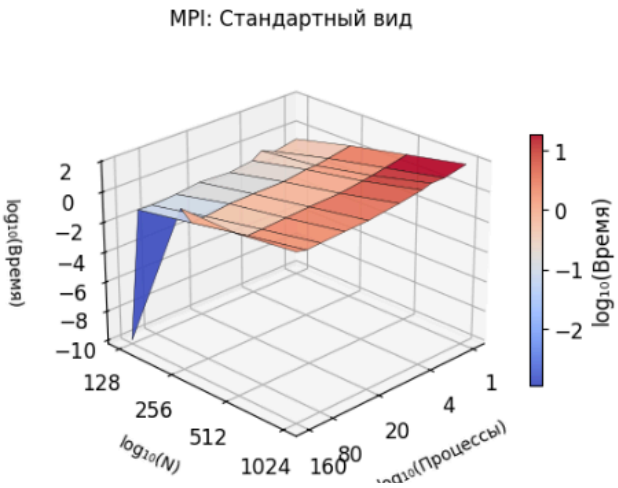
Циклическая передача данных: Используется MPI_Sendrecv для безопасного обмена граничными слоями между соседними процессами.

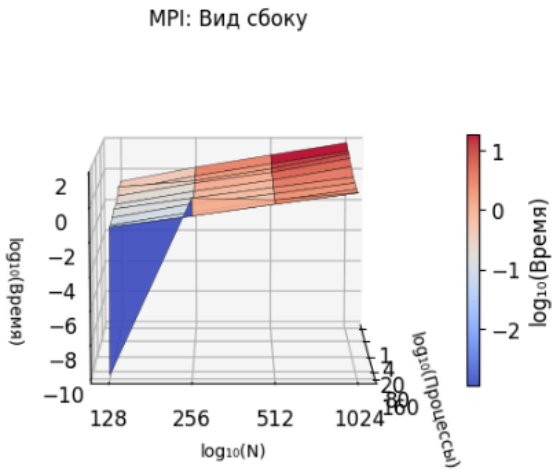
Глобальная редукция: Максимальная ошибка eps вычисляется через MPI_Allreduce для синхронизации всех процессов.

Таблица 5. Время работы (сек.) программы при различных числах нитей и N

Dataset / Threads	SMALL	MEDIUM	LARGE	EXTRALARGE
1	0.214568	1.756682	10.789779	65.146038
2	0.108489	0.91216	5.487352	32.907374
3	0.266857	0.665436	3.795742	32.859516
4	0.094637	0.617340	2.928042	21.343183
8	0.057285	0.367868	1.802257	11.923761
20	0.073939	0.353209	1.513745	6.389919
40	0.045336	0.314297	1.207165	5.496739
80	0.033583	0.193937	0.942258	4.446048
120	0.039181	0.174988	0.865365	3.937961
160	-	2.606553	2.088334	5.549424







Сравнение эффективности параллельных версий
Parallel for показывает стабильно лучшую производительность по следующим причинам:

- Меньшие накладные расходы - директива for оптимальна для регулярных циклов с равномерной нагрузкой
- Эффективное распределение нагрузки - автоматическое распределение итераций между потоками
- Распараллеливание осуществляется по внешнему циклу, что обеспечивает минимальные накладные расходы на управление потоками

Версия на основе механизма задач (omp task) обеспечивает корректное распараллеливание вычислений, но в большинстве случаев уступает реализации omp for

Причины снижения эффективности:

- дополнительные накладные расходы на создание и планирование задач
- более высокая стоимость синхронизации
- относительно мелкая гранулярность задач по сравнению с затратами на их управление

Использование task оправдано при сложной или неравномерной структуре вычислений, однако для данной задачи такой подход оказывается менее эффективным.

MPI-версия позволяет масштабировать вычисления за пределы одного вычислительного узла и корректно распределяет данные между процессами.

Преимущества MPI:

- отсутствие ограничения по объёму памяти одного узла
- возможность использования нескольких вычислительных узлов

Недостатки:

- значительные накладные расходы на обмен граничными слоями
- необходимость глобальных операций синхронизации
- снижение эффективности при слишком большом числе процессов

MPI-версия демонстрирует наибольшую эффективность на больших размерах задачи и при умеренном числе процессов.

OpenMP for является наиболее эффективной параллельной реализацией при использовании общей памяти.

OpenMP task обеспечивает гибкость, но имеет большие накладные расходы и уступает по производительности.

MPI позволяет масштабировать задачу, но эффективность ограничивается затратами на межпроцессные коммуникации.

Эффективность параллельных версий существенно зависит от архитектуры вычислительной системы и характера задачи. Для задач, подобных нашей на одном узле предпочтительно использование OpenMP с директивой for, тогда как MPI становится целесообразным при необходимости распределения данных между несколькими узлами и увеличения доступного объёма памяти.

Исследование масштабируемости

Полученные результаты показывают, что ускорение программы носит **сублинейный характер** и при достижении определённого числа потоков или процессов эффективность начинает снижаться.

Основные причины недостаточной масштабируемости

Обращения к памяти

Рассматриваемая задача относится к классу зависящих от доступа к памяти. Для каждого вычисляемого элемента выполняется сравнительно небольшое число арифметических операций при большом количестве обращений к памяти.

В результате:

- производительность ограничена пропускной способностью памяти
- увеличение числа вычислительных ядер не приводит к пропорциональному росту скорости,

- потоки и процессы начинают конкурировать за доступ к общей памяти

Это является основной причиной ограничения масштабируемости как OpenMP-, так и MPI-реализаций.

Также к недостаточной масштабируемости приводят накладные расходы на синхронизацию

OpenMP

- барьеры синхронизации между итерациями
- ожидание завершения всех задач
- косвенные накладные расходы на управление потоками

При увеличении числа потоков время, затрачиваемое на синхронизацию, становится сопоставимым с временем полезных вычислений.

Коммуникационные издержки в MPI

В MPI-версии при каждом шаге алгоритма выполняются:

- обмен граничными (halo) слоями
- глобальные операции

При увеличении числа процессов:

- возрастает суммарный объём передаваемых данных
- уменьшается объём локальных вычислений
- доля коммуникаций в общем времени выполнения существенно увеличивается

Это приводит к падению эффективности при большом числе процессов.

Вывод

Недостаточная масштабируемость программы при максимальном числе используемых ядер и процессов обусловлена совокупностью факторов: характером задачи, конкуренцией за ресурсы памяти, накладными расходами на синхронизацию и межпроцессные коммуникации. При умеренном числе потоков и процессов достигается наилучшее соотношение между временем вычислений и накладными расходами, тогда как дальнейшее увеличение вычислительных ресурсов не приводит к росту производительности.