Московский государственный университет

Имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Отчет о выполнении задания по параллельному программированию

Вариант 502

Выполнил:

Борисов Т.В. 325 группа

Лектор:

Доцент, к.ф.-м.н. Бахтин Владимир Александрович

Москва, 2025 г.

1. Постановка задачи

В рамках данной лабораторной работы стояла задача исследования производительности алгоритма решения системы линейных алгебраических уравнений методом Гаусса при различных способах распараллеливания и уровнях оптимизации компилятора.

Исходная программа реализует алгоритм метода Гаусса без выбора главного элемента и состоит из двух основных этапов: прямого и обратного хода.

Прямой ход (elimination) — заключается в приведении расширенной матрицы системы к верхней треугольной форме.

Обратная подстановка (reverse substitution) — алгоритм, в котором непосредственно происходит вычисление решения системы.

Основные цели работы:

- Исследовать влияние оптимизаций компилятора на производительность последовательной версии программы;
- Реализовать параллельные версии алгоритма с использованием OpenMP (директивы for и task);
- Реализовать MPI-версию программы;
- Сравнить масштабируемость различных реализаций;
- Сравнить ускорения по сравнению с последовательной версией программы.

```c
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

void prt1a(char *t1, double *v, int n,char *t2);
void wtime(double *t) {
        static int sec = -1;
        struct timeval tv;
        gettimeofday(&tv, (void *)0);
        if (sec < 0) sec = tv.tv_sec;
        *t = (tv.tv_sec - sec) + 1.0e-6*tv.tv_usec;
}

int N;
double *A;
#define A(i,j) A[(i)*(N+1)+(j)]
double *X;

int main(int argc,char **argv) {
        double time0, time1;
        FILE *in;
        int i, j, k;
        in=fopen("data.in","r");
        if(in==NULL) {
                printf("Can not open 'data.in' "); exit(1);
        }
        i=fscanf(in,"%d", &N);
        if(i<1) {
                printf("Wrong 'data.in' (N ...)"); exit(2);
        }
        /* create arrays */
        A=(double *)malloc(N*(N+1)*sizeof(double));
        X=(double *)malloc(N*sizeof(double));
        printf("GAUSS %dx%d\n--------------------------------\n",N,N);
        /* initialize array A*/
        for(i=0; i <= N-1; i++)
                for(j=0; j <= N; j++)
                        if (i==j || j==N)
                                A(i,j) = 1.f;
                        else
                                A(i,j)=0.f;
        wtime(&time0);
        /* elimination */
        for (i=0; i<N-1; i++) {
                for (k=i+1; k <= N-1; k++)
                        for (j=i+1; j <= N; j++)
                                A(k,j) = A(k,j)-A(k,i)*A(i,j)/A(i,i);
        }
        /* reverse substitution */
        X[N-1] = A(N-1,N)/A(N-1,N-1);
        for (j=N-2; j>=0; j--) {
                for (k=0; k <= j; k++)
                        A(k,N) = A(k,N)-A(k,j+1)*X[j+1];
                X[j]=A(j,N)/A(j,j);
        }
        wtime(&time1);
        printf("Time in seconds=%gs\n",time1-time0);
        prt1a("X=(", X,N>9?9:N,"...)\n");
        free(A);
        free(X);
        return 0;
}

void prt1a(char * t1, double *v, int n,char *t2) {
        int j;
        printf("%s",t1);
        for(j=0;j<n;j++)
                printf("%.4g%s",v[j], j%10==9? "\n": ", ");
        printf("%s",t2);
}
```

Исходный код программы файла var502.c

2. Сравнение работы оптимизаторов компилятора

2.1 Способы сравнения

Для исследования влияния оптимизаций компилятора использовались стандартные уровни оптимизации IBM XL C Compiler:

- **O1** — оптимизатор обеспечивает базовые оптимизации;
- **O2** — дает средний уровень оптимизации;
- **O3** — выполняет агрессивные оптимизации, включающие в себя более глубокую перестановку инструкций.

Компиляция последовательной версии программы выполнялась следующим образом:

```
xlc -O1 -qarch=pwr8 -o gauss-seq_O1 gauss-seq.c -lm

xlc -O2 -qarch=pwr8 -o gauss-seq_O2 gauss-seq.c -lm

xlc -O3 -qarch=pwr8 -o gauss-seq_O3 gauss-seq.c -lm
```

Запуски выполнялись для различных значений N с несколькими повторами, после чего усреднялось время выполнения. Количество повторений было выбрано равным 3.

2.2 Результаты

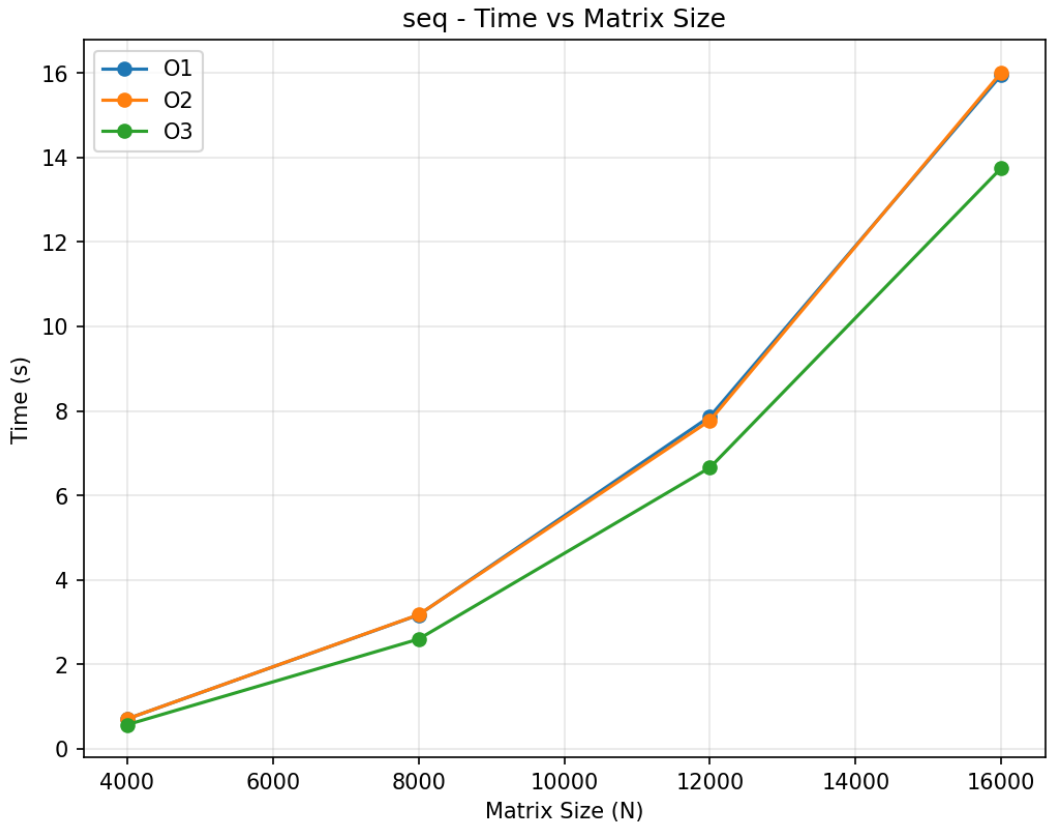

Рис 1. Сравнение работы оптимизаторов (sequential реализация) при различных объемах данных

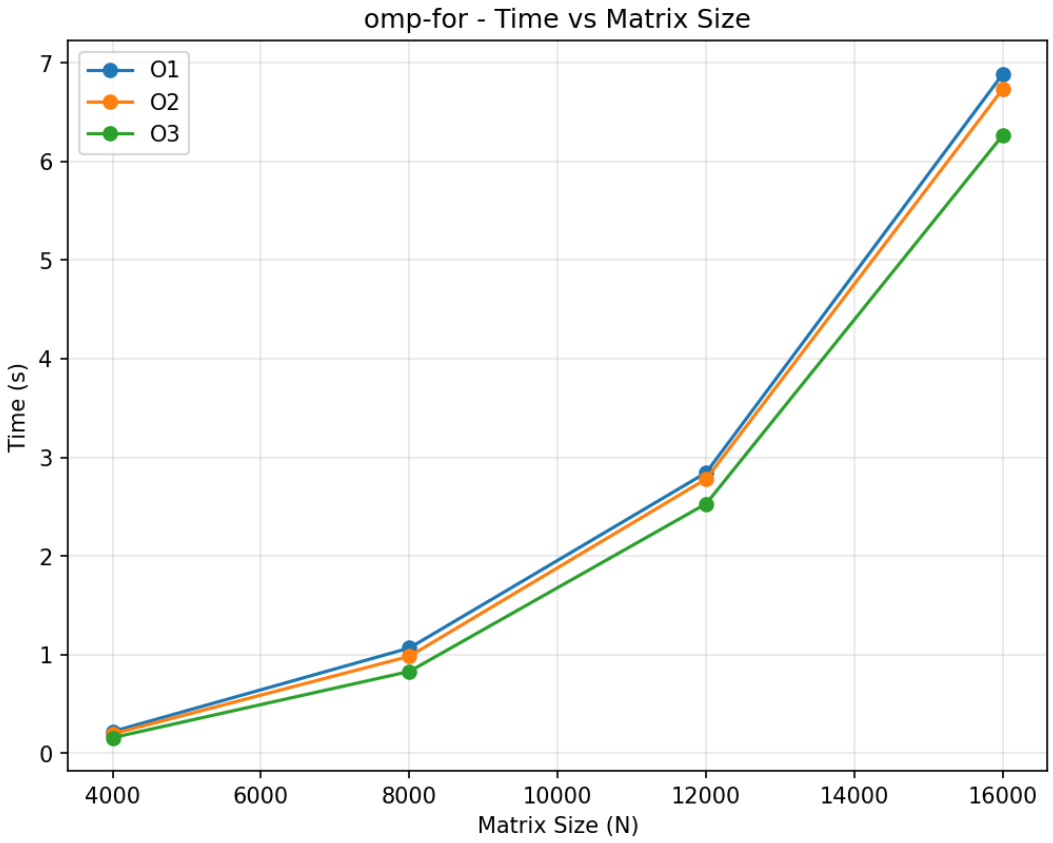

Рис 2. Сравнение работы оптимизаторов (for реализация) при различных объемах данных

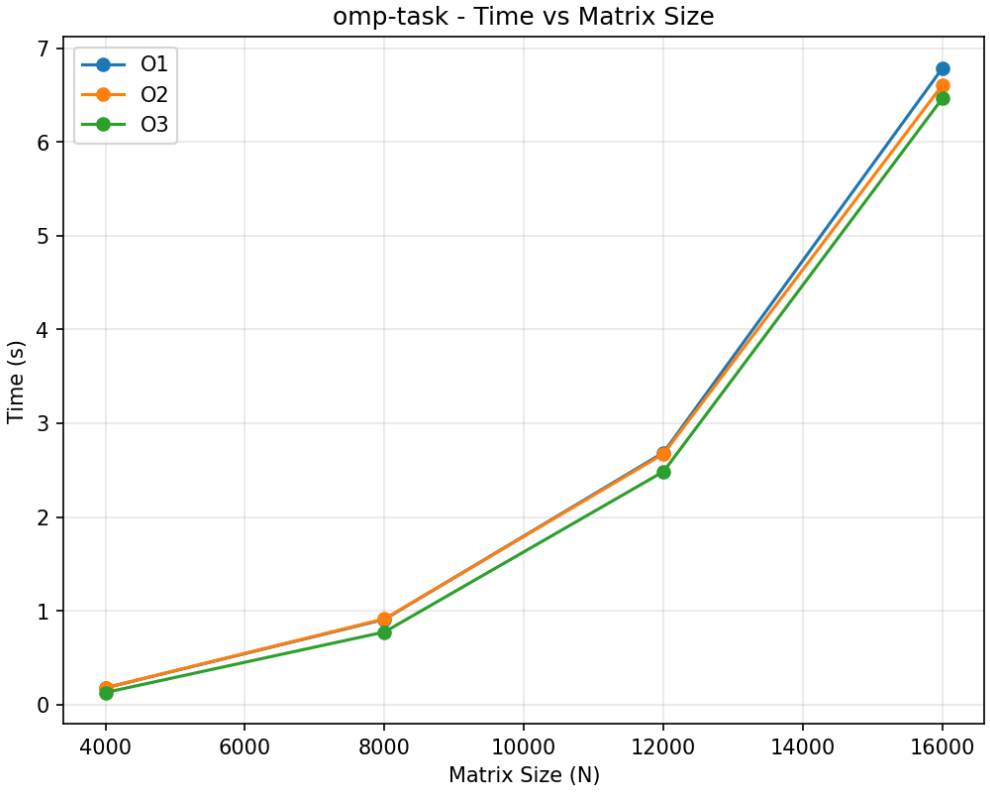

Рис 3. Сравнение работы оптимизаторов (task реализация) при различных объемах данных

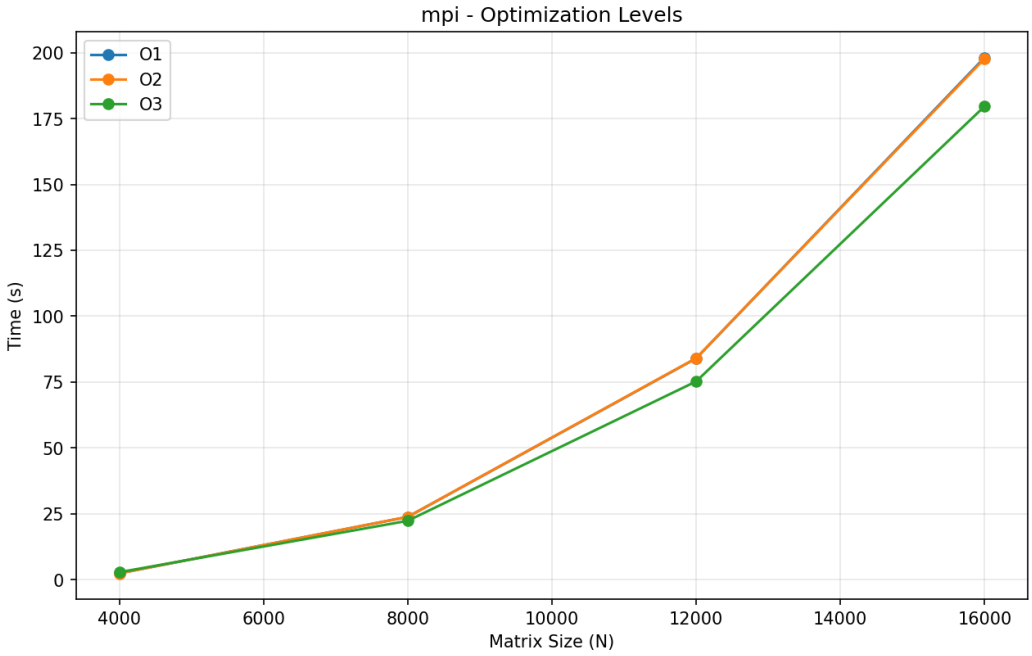

Рис 4. Сравнение работы оптимизаторов (MPI реализация) при различных объемах данных

Размер матрицы (N)	**Seq**	**omp for**	**omp task**	**MPI (8 процессов)**
4000	0.7087	0.1963	0.1791	2.3581
8000	3.1737	0.9827	0.9140	23.8248
12000	7.7722	2.7803	2.6738	84.0583
16000	16.0148	6.7362	6.6093	197.8060

Таблица 1. Сравнение работы оптимизаторов при различных объемах входных данных (N) и различных реализациях параллельной версии программы

Видим, что несущественно оптимизатор O3 дал наилучший результат, что неудивительно, ведь из всех вышеперечисленных он является наиболее «сильным».

Тем не менее, в дальнейших экспериментах все параллельные версии программ компилировались с уровнем оптимизации O2 как обеспечивающим оптимальный баланс между производительностью и стабильностью.

3. Реализация параллельной версии программы с использованием OpenMP (#pragma omp for)

3.1 Идея распараллеливания

Наиболее вычислительно затратной частью алгоритма является прямой ход метода Гаусса. На каждом шаге i вычисления для различных строк k независимы, что позволяет распараллелить внешний цикл. Однако, помимо распараллеливания прямого хода в алгоритме, также директива pragma omp for была применена и для обратного хода.

Директива #pragma omp for, использованная при распараллеливании программы и позволяющая оптимизировать вычисления, позволяет автоматически распределять итерации цикла между доступными потоками. Ниже представил ключевые изменения, которые были внесены в исходный (неоптимизированный) код

3.2 Изменения в коде

Замечание: первый фрагмент – относится к неоптимизированной версии программы, второй – к исправленной и доработанной версии.

Оптимизация прямого хода

```
for (i=0;i<it;i++)
        for (k=i+1;k<N;k++)
            for (j=i+1;j<=N;j++)
                A(k,j) -= A(k,i)*A(i,j)/A(i,i);
```

```
#pragma omp parallel for private(j) schedule(static)
        for(k=i+1;k<N;k++)
            for(j=i+1;j<=N;j++)
                A(k,j)-=A(k,i)*A(i,j)/A(i,i);
```

- Внешний цикл по i оставлен последовательным, так как каждая следующая итерация зависит от результатов предыдущей (обработка очередного ведущего элемента).
- Цикл по k распараллелен с помощью #pragma omp parallel for. Этот цикл перебирает строки, которые нужно преобразовать на текущем шаге элиминации.
- Итерации независимы, так как каждая строка модифицируется на основе значений из одной и той же ведущей строки i, но не зависит от других строк k.
- Переменная j объявлена как private, чтобы каждый поток имел свою локальную копию для внутреннего цикла.
- Использована политика планирования schedule(static) для равномерного распределения итераций между потоками без накладных расходов на динамическое распределение.

Оптимизация обратного хода не выполнялась, поскольку у нас возникают строгие зависимости в данных, их обработка должна происходить последовательно.

Плюсы и минусы использования директивы for

Из плюсов: простота реализации, так как вносятся минимальные изменения в структуру кода; хорошая масштабируемость для больших размеров матрицы; эффективное

использование кэша, так как каждый поток обрабатывает непрерывные блоки строк, что улучшает локальность данных.

Из минусов: внешние циклы остаются последовательными. Также накладные расходы на создание и синхронизацию потоков могут снижать эффективность для малых размеров задачи.

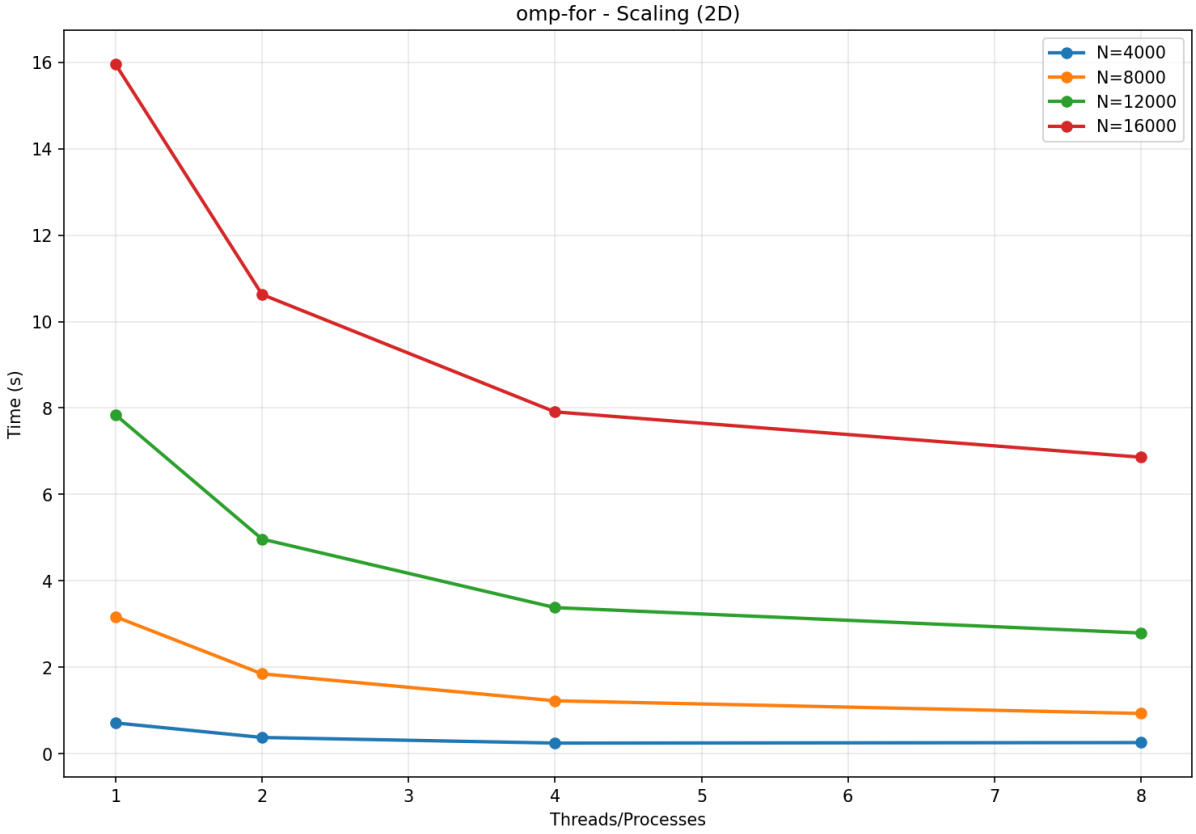

Рис. 5. Сравнение времени работы программы (for) при различных числах нитей.

Исходя из графика, мы можем видеть, что с увеличением количества нитей скорость работы программы лишь становится выше. После количества нитей, равным 4, рост ускорения уже становится не таким существенным. Наиболее заметно ускорение на наибольших объемах данных. При N=16000 оно наиболее существенно с увеличением числа нитей.

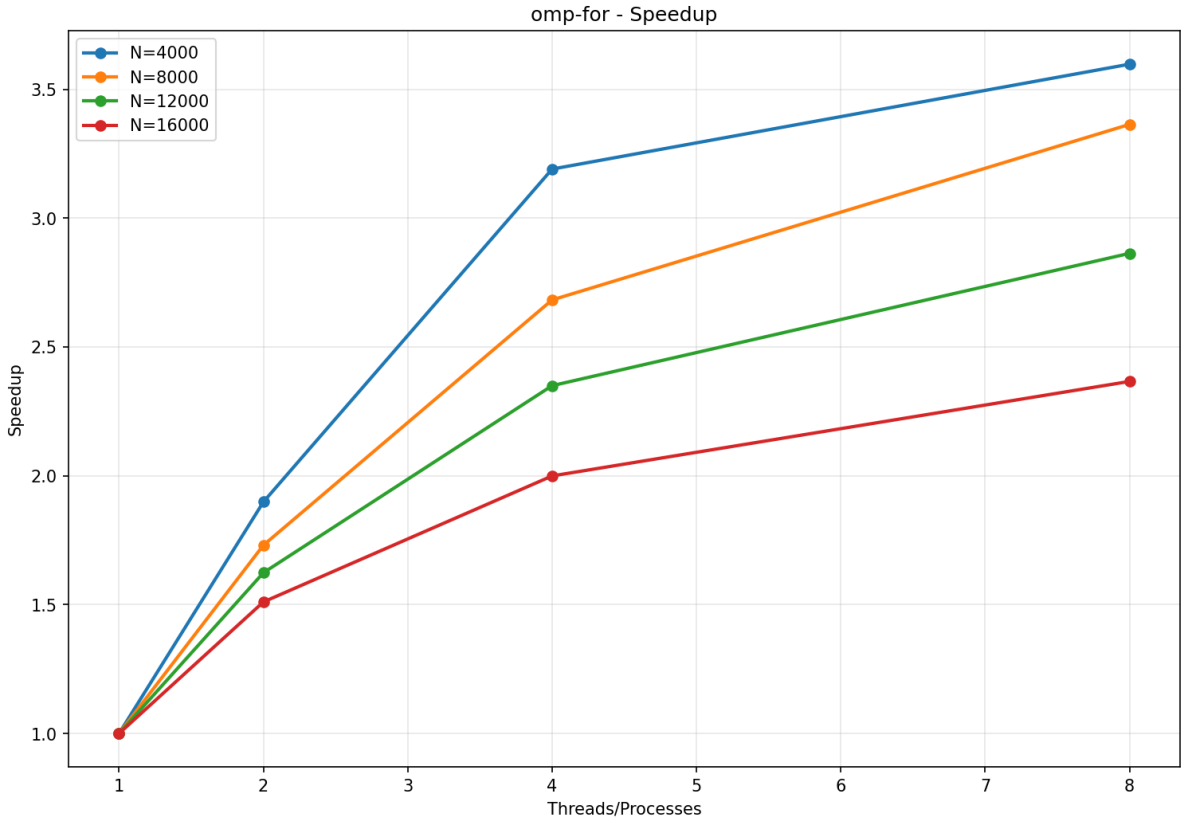

Рис. 6. Сравнение ускорений программы (for) по сравнению с последовательной версией при различных числах нитей.

N / Число потоков	1 поток	2 потока	4 потока	8 потоков
4000	0.7066	0.3721	0.2215	0.1964
8000	3.1555	1.8232	1.1763	0.9381
12000	7.8474	4.8327	3.3395	2.7406
16000	15.9042	10.5297	7.9540	6.7211

Таблица 2. Время работы программы в зависимости от объема данных (N) и числа нитей

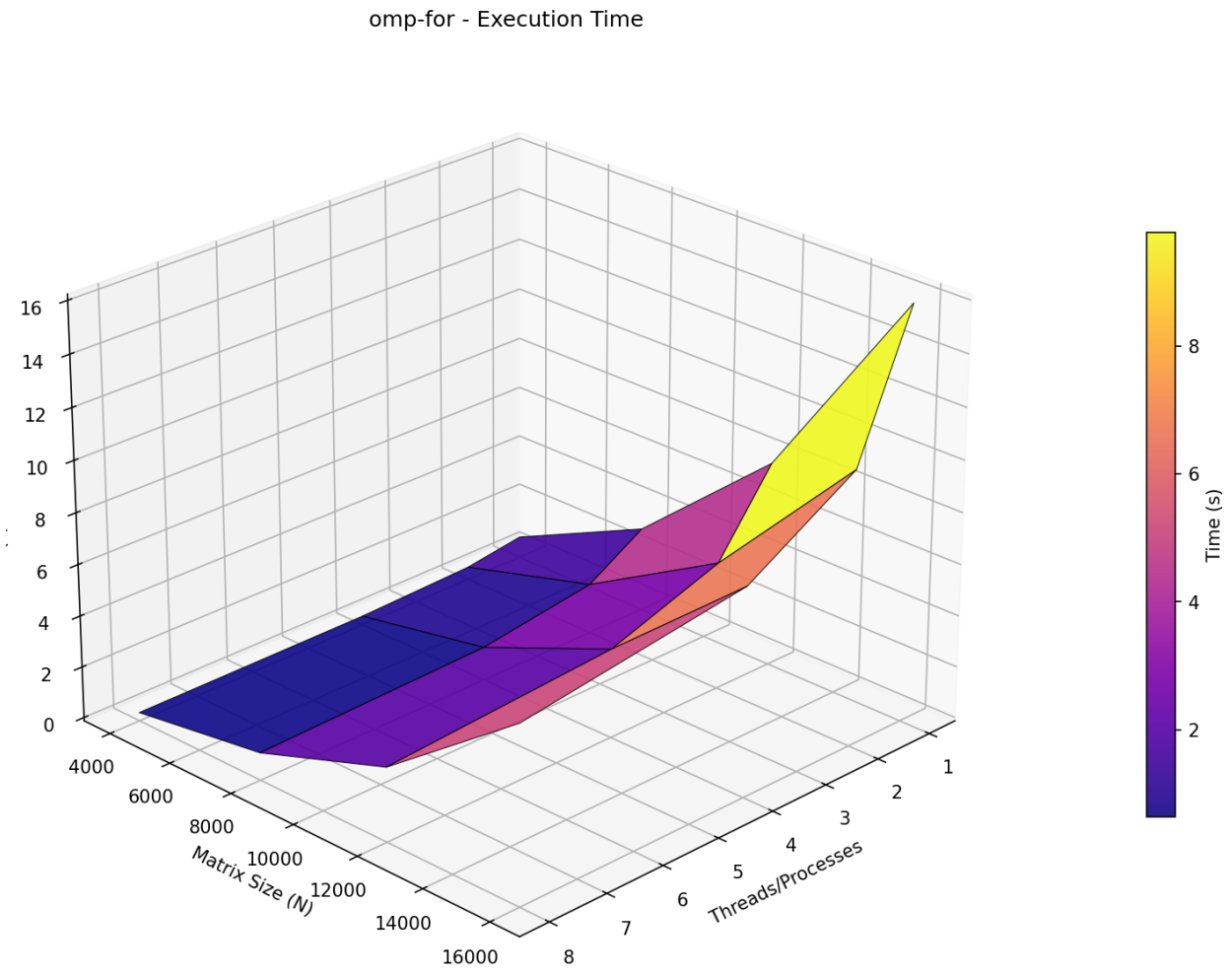

Рис. 7. 3D график, показывающий зависимость времени выполнения программы от числа нитей и объема данных (for)

4. Реализация параллельной версии программы с использованием OpenMP (#pragma omp task)

Идея реализации

В отличие от директивы for, директива task позволяет создавать независимые задачи, которые динамически распределяются между потоками. Каждая задача представляет собой логически завершенную единицу работы, которая может выполняться любым доступным потоком.

Оптимизация прямого хода

```
#pragma omp parallel
    #pragma omp single
    for(i=0;i<it;i++){
        for(k=i+1;k<N;k++)
            #pragma omp task firstprivate(i,k)
            for(j=i+1;j<=N;j++)
                A(k,j)-=A(k,i)*A(i,j)/A(i,i);
        #pragma omp taskwait
    }
```

Пояснения:

- single — гарантирует, что задачи создаёт только один поток. Здесь создаётся параллельная область, в которой одна нить (single) генерирует задачи;
- task — создаёт отдельную задачу для каждой строки k;
- firstprivate — фиксирует значения k и i для каждой задачи;
- taskwait — Обеспечивает завершение всех задач текущего шага i перед переходом к следующему. Необходимо для соблюдения зависимостей по данным

Аналогично предыдущей ситуации, поскольку данные во время обратного хода обрабатываются последовательно, распараллеливание обратного хода не было реализовано.

N / Число потоков	1 поток	2 потока	4 потока	8 потоков
4000	0.7097	0.3845	0.2440	0.1728
8000	3.1744	1.8550	1.1809	0.8744
12000	7.8096	4.8610	3.3866	2.7318
16000	15.9820	10.7433	7.8766	6.5478

Таблица 3. Время работы программы в зависимости от объема данных (N) и числа нитей

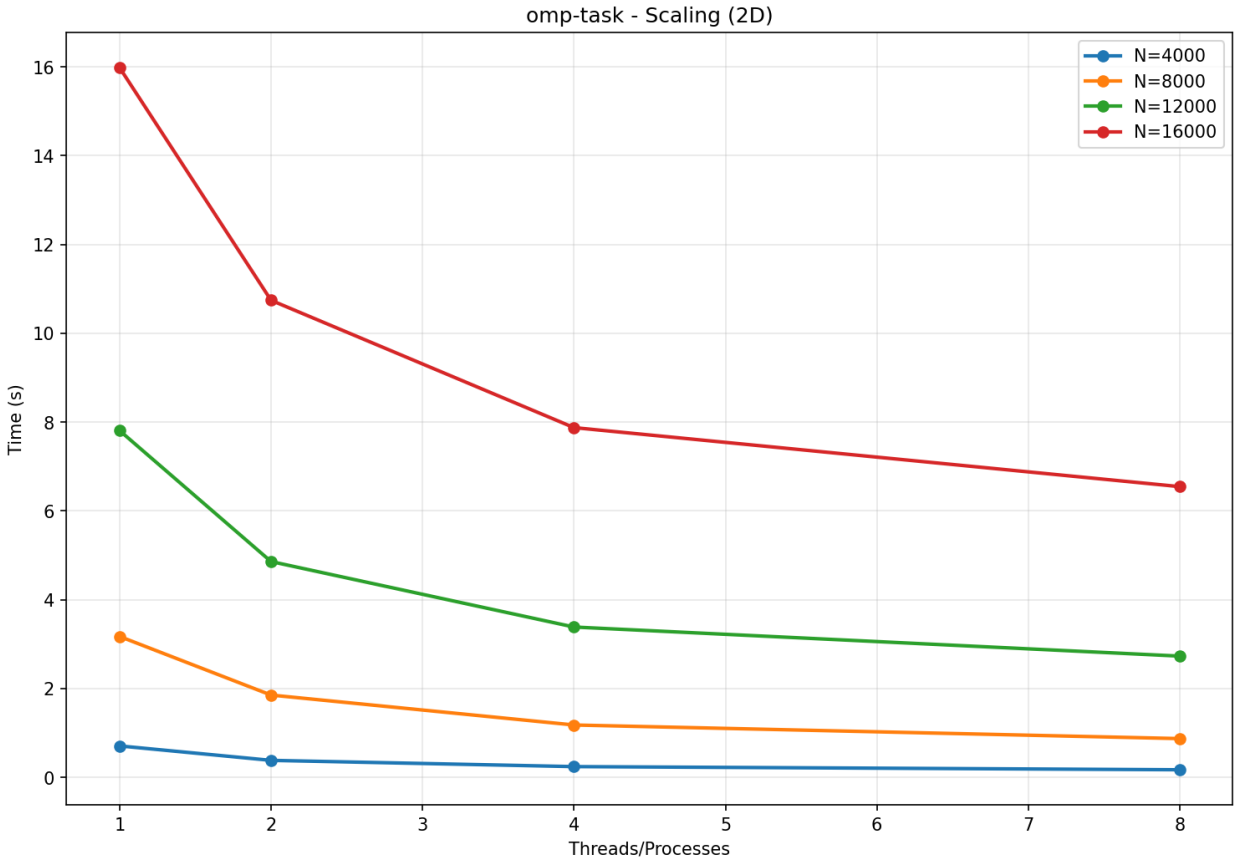

Рис. 8. Сравнение времени работы программы (task) при различных числах нитей.

Здесь наблюдается то, что увеличение количества нитей для вычислений, аналогично for, дает аналогичное ускорение. Заметим, что графики для task получились практически идентичным графикам, построенным для директивы for, изменения минимальны.

Таким образом, подход с использованием директивы task показал аналогичные результаты в ускорении работы програмы.

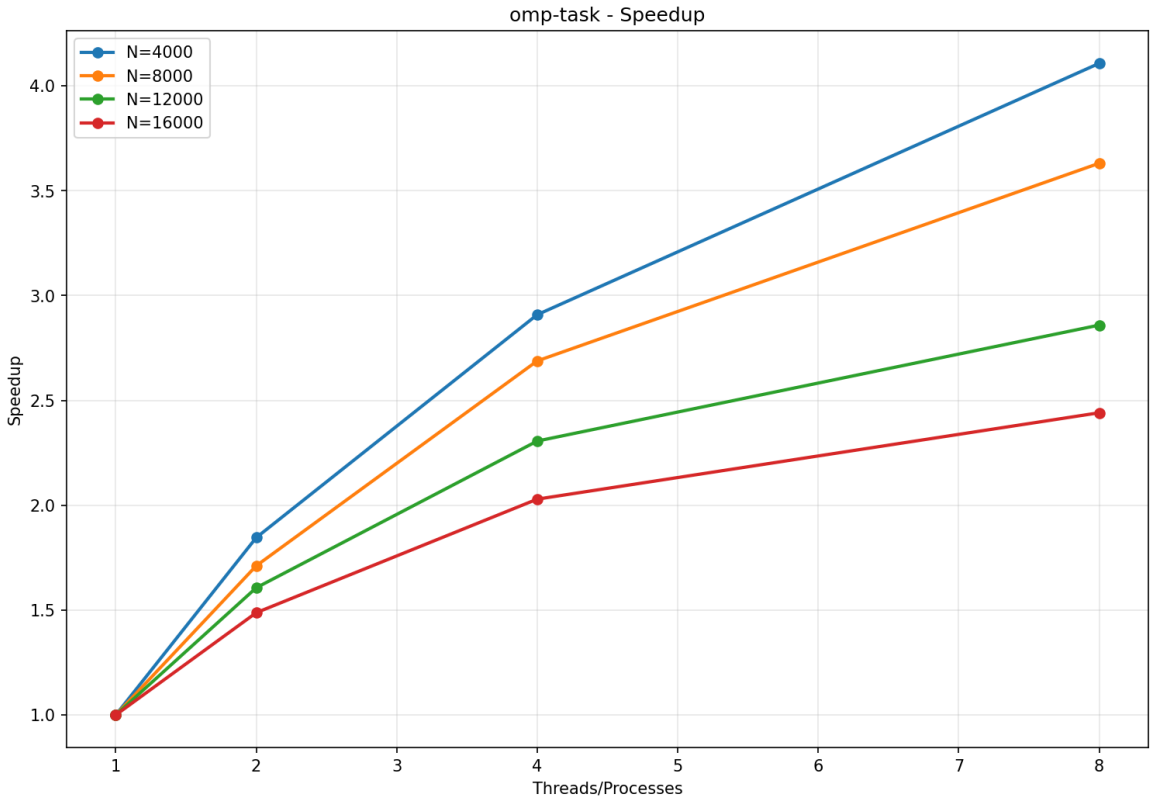

Рис. 9. Сравнение ускорений программы (task) по сравнению с последовательной версией при различных числах нитей.

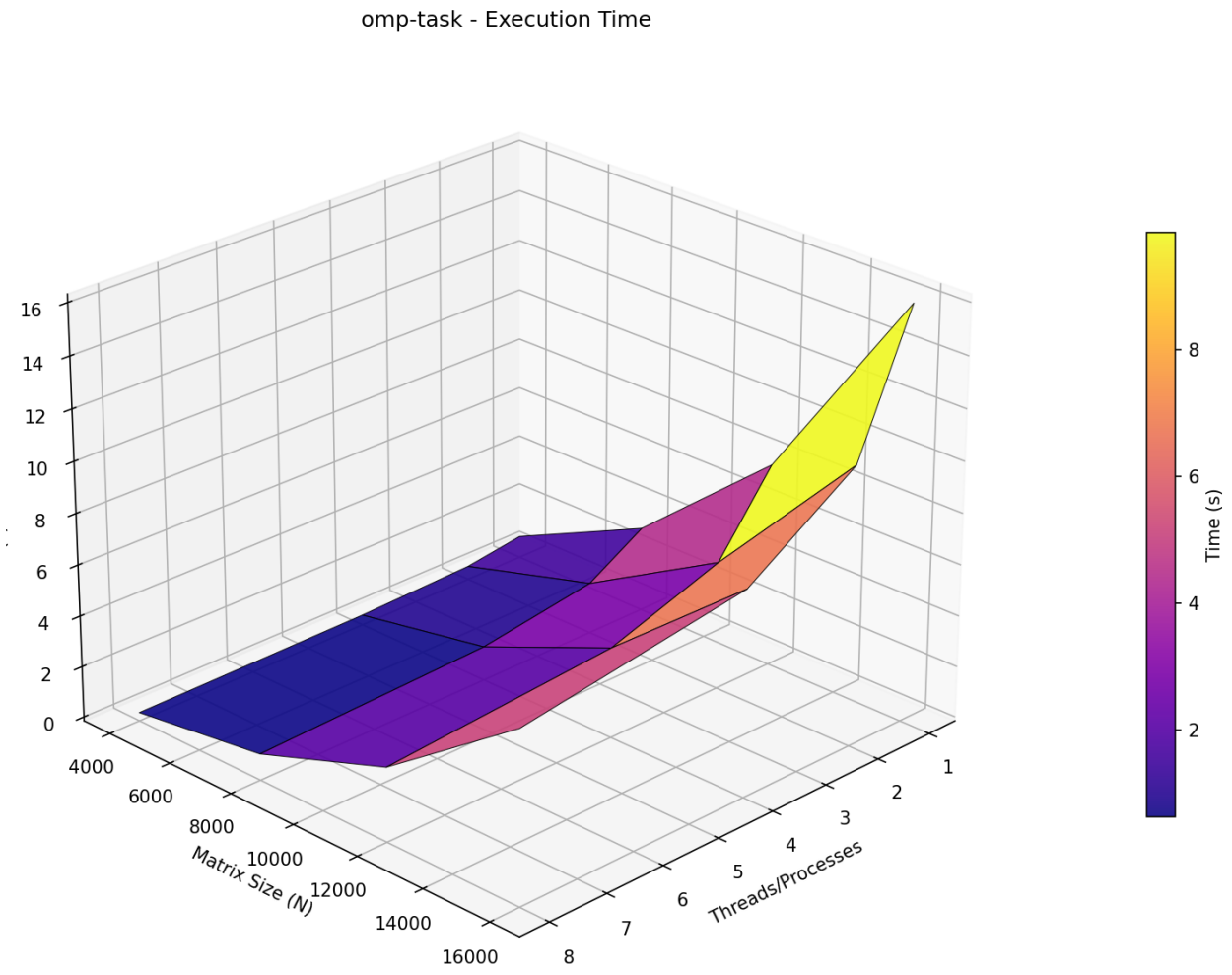

Рис. 10. 3D график, показывающий зависимость времени выполнения программы от числа нитей и объема данных (task)

5. Реализация MPI-версии программы

Идея реализации

В MPI-версии матрица распределяется по строкам между процессами.
Каждый процесс выполняет вычисления для своего блока строк.

Распределение данных

```
int rows_per_proc = N / size;
    int extra_rows = N % size;
    int start_row, local_rows;

    if (rank < extra_rows) {
        start_row = rank * (rows_per_proc + 1);
        local_rows = rows_per_proc + 1;
    } else {
        start_row = rank * rows_per_proc + extra_rows;
        local_rows = rows_per_proc;
    }
```

Матрица размера N×N распределяется по строкам между процессами. Если N не делится нацело на количество процессов, первые extra_rows процессов получают на одну строку больше.

Инициализация локальных данных

```
double *A_local = NULL;
    double *X = NULL;
    double *row_i = NULL;

    if (local_rows > 0) {
        A_local = (double *)malloc(local_rows * (N+1) * sizeof(double));
    }
```

```
for (int i = 0; i < local_rows; i++) {
        int global_i = start_row + i;
        for (int j = 0; j < N; j++) {
            A_local[i*(N+1)+j] = (double)(rand() % 100) / 100.0;
        }
        A_local[i*(N+1)+global_i] = 1.0 + N * 2.0;
        A_local[i*(N+1)+N] = (double)(rand() % 100) / 10.0;
    }
```

Каждый процесс инициализирует только свою часть матрицы. Диагональное преобладание (A[i][i] = 1.0 + N * 2.0) обеспечивает устойчивость метода Гаусса.

Прямой ход метода Гаусса

```
for (int i = 0; i < max_iter; i++) {
        int owner;
        if (i < extra_rows * (rows_per_proc + 1)) {
            owner = i / (rows_per_proc + 1);
        } else {
            owner = extra_rows + (i - extra_rows * (rows_per_proc + 1)) /
rows_per_proc;
        }

        if (rank == owner) {
            int local_i = i - start_row;
            for (int j = 0; j <= N; j++) {
                row_i[j] = A_local[local_i*(N+1)+j];
            }
        }

        MPI_Bcast(row_i, N+1, MPI_DOUBLE, owner, MPI_COMM_WORLD);
```

На каждом шаге определяем «владельца» текущей строки i. Процесс-«владелец» копирует строку в буфер row_i. С помощью MPI_Bcast строка рассылается всем процессам. Затем каждый процесс исключает переменную i из своих строк

Обратный ход

```
for (int j = N-1; j >= 0; j--) {
        int owner;
        if (j < extra_rows * (rows_per_proc + 1)) {
            owner = j / (rows_per_proc + 1);
        } else {
            owner = extra_rows + (j - extra_rows * (rows_per_proc + 1)) /
rows_per_proc;
        }

        if (rank == owner) {
            int local_j = j - start_row;
            double sum = 0.0;
            for (int k = j+1; k < N; k++) {
                sum += A_local[local_j*(N+1)+k] * X[k];
            }
            X[j] = (A_local[local_j*(N+1)+N] - sum) / A_local[local_j*(N+1)+j];
        }

        MPI_Bcast(&X[j], 1, MPI_DOUBLE, owner, MPI_COMM_WORLD);
    }
```

При обратном ходе каждый X[j] вычисляется процессом-владельцем строки j, затем результат рассылается всем процессам для использования в последующих вычислениях.

Временные затраты

```
MPI_Barrier(MPI_COMM_WORLD);
    double t0 = MPI_Wtime();

_____________________________________________________________________________________

double t1 = MPI_Wtime();

    if (rank == 0) {
        printf("Time=%g\n", t1-t0);
```

Время измеряем только на процессе с rank=0.

Для коммуникационных операций использовали MPI_Bcast (метод используется для рассылки строк матрицы и решений).

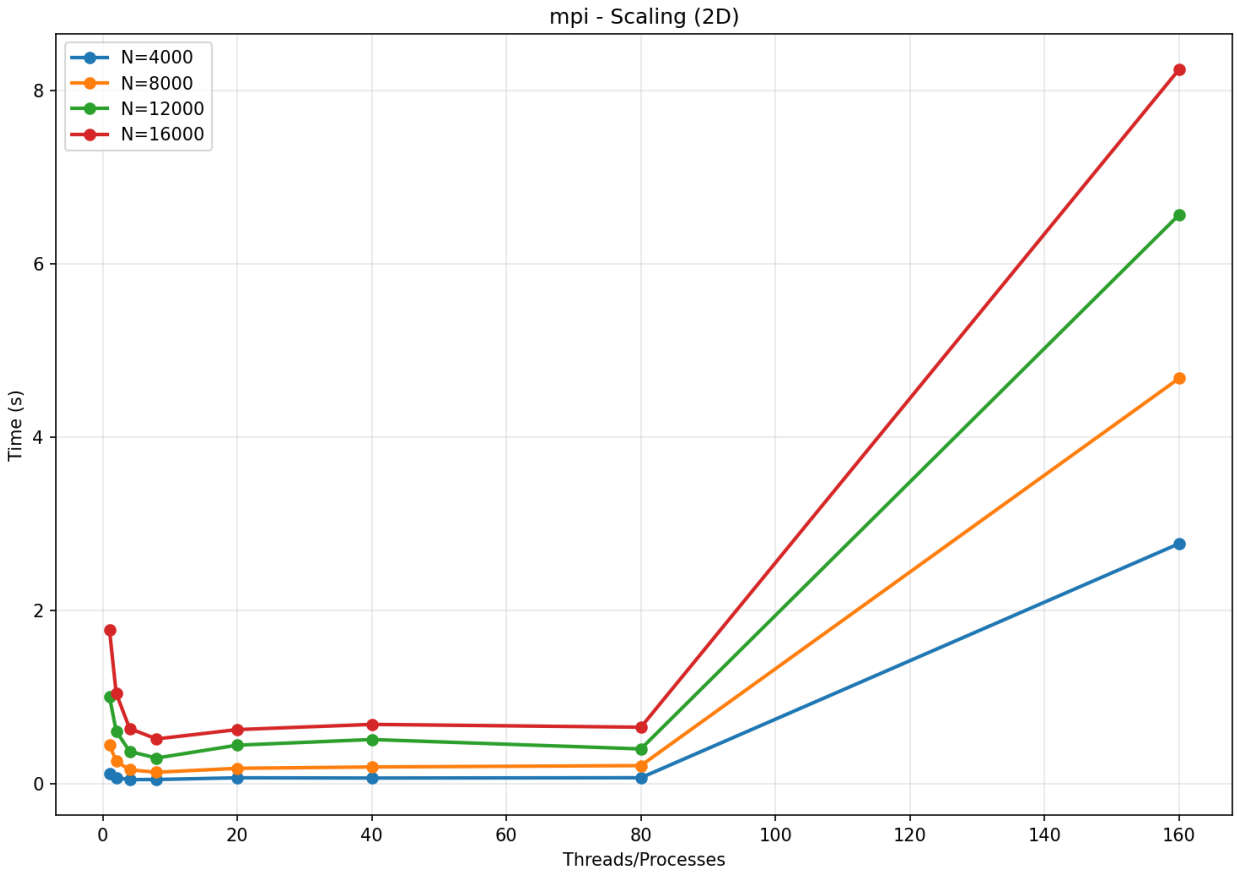

Рис. 11. Сравнение ускорений программы (MPI) при различных числах нитей

Как мы можем видеть, при увеличении количества процессов до 160 программа становится крайне неэффективной. Это связано с накладными расходами на коммуникацию между процессами. MPI показал преимущество лишь для небольшого количества процессов. Далее он неэффективен.

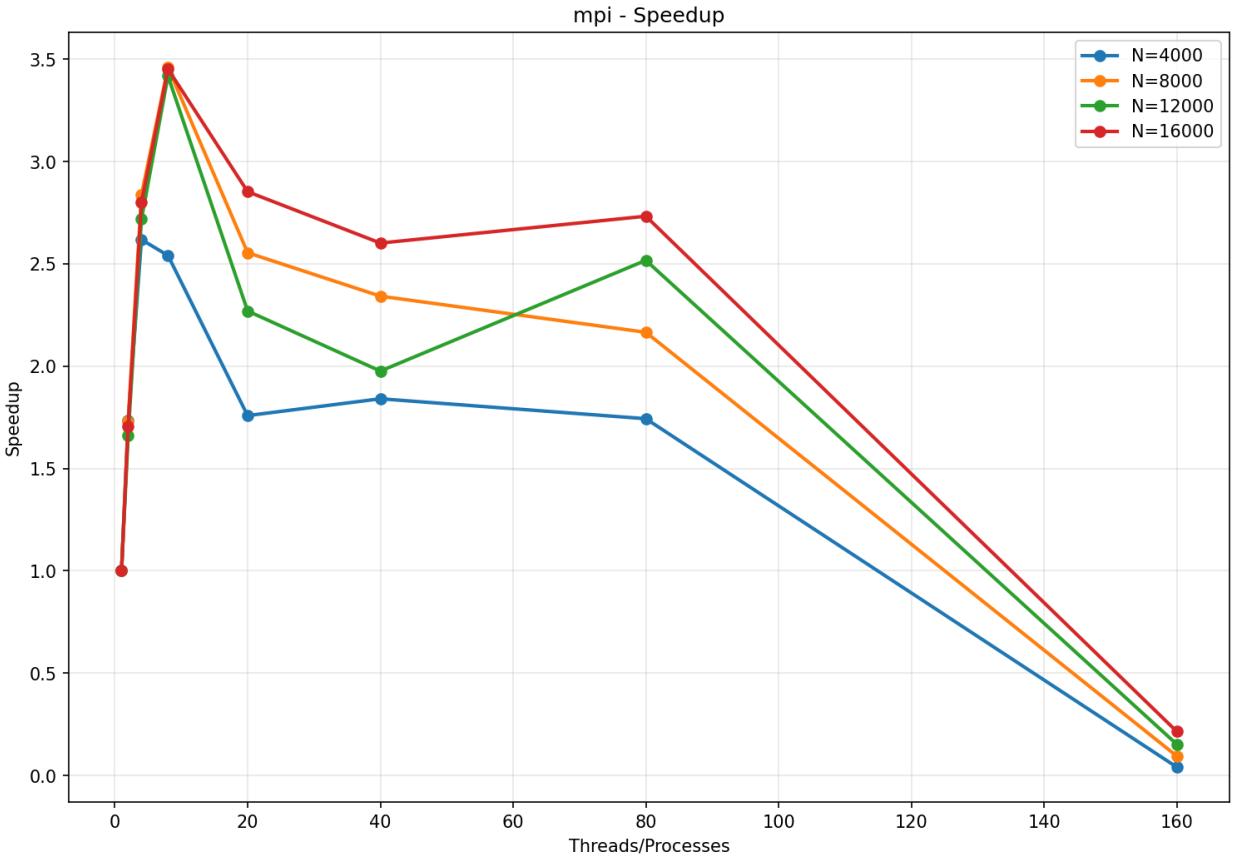

Рис. 12. Сравнение ускорений программы (mpi-версии) по сравнению с последовательной версией при различных числах нитей.

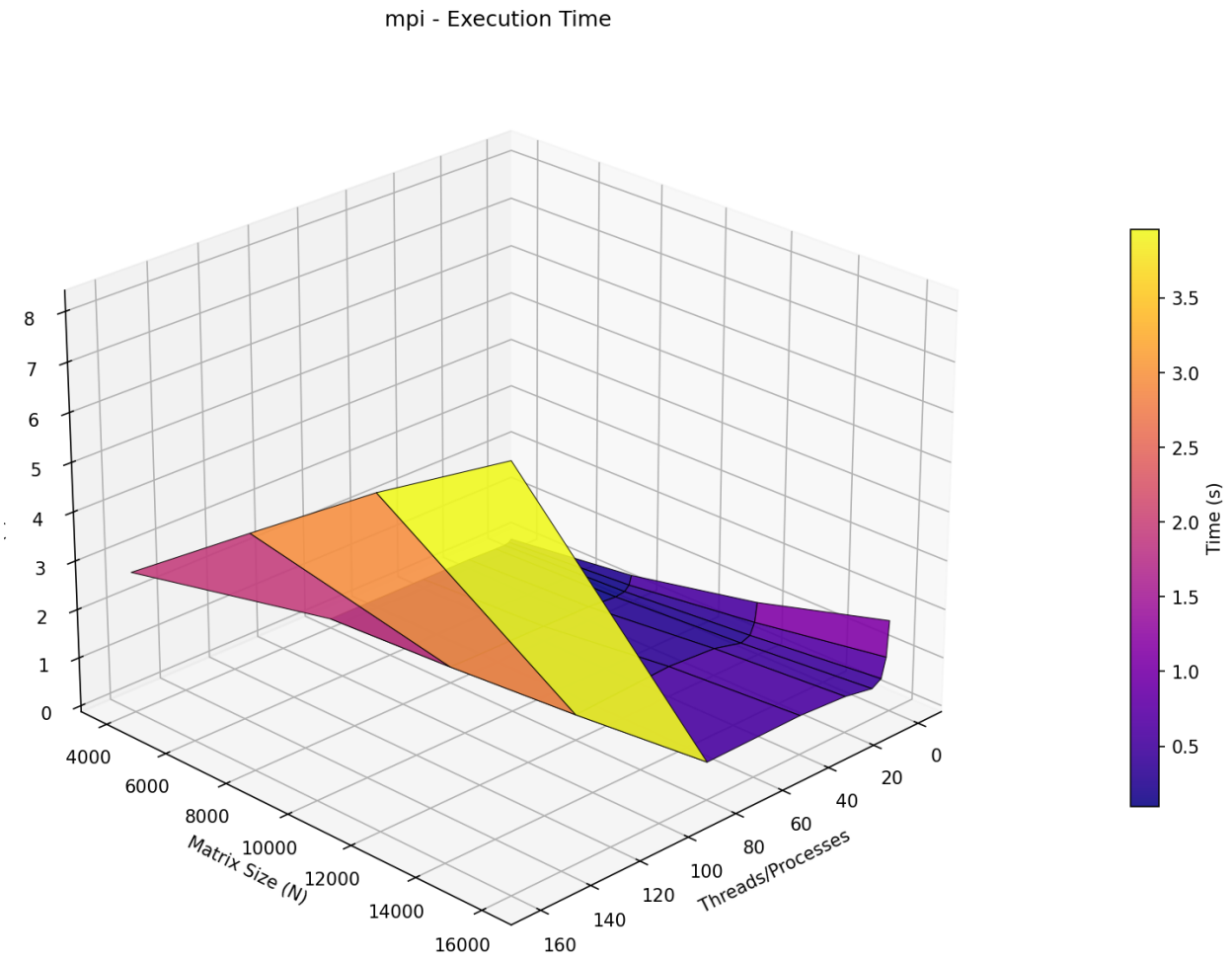

Рис. 13. 3D график, показывающий зависимость времени выполнения программы от числа нитей и объема данных (MPI). Диапазон процессов увеличен

N / Процессы	1	2	4	8	20	40	80	160
4000	0.1133	0.0653	0.0432	0.0446	0.0644	0.0615	0.0650	2.7666
8000	0.4436	0.2562	0.1564	0.1282	0.1736	0.1894	0.2048	4.6739
12000	1.0006	0.6023	0.3681	0.2927	0.4408	0.5063	0.3975	6.5613
16000	1.7709	1.0370	0.6325	0.5129	0.6207	0.6806	0.6479	8.2393

Таблица 4. Время работы программы в зависимости от объема данных (N) и числа нитей

6. Заключение

В ходе работы были реализованы и исследованы последовательная, OpenMP- и MPI-версии алгоритма Гаусса.
Показано, что:

- OpenMP for и task являются наиболее эффективным способом распараллеливания для данной задачи, методы показали схожие результаты;
- MPI-версия обеспечивает лучшую масштабируемость, но ограничивается коммуникационными затратами, которые становятся существенными при большом количестве процессов (например, 160);
- при увеличении числа потоков и процессов эффективность снижается.
- Было получено преимущество оптимизатора O3 по сравнению с O2 и O1, однако благодаря несущественным отличиям, компиляция выполнялась преимущественно с помощью O2.

Полученные результаты соответствуют теоретическим ожиданиям.