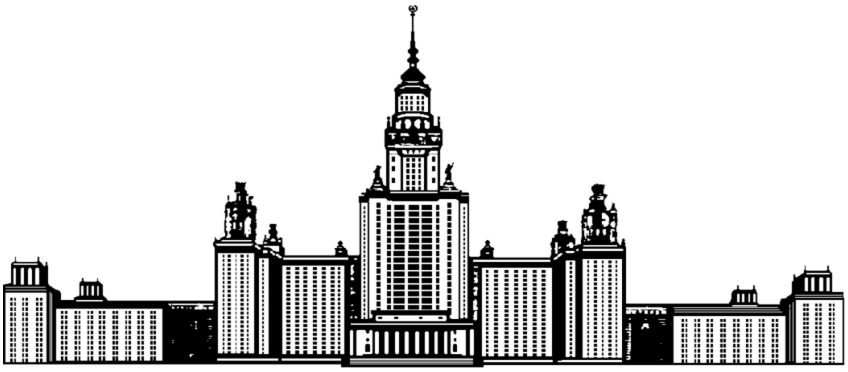
Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики



Практикум по курсу
«Суперкомпьютеры и параллельная обработка данных»
Разработка параллельной программы для задачи
«Метод Гаусса решения системы линейных алгебраических уравнений»

Отчет о выполненном задании

Студента 321 группы

Гудиса Александра Вячеславовича

Лектор: доцент, к.ф.-м.н.

Бахтин Владимир Александрович

Москва, 2024г

Оглавление

1. Постановка и описание задачи

- Для предложенного алгоритма «Метод Гаусса для решения СЛАУ» реализовать несколько версий параллельных программ с использованием технологии OpenMP при помощи директивы for и директивы task, а также, используя технологию MPI
- Протестировать полученные программы на вычислительном комплексе Polus с различными параметрами конфигурации нитей, а также на личном ПК.
- Построить графики зависимости времени выполнения программ от числа нитей для различного объёма входных данных.
- Провести сравнительный анализ эффективности и масштабируемости полученных программ.
- Определить основные причины недостаточной масштабируемости программы при максимальном числе используемых нитей.

2. Описание предложенного алгоритма

Предложенная программа реализует метод Гаусса для решения системы линейных алгебраических уравнений, заданных в виде матрицы $A \times X = B$, где A — квадратная матрица коэффициентов, X — вектор неизвестных, а B — столбец свободных членов. Рассмотрим основные этапы и фрагменты программы:

1. Инициализация данных:
    - Считывается размер матрицы (N) из файла `data.in`. В файле находится одно единственное число – размерность матрицы A
    - фрагмент «*create arrays*». Выделяются динамические массивы A (для хранения расширенной матрицы A|B) и X (вектора решения).
2. Заполнение матрицы A|B (фрагмент «*initialize array A*»):
    - Матрица A заполняется таким образом, что она становится диагональной с единицами на главной диагонали и последним столбцом (правой частью) также заполненным единицами.
3. Метод Гаусса**:**
    - Прямой ход (фрагмент «*eliminatio*n»): Преобразование матрицы A|B к верхнетреугольному виду. Это делается путём исключения переменных снизу вверх.
    - Обратный ход (фрагмент «*reverse substitution*»): Подставляя найденные переменные $X[N-1]$, вычисляются остальные $X[j]$ путём обратной подстановки.
4. Измерение времени:
    - Используется функция `wtime()` для измерения времени выполнения основной «*тяжелой*» части алгоритма.
5. Вывод результатов:
    - Вектор решений X частично выводится (до 9 элементов или меньше, если $N < 9$).
    - Выводится время выполнения алгоритма.

3. Исходный код программы

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

void prt1a(char *t1, float *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;
float *A;
#define A(i,j) A[(i)*(N+1)+(j)]
float *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=(float *)malloc(N*(N+1)*sizeof(float));
	X=(float *)malloc(N*sizeof(float));
	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, float *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);
}
```

4. Модификации исходной программы

1) Операция деления считается одной из самых дорогостоящих арифметических операций. Заметим, что количество операций деления на фазе прямого хода, а именно $A(k,i) \times A(i,j)/A(i,i)$, то есть операция $A(k,j)/A(i,i)$, в самом худшем случае достигает N^3 операций, что на больших входных данных приводит к значительным накладным расходам.
Было принято решение минимизировать количество делений, используя предварительные вычисления, такие как сохранение значения выражения $1.0 \,/\, A(i,i)$ в отдельной переменной типа `float` и использование её для всех вычислений непосредственно в цикле.
Измененную часть программы привожу ниже.

```
/* elimination */
    for (i=0; i<N-1; i++) {
        float pivot = 1.0f / A(i,i);
        for (k=i+1; k <= N-1; k++) {
            float factor = A(k,i) * pivot;
            for (j=i+1; j <= N; j++) {
                A(k,j) -= factor * A(i,j);
            }
        }
    }
```

2) При анализе фрагмента обратного хода метода Гаусса было обнаружено немного излишнее количество операций записи в $A(k,N)$. Была добавлена реализация, при которой $A(k,j)$ сразу изменяется для всех столбцов.
Измененную часть программы привожу ниже.

```
/* 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,j+1) * X[j+1];
        }
        X[j] = A(j,N) / A(j,j);
    }
```

3) Заметим, что матрица A имеет заданные значения в исходной реализации. Сделано это, очевидно, для удобства проверки результатов вычисления в дальнейшем. Для ускорения инициализации матрицы воспользуемся функцией `memset()`, так как это стандартная библиотечная функция, которая, как правило, оптимизирована для конкретной архитектуре процессора и операционной системы. Она часто реализована на низком уровне, с использованием инструкций процессора, которые позволяют гораздо быстрее заполнять память определённым значением, чем это делает обычный цикл.

Измененную часть программы привожу ниже.

```
/* initialize array A */
    memset(A, 0, N*(N+1)*sizeof(float));

    for(i=0; i < N; i++) {
        A(i,i) = 1.0f;
        A(i,N) = 1.0f;
    }
```

5. Тестирование исходной и модифицированной программ

Ввиду достаточно высокой вычислительной сложности задачи (O(N^3)) для локального тестирования было принято решения использовать небольшие значения N, дабы иметь возможность дождаться времени выполнения программы. Замеры времени осуществлялись с помощью уже реализованной в исходной программе функции `wtime()`.
Тестирование проводилось на ноутбуке со следующей конфигурацией:
11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz – 4 ядра, 8 потоков.

Версия программы	N = 100	N = 1000	N = 1500	N = 2000	N = 3000	N =5000
Исходная	0. 005222	1.26337	4.10758	9.76674	33.1832	153.043
Оптимизированная	0.003964	0.890763	3.05513	7.13575	24.0896	114.577

Таблица 1. Результаты локального тестирования исходной программы

Визуализируем полученные значения на графике.

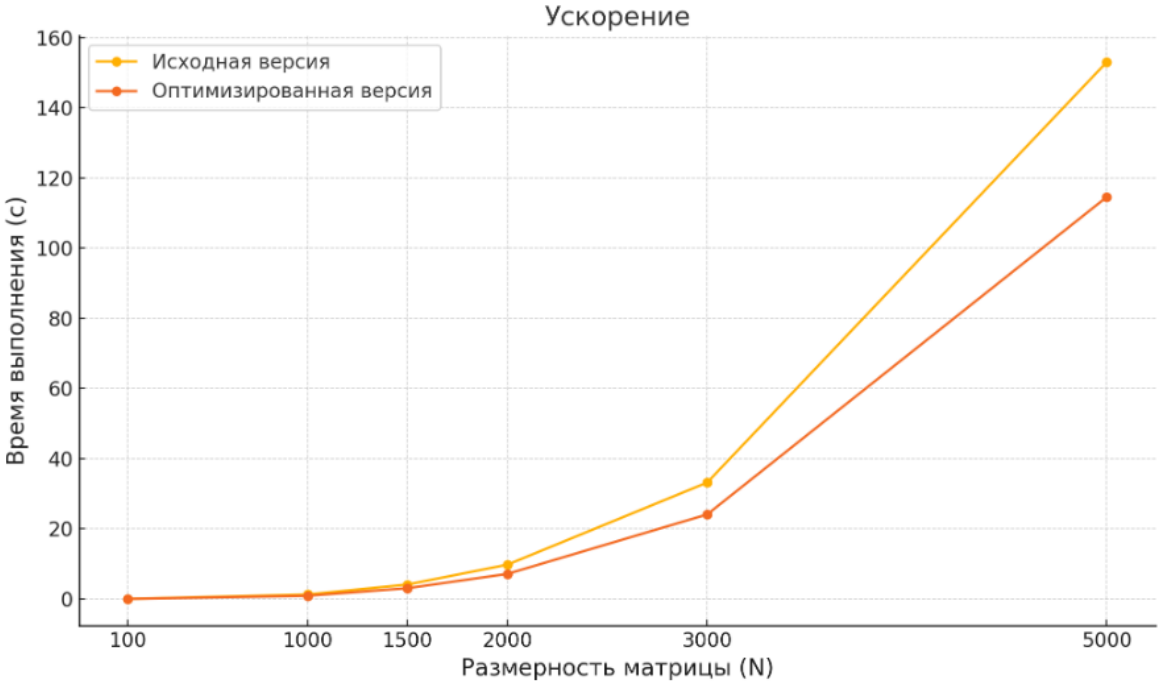


Диаграмма 1. Визуализация зависимости времени выполнения от размерности матрицы для исходной и модифицированной версий программ

Из полученных результатов можем сделать вывод, что оптимизированная версия программы работает незначительно быстрее на небольших размерностях, при этом с увеличением размерности матрицы возрастает разница во времени выполнения с исходной версией.

6. Применение технологии OpenMP. Директива for

6.1. Описание внесенных изменений

Исходный код программы подразумевает 3 «*проблемные*» зоны: инициализация матрицы, прямой ход метода Гаусса, обратный ход метода Гаусса. Конечно, наибольшей, то есть кубической, сложностью обладает второй из этих этапов, но для получения максимальной производительности при распараллеливании программы предлагаю применить средства openMP к каждому и них.

Распараллеливание инициализации матриц.
На этом этапе каждую итерацию цикла `for (i = 0; i <= N - 1; i++)` можно выполнять независимо, так как каждая строка `A(i, j)` задаётся независимо от других.
Для реализации этой возможности добавим в код команду:
`#pragma omp parallel for private(i, j) shared(A, N)`

Обоснование выбранных параметров:
`private(i, j)`: Переменные `i` и `j` объявим как `private`, так как каждый поток должен иметь свою копию этих индексов (они зависят от текущей итерации).
`shared(A, N)`: Переменные `A` и `N` объявим как `shared`, так как все потоки совместно используют матрицу `A` размерности `N`.
В результате модифицированный фрагмент выглядит следующим образом:

```
/* Initialize array A */
    #pragma omp parallel for private(i, j) shared(A, N)
    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;
        }
    }
```

Распараллеливание прямого хода Гаусса.
Прежде всего заметим, что каждую строку k (внутренний цикл `for (k = i + 1; k < N; k++)`) можно обрабатывать независимо, так как вычисления в строках матрицы A независимы друг от друга. Соответственно добавим команду:
`#pragma omp parallel for private(k, j) shared(A, i, N)`

Обоснование выбранных параметров:

`private(k, j)`: Переменные `k` и `j` объявлены как `private`, так как они используются только в текущей итерации внутреннего цикла.

`shared(A, i, N)`: Переменные `A`, `i` и `N` объявлены как `shared`, так как все потоки совместно используют матрицу A, текущий ведущий элемент `i` и размерность `N`.

Хочу отметить, что для корректности необходимо, чтобы внешние циклы по `i` выполнялись последовательно (поэтому нет директив OpenMP), так как вычисления в строках `A(k, j)` зависят от предыдущих строк.

В результате получаем следующий модифицированный фрагмент:

```
/* Elimination */
for (i = 0; i < N - 1; i++) {
    #pragma omp parallel for private(k, j) shared(A, i, N)
    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);
        }
    }
}
```

Распараллеливание обратного хода Гаусса.

На этом этапе каждую строку `k` можно обрабатывать независимо, так как вычисления в строках матрицы A не зависят друг от друга.
Для реализации этого добавим команду:
`#pragma omp parallel for private(k) shared(A, X, j, N)`

Обоснование выбранных параметров:
`private(k)`: Переменная `k` объявлена как `private`, так как она используется только в текущей итерации внутреннего цикла.
`shared(A, X, j, N)`: Переменные `A, X, j` и `N` объявлены как `shared`, так как все потоки совместно используют массивы `A, X`, текущий ведущий элемент `j` и размерность `N`.
Также отмечу, что внешний цикл по `j` остаётся последовательным, так как `X[j]` вычисляется последовательно и используется в следующих итерациях. Таким образом, обратный ход Гаусса приобретает вид:

```
/* Reverse substitution */
X[N - 1] = A(N - 1, N) / A(N - 1, N - 1);

for (j = N - 2; j >= 0; j--) {
    #pragma omp parallel for private(k) shared(A, X, j, N)
    for (k = 0; k <= j; k++) {
        A(k, N) -= A(k, j + 1) * X[j + 1];
    }
    X[j] = A(j, N) / A(j, j);
}
```

6.2. Тестирование полученной программы

Теперь сделаем замеры времени работы полученной программы на разном количестве нитей и различных размерностях исходной матрицы на суперкомпьютере Polus. Время работы программы будем, согласно заданию, вычислять с помощью функции `omp_get_wtime()`. Полученные результаты после запусков усредним и представим в виде таблицы. Строка – размерность матрицы A, столбец – количество потоков, на пересечении усредненное время работы программы.

	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
1000	1,0953	0,5532	0,3756	0,2746	0,2213	0,1848	0,1641	0,1495	0,1351	0,2041	0,1041	0,1148	0,1158	0,1295	0,2195	0,1851	0,1857	0,1947
1500	8,1213	6,4611	5,1631	4,1711	3,4711	3,581	3,3548	2,1441	1,2641	0,6978	0,3847	0,3978	0,4174	0,4595	0,5356	0,4944	0,5011	0,4701
2000	8,6131	6,1653	4,1471	2,6947	2,2318	2,4611	2,2541	2,0371	1,7174	1,5847	1,5938	1,2349	1,1031	0,9371	0,9478	0,8646	0,8481	1,3868
2500	16,471	12,491	5,9611	5,5918	3,7471	3,7647	3,2848	3,5837	3,3418	2,9418	2,4711	2,2791	2,1491	1,8047	1,6547	1,4014	1,5781	2,1944

Таблица 2. Данные для директивы for.

По таблице построим для наглядности два 3d графика, на которых покажем полученную закономерность.

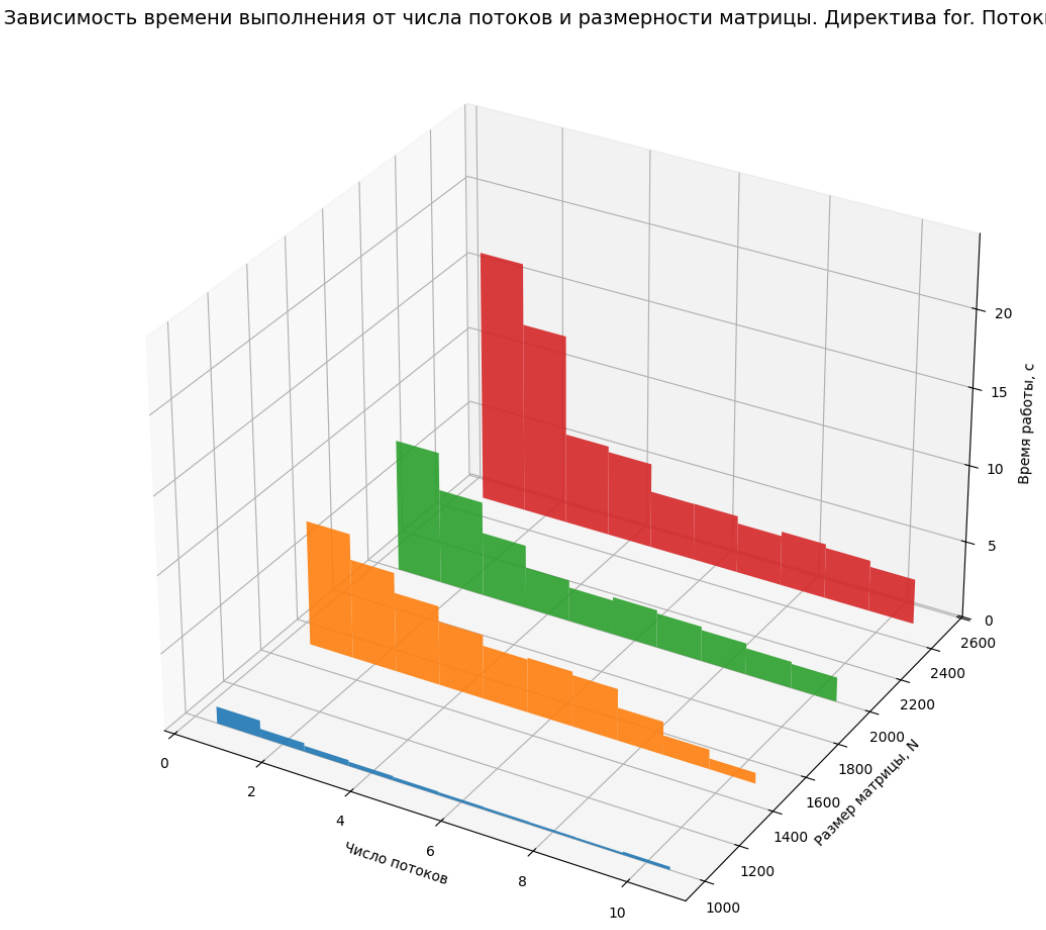


Диаграмма 2. Директива for. Потоки 1-10.

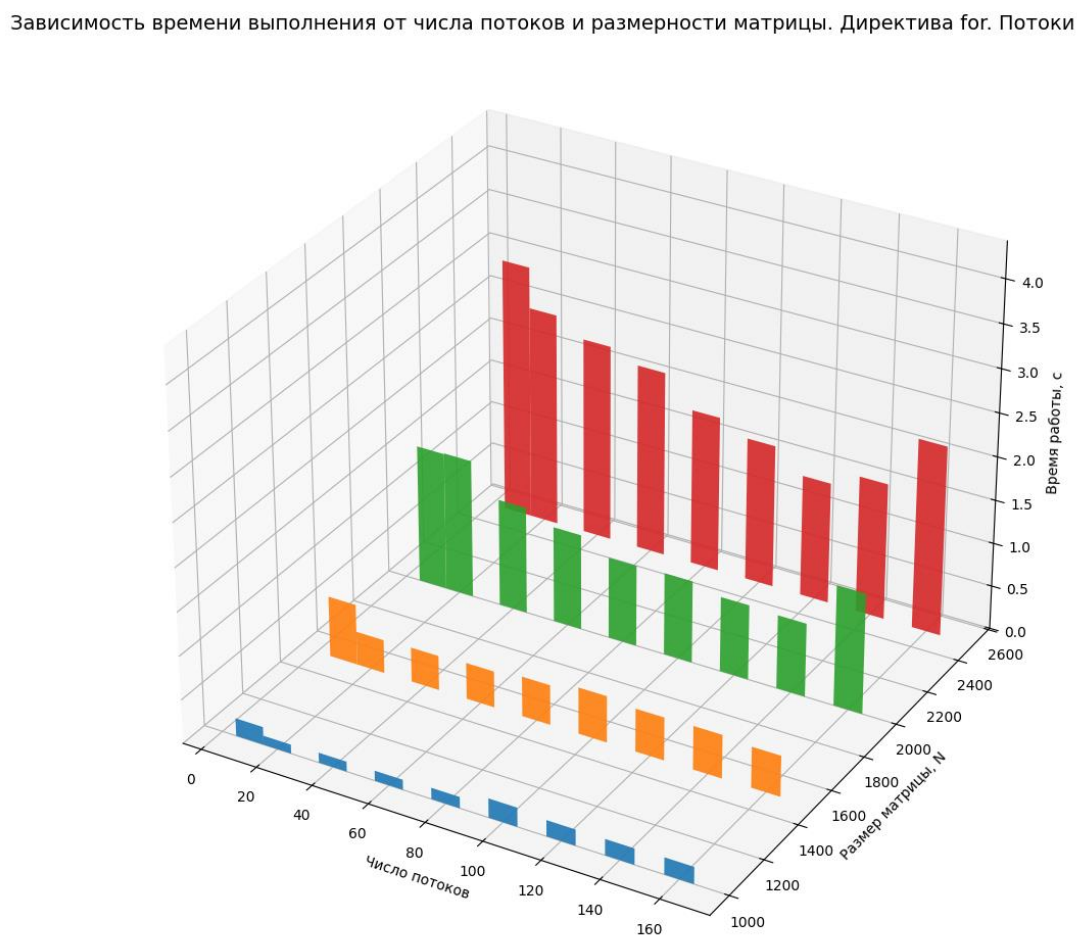


Диаграмма 3. Директива for. Потоки 10-160

Из полученных результатов можно сделать вывод, что использование директивы `# pragma for` существенно ускоряет время работы программы. Заметим, что максимального значения ускорения получается добиться при использовании примерно 20 потоков. В сравнении с исходной версией программы полученная распараллеленная работает до 10 раз быстрее.

7. Применение технологии OpenMP. Директива task.

7.1. Описание внесенных изменений

В данном случае снова рассмотрим основные этапы программы и попробуем их модифицировать с помощью предложенной команды.

Распараллеливание прямого хода Гаусса.

`#pragma omp parallel shared(i) private(j, k)`

Создаём область параллельных вычислений. В этой области создаётся пул потоков, которые будут использоваться для выполнения задач. Данная команда позволяет нам эффективно распределять задачи между потоками, чтобы несколько потоков могли обрабатывать разные участки данных одновременно. По сути, эта директива обеспечивает своего рода подготовку программной среды для выполнения следующих директив вида `#pragma omp single` и `#pragma omp task`.

Далее воспользуемся командой:
`#pragma omp single`

Тем самым мы указываем, что следующий блок кода должен быть выполнен только одним потоком. Нужно это для того, чтобы избежать множественного создания задач разными потоками. Все задачи (то есть созданные через `#pragma omp task`) создаются только одним потоком.
Здесь же воспользуемся `single`, чтобы один поток создавал задачи для обработки блоков строк в функции `elimination_phase()`

Наконец используем команду
`#pragma omp task`

С её помощью мы определяем задачу, которая может быть выполнена любым потоком из пула. Благодаря этому мы имеем возможность эффективно распараллелить обработку строк матрицы.

После наша задача вызывает функцию `elimination_phase`, которая обрабатывает часть строк матрицы, а именно блоки строк размером `tasksize`.

В результате получаем следующий модифицированный код этого этапа:

```
/* elimination */
for (i=0; i<N-1; i++) {
      #pragma omp parallel shared(i) private(j, k)
      #pragma omp single
```

```
            for (k = i + 1; k <= N - 1; k += tasksize)
                  #pragma omp task shared(A) shared(i, k)
            {
            elimination_phase(A, i, k);
                  }
      }
```

Распараллеливание обратного хода Гаусса.

Сперва, аналогичным образом, как и в прямом ходе, используем команду:

`#pragma omp parallel shared(A,j) private(k)`
С той же самой целью.

Затем воспользуемся командой:

`#pragma omp task firstprivate(k)`

С её помощью мы передаём копию переменной k в конкретную задачу. Это важно для того, чтобы каждая задача имела свою локальную копию переменной, ведь переменные в задачах должны быть независимыми, чтобы избежать ошибок, вызванных одновременным доступом потоков.

После чего идет вызов функции `reverse_phase`, смысл которой рассмотрим далее.

Таким образом, получаем следующую модификацию фрагмента кода:

```
/* reverse substitution */
X[N - 1] = A(N - 1, N) / A(N - 1, N - 1);
for (j = N - 2; j >= 0; j--)
{
    #pragma omp parallel shared(A,j) private(k)
    for (k = 0; k <= j; k += tasksize)
    {
        #pragma omp task firstprivate(k)
        {
            reverse_phase(A,X,j,k);
        }
    }
}
for (j = N - 2; j >= 0; j--)
{
    X[j] = A(j, N) / A(j, j);
}
```

Теперь отдельно рассмотрим реализованные функции.

Функция `elimination_phase`**.** Выполняет часть работы по исключению переменной из системы уравнений. Она вызывается в задаче, созданной директивой `#pragma omp task`. Потоки выполняют эту функцию параллельно, каждый обрабатывая свой блок строк.

Функция `reverse_phase`**.** Выполняет часть работы по обратной подстановке. Она обновляет элементы правой части уравнения с учётом уже найденных значений. Аналогично `elimination_phase`, вызывается в задачах. Каждый поток работает с независимыми блоками данных.

Приведем реализации описанных функций:

```
#define tasksize 32
#define min(a, b) (a) < (b) ? (a) : (b)


void elimination_phase(float *A, int i, int k_start)
{
    int k_end = min(k_start + tasksize, N);
    for (int k = k_start; k < k_end; k++)
    {
        for (int j = i + 1; j <= N; j++)
            A(k, j) = A(k, j) - A(k, i) * A(i, j) / A(i, i);
    }
}

void reverse_phase(float *A, float *X, int j, int k_start) {
    int k_end = min(k_start + tasksize, j+1);
    for (int k = k_start; k < k_end; k++)
    {
        A(k, N) = A(k, N) - A(k, j + 1) * X[j + 1];
    }
}
```

7.2. Тестирование полученной программы

Аналогично тестированию директивы for, по полученным запускам программы построим таблицу и визуализируем её с помощью 3d графиков

Таблица 3. Данные для директивы task.

	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
1000	1,1047	0,5394	0,4349	0,3754	0,3957	0,4051	0,5285	0,3784	0,3775	0,3791	0,2292	0,1843	0,2447	0,2574	0,4184	0,4857	0,5171	0,5481
1500	4,4155	1,8543	1,6284	1,3884	1,2294	1,2205	1,2643	1,0946	1,04942	1,07424	0,9712	0,7481	0,3051	1,8947	3,5724	4,581	5,5714	4,8764
2000	12,0574	4,9481	3,0487	2,6541	3,1744	2,5746	2,6257	2,5846	2,5164	2,0476	1,4495	1,0475	3,375	4,4615	7,4507	8,8646	9,9474	13,5714
2500	17,0573	8,4714	5,8975	4,6389	4,5605	4,3257	3,8957	3,5385	3,3395	3,9954	4,6471	4,8857	5,1173	7,7793	8,0841	12,543	17,5614	21,4749

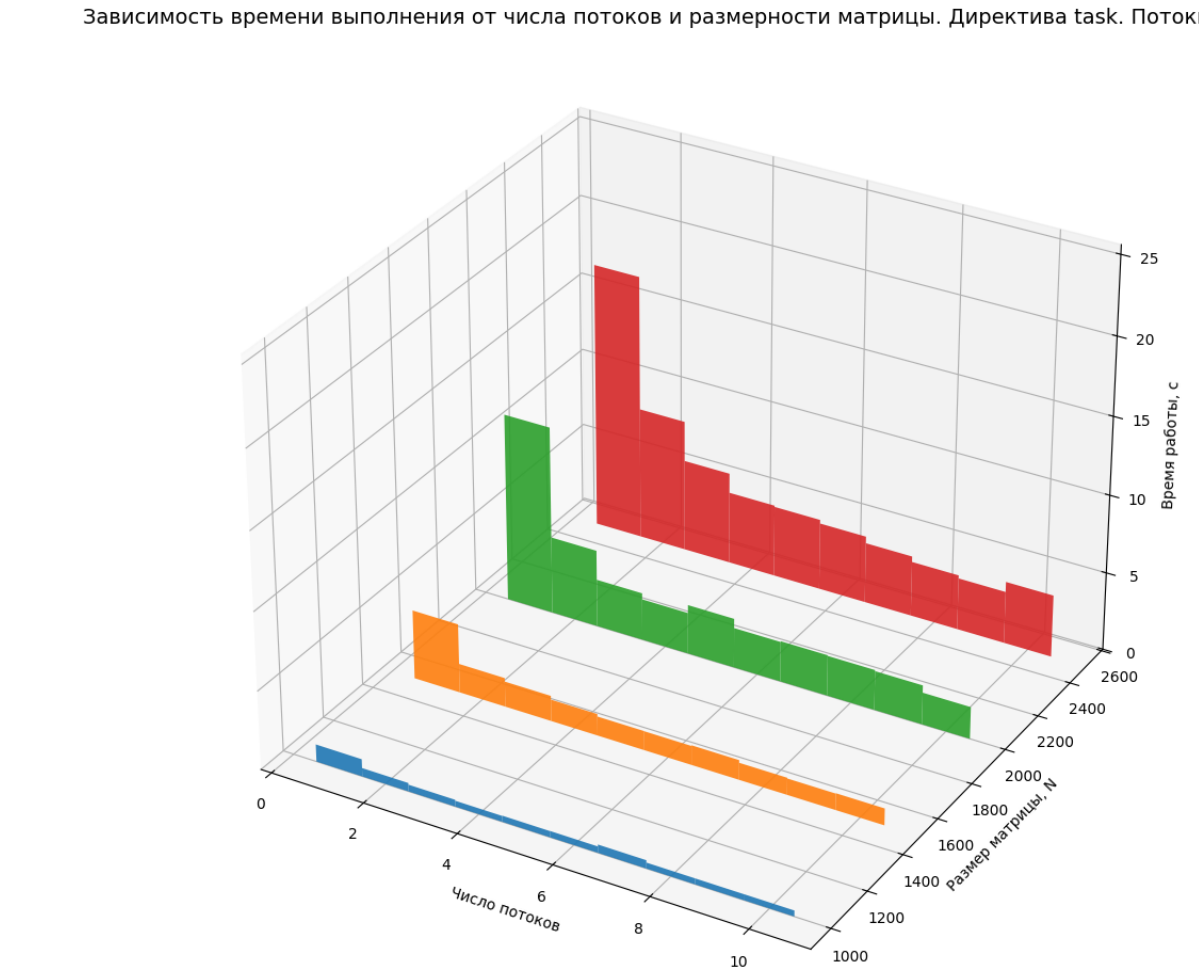


Диаграмма 4. Директива task. Потоки 1-10.

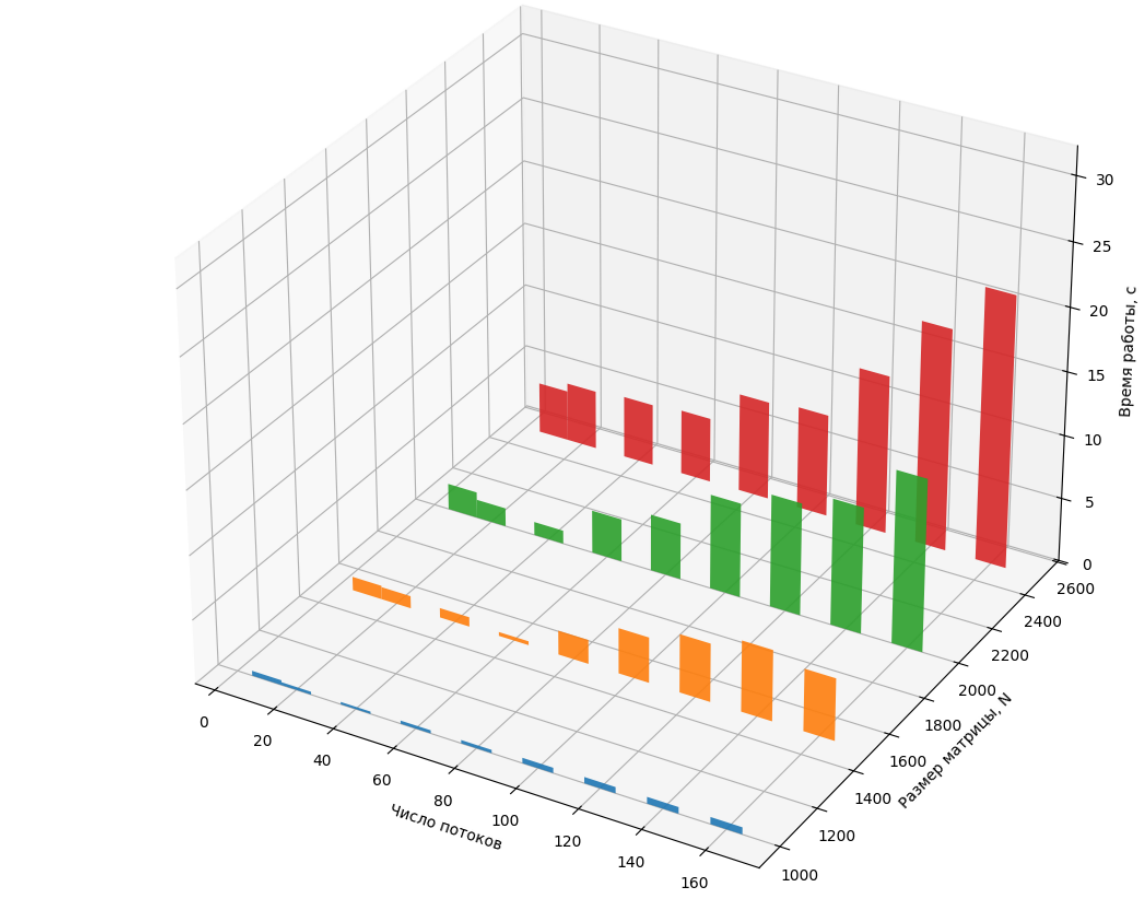


Диаграмма 5. Директива task. Потоки 10-160.

Из полученных результатов можно сделать следующий вывод. Использование директивы task на небольшом количестве нитей (<20) не просто ускоряет время выполнение программы, но и, как правило, превосходит по эффективности версию программы, реализованную с помощью директивы for. Тем самым получается ускорение в среднем в 10-12 раз. Однако, в отличие от for, из второго графика можно заметить, что директива task существенно проигрывает с увеличением числа потоков директиве for, при этом всё равно оставаясь в несколько раз лучше по скорости, чем исходная версия программы.

На мой взгляд, подобное происходит из-за относительной простоты самих задач, выполняемых с помощью директивы task. Таким образом, с увеличением числа потоков растут и накладные расходы на поддержание числа активных задач, что влечет за собой снижение эффективности программы.

8. Применение технологии MPI.

8.1. Описание внесенных изменений

В данном разделе предлагаю изучить одни из основных значительных изменений исходной программы.

Рассмотрим этапы алгоритма Гаусса для решения СЛАУ и применим к ним средства технологии MPI для распараллеливания.

Инициализация и подготовка

Написание любой MPI программы начинается, как правило, со следующих команд:

```
MPI_Init(&argc, &argv);

MPI_Comm_size(MPI_COMM_WORLD, &proc_num);

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

Первая команда инициализирует среду выполнения MPI, позволяя использовать MPI-функции в программе. Она должна быть вызвана первой в любой MPI-программе. Без нее остальные MPI-вызовы будут просто недействительными и бессмысленными.

`&argc` и `&argv` передаются для обеспечения возможности передачи параметров командной строки всем процессам MPI, однако в предлагаемой мною реализации это излишне, так как программа не предполагает получения каких-либо аргументов командной строки.

Следующая функция определяет общее количество процессов, участвующих в коммуникации в указанном коммуникаторе. Данное значение передается программе при компиляции с помощью команды `mpirun -np <N> ./a.out`

Заключительная функция определяет ранг текущего процесса в указанном коммуникаторе. Ранг можно понимать как идентификатор процесса в пределах коммуникатора, который используется для идентификации процесса в коммуникациях. Далее мы увидим, как он будет использоваться в вычислениях.

Следующим, на мой взгляд, важным изменением исходной программы является присвоение каждому MPI процессу «своих» столбцов, с которыми он будет работать в дальнейшем, ведь, согласно условию задания ни один процесс программы не должен работать с массивом целиком.

```
int j_start = (myrank * (N + 1)) / proc_num;
int j_end = ((myrank + 1) * (N + 1)) / proc_num;
myN = j_end - j_start;
```

Где:
`j_start` — начальный столбец.
`j_end` — конечный столбец (не включительно).
`myN` — количество столбцов для текущего процесса.

Также для выполнения условия задания по разбиению исходного массива на блоки было принято решение использовать пользовательский тип данных с помощью функций:

```
MPI_Datatype vectype;
MPI_Type_vector(N, 1, myN + 1, MPI_DOUBLE, &vectype);
MPI_Type_commit(&vectype);
```

Первая из них объявляет переменную специального типа, который в дальнейшем будет использоваться как пользовательский тип данных.

Вторая создает новый тип данных, который представляет собой вектор из блоков, расположенных в памяти с определенным шагом, что позволит каждому MPI процессу иметь свою область памяти для работы и значительно упростит процесс передачи столбцов матрицы между процессами.

В конце создания «нового» типа данных его необходимо зафиксировать, чтобы MPI мог использовать его в операциях. Именно этим занимается третья функция.

Распараллеливание прямого хода Гаусса

Основным изменением, что в прямом, что в обратном ходе, будет определение ответственного процесса за текущий столбец.

```
int index = i;
int proc_id = 0;
int j_start = 0;
int j_end = (N + 1) / proc_num;
while (j_end <= index)
{
    proc_id += 1;
    j_start = j_end;
    j_end = ((proc_id + 1) * (N + 1)) / proc_num;
}
index -= j_start;
```

Таким образом, для текущей итерации строки `i`, вычисляется, какой процесс `proc_id` "владеет" соответствующим столбцом, на основе равномерного разбиения столбцов между процессами.

Для дальнейших вычислений прямого хода нам необходимо передать главный элемент текущего процесса всем остальным:

```
int bcast = A(i, index);
MPI_Bcast(&bcast, 1, MPI_DOUBLE, proc_id, MPI_COMM_WORLD);
```

После чего необходимо скопировать элементы найденного столбца `index` в дополнительный столбец и передать его другим процессам опять же для его использования в последующих вычислениях:

```
for (int i = 0; i < N; i++)
    A(i, myN) = A(i, index);
MPI_Bcast(&A(0, myN), 1, vectype, proc_id, MPI_COMM_WORLD);
```

Затем в зависимости от номера текущего процесса необходимо определить для него столбцы для произведения вычислений прямого хода:

```
if (myrank < proc_id)
    j_start = myN;
else if (myrank > proc_id)
    j_start = 0;
else
    j_start = index + 1;
```

После чего требуется выполнить уже классические действия алгоритма Гаусса, позволяющие для текущей строки исключить ее переменную из нижестоящих в системе уравнений:

```
for (k = i + 1; k <= N - 1; k++)
    for (j = j_start; j < myN; j++)
        A(k, j) = A(k, j) - A(k, myN) * A(i, j) / bcast;
```

Распараллеливание обратного хода Гаусса

Сперва необходимо проверить, является ли текущий процесс последним, то есть нужно ли ему вычислять значение переменной из вектора решений:

```
if (myrank == proc_num - 1){
    X[N - 1] = A(N - 1, myN - 1) / A(N - 1, myN - 2);
}
```

После начинается непосредственно сам обратный ход Гаусса с уже обозначенного выше определения процесса, которому принадлежит столбец `j + 1`:

```
int index = j + 1;
int proc_id = 0;
int j_start = 0;
int j_end = (N + 1) / proc_num;
while (j_end <= index)
{
    proc_id += 1;
    j_start = j_end;
    j_end = ((proc_id + 1) * (N + 1)) / proc_num;
}
index -= j_start;
```

Затем для корректной обработки главного элемента требуется определить номер процесса, ответственного за текущую строку:

```
int cent_index = j;
int proc_id_2 = 0;
int j_start_2 = 0;
int j_end_2 = (N + 1) / proc_num;
while (j_end_2 <= cent_index)
{
    proc_id_2 += 1;
    j_start_2 = j_end_2;
    j_end_2 = ((proc_id_2 + 1) * (N + 1)) / proc_num;
}
cent_index -= j_start_2;
```

Далее аналогично прямому ходу для продолжения вычислений нужно распространить главный элемент остальным процессам.

```
int bcast = A(j, cent_index);
MPI_Bcast(&bcast, 1, MPI_DOUBLE, proc_id_2, MPI_COMM_WORLD);
```

А после передать столбец коэффициентов:

```
for (int i = 0; i < N; i++)
    A(i, myN) = A(i, index);
MPI_Bcast(&A(0, myN), 1, vectype, proc_id, MPI_COMM_WORLD);
```

И, если текущий процесс является последним, завершаем обратный ход Гаусса его классической реализацией с обновлением свободных членов для всех строк от 0 до j и вычислением `x[j]`:

```
if (myrank == proc_num - 1)
{
    for (k = 0; k <= j; k++)
        A(k, myN - 1) = A(k, myN - 1) - A(k, myN) * X[j + 1];
    X[j] = A(j, myN - 1) / bcast;
}
```

Завершение работы

Завершаем основную часть программы аннулированием использования ранее созданного производного типа данных (`vectype`), чтобы освободить ресурсы, выделенные для него. После вызова этой функции, переменная `vectype` становится невалидной и больше не может использоваться в MPI-операциях.

```
MPI_Type_free(&vectype);
```

8.2. Тестирование полученной программы

Измерим время работы программы с помощью специальной функции `MPI_Wtime()`, которая возвращает текущее время в секундах с момента, который MPI считает "нулевым" (обычно за это время принимается момент запуска программы). В качестве итоговых значений времени выполнения программы в зависимости от количества нитей и объема входных данных возьмем среднее значение по нескольким запускам.

	1	2	3	4	5	6	7	8	9	10	20	40	60	80	100	120	140	160
1000	1,1238	0,8247	0,5956	0,4647	0,5043	0,4258	0,3957	0,3359	0,3247	0,2841	0,2158	0,2257	0,1859	0,1552	0,1475	0,1258	0,1042	0,1945
1500	9,5712	6,9938	5,0465	4,9584	3,2358	2,9147	2,3209	2,4494	2,2653	1,9956	1,3204	1,1308	0,8314	0,6747	0,6274	0,5937	0,5352	0,8756
2000	15,4823	10,5812	5,8957	3,9746	3,6184	3,2142	2,5801	2,5158	2,4458	2,3731	1,8736	1,5484	1,3183	1,2746	1,0184	0,9248	0,8378	1,1532
2500	19,4713	13,5719	9,1245	7,4719	5,8811	5,0156	4,7514	4,1158	3,9846	3,4856	3,1351	2,9174	2,5361	1,9846	1,7842	1,5375	1,1285	1,9576

Таблица 4. Данные для технологии MPI.

Полученные табличные значения визуализируем на нескольких 3d графиках:

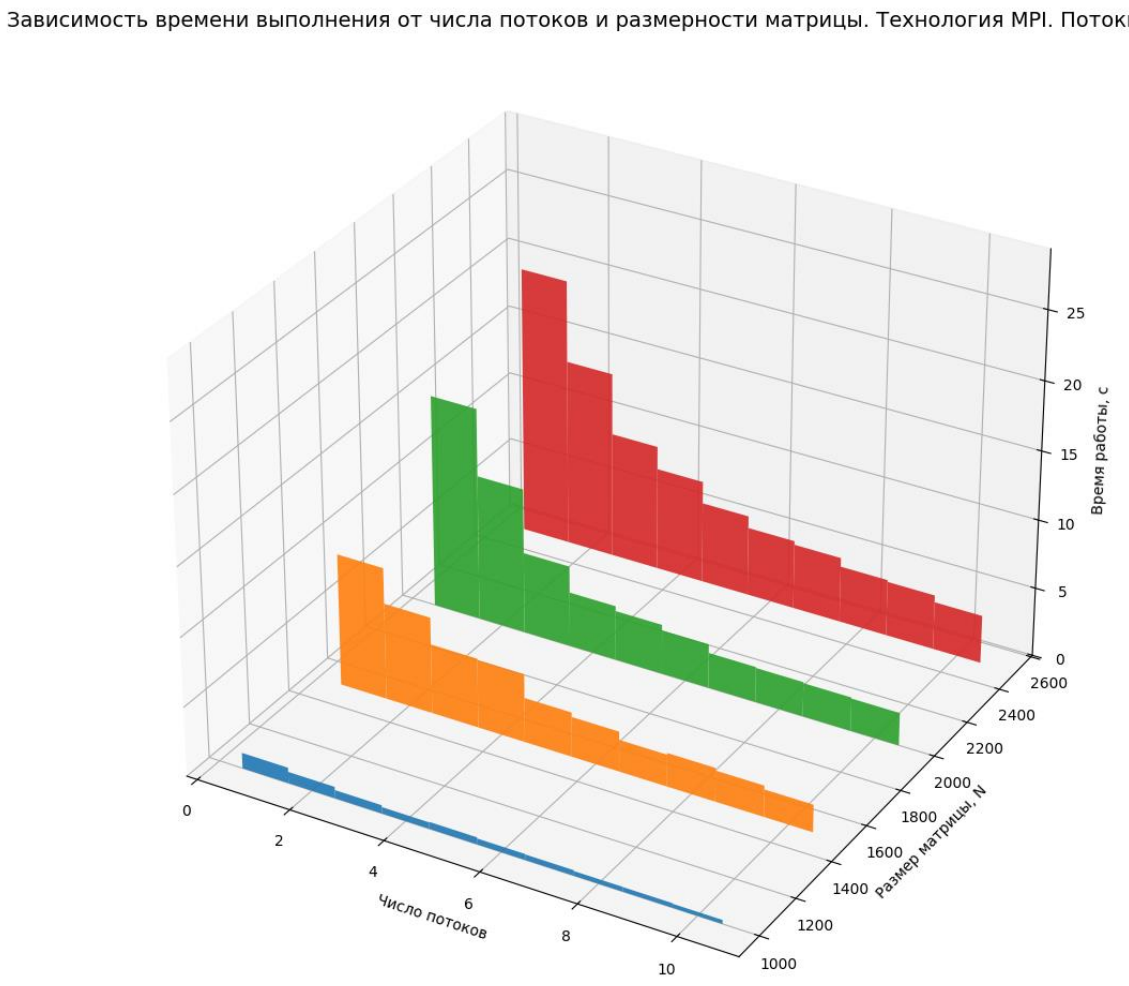


Диаграмма 6. Технология MPI. Потоки 1-10.

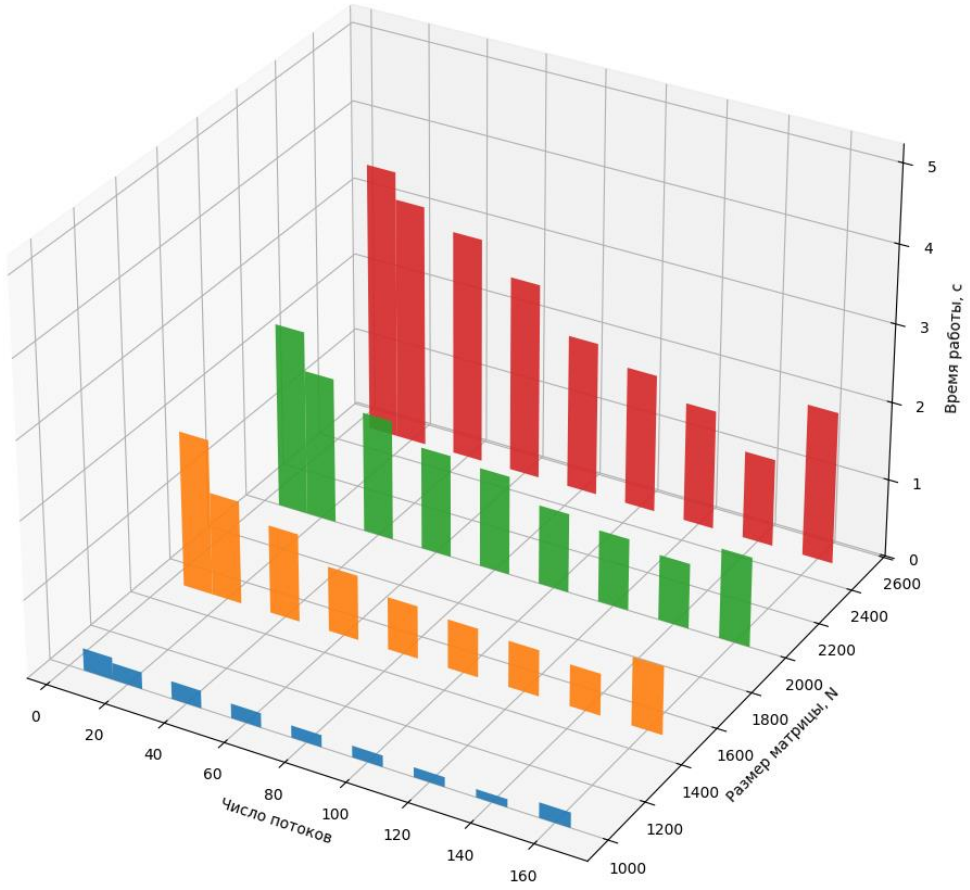


Диаграмма 7. Технология MPI. Потоки 10-160.

По результатам проведенных экспериментов можно сделать вывод, что технология MPI позволяет добиться наибольшего среди представленных технологий и реализаций прироста эффективности работы программы вплоть до 12-15 раз при использовании большого числа нитей – порядка 100-140.

9. Общий вывод

В результате выполнения практической работы в течение семестра мною были изучены технологии параллельного программирования OpenMP и MPI и разработаны с их помощью распараллеленные версии программы для задачи «Метод Гаусса решения системы линейных алгебраических уравнений».

Дополнительно была исследована эффективность и масштабируемость полученных программ на различных параметрах количества нитей и размерности входной матрицы при запуске на вычислительном кластере Polus, а также определенны основные причины недостаточной масштабируемости программ при максимальном числе используемых нитей/процессов.

В ходе выполнения практической работы и получения данных после проведения экспериментов удалось прийти к выводу, что подобная задача хорошо поддается распараллеливанию и позволяет уменьшить время своего выполнения в несколько раз.

Для каждой из реализованных версий программы были построены графики зависимости времени выполнения от числа нитей/процессов для различного объёма входных данных.

В конечном итоге хочется отметить, что использование технологий параллельной обработки данных позволяет порой существенно уменьшить время выполнения различных практических сложных вычислительных задач по сравнению с классической последовательной реализацией.

10. Дополнительно

К данному отчету прикрепляю на сайт следующие материалы:

- `gauss_base.c` – исходная версия программы
- `gauss_base_optimased.c` – модифицированная версия программы без использования технологии параллельной обработки данных
- `gauss_omp_for.c` – распараллеленная версия программы с помощью технологии openMP директивы for
- `gauss_omp_task.c` – распараллеленная версия программы с помощью технологии openMP директивы task
- `mpi_vf.c` – распараллеленная версия программы с использованием технологии MPI
- `final_report.pdf` – непосредственно сам отчет

Также весь реализованный код с различными версиями модификаций программ, скриптами для создания 3d графиков, сформированными таблицами данных, полученными графиками в формате `.png` можно изучить на `https://github.com/AlexGudis/SkiPod`