МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью MPI»

студента 2 курса, 21204 группы

Осипова Александра Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Кандидат технических наук
А. Ю. Власенко

Новосибирск 2023

СОДЕРЖАНИЕ

ЦЕЛЬ

Реализовать параллельную программу, вычисляющую приближенное решение СЛАУ с помощью метода сопряженных градиентов, используя библиотеку MPI.

ЗАДАНИЕ

1. Написать 2 программы (последовательную и параллельную с использованием MPI) на языке C/C++, которые реализуют итерационный алгоритм решения системы линейных алгебраических уравнений вида $Ax=b$ методом сопряженных градиентов. Здесь A– матрица размером N×N, x и b – векторы длины N. Тип элементов – double.

   В методе сопряженных градиентов преобразование решения на каждом шаге задается следующими формулами:

$$r^0 = b - Ax^0,$$
$$z^0 = r^0,$$
$$\alpha^{n+1} = \frac{(r^n, r^n)}{(Az^n, z^n)},$$
$$x^{n+1} = x^n + \alpha^{n+1} z^n,$$
$$r^{n+1} = r^n - \alpha^{n+1} Az^n,$$
$$\beta^{n+1} = \frac{(r^{n+1}, r^{n+1})}{(r^n, r^n)},$$
$$z^{n+1} = r^{n+1} + \beta^{n+1} z^n.$$

   где $(u, v) = \sum_{i=0}^{N-1} u_i v_i$. Критерий завершения счета в методе сопряженных градиентов следующий:

$$(u, v) = \sum_{i=0}^{N-1} u_i v_i,$$

   где $\|u\|_2 = \sqrt{\sum_{i=0}^{N-1} u_i^2}$.

2. Параллельную программу реализовать с тем условием, что матрица A и вектор b инициализируются на каком-либо одном процессе, а затем матрица A «разрезается» по строкам на близкие по размеру, возможно не одинаковые, части, а вектор b раздается каждому процессу.

3. Замерить время работы последовательного варианта программы, а также время работы параллельного при использовании различного числа процессорных ядер (2, 4, 8, 16, 24). Построить графики

зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер.

4. Выполнить профилирование двух вариантов программы с помощью jumpshot или ITAC (Intel Trace Analyzer and Collecter) при использовании 16-и или 24-х ядер.

5. На основании полученных результатов сделать вывод.

ОПИСАНИЕ РАБОТЫ

Для чистоты эксперимента два варианта программы были запущены на вычислительном кластере НГУ и для матрицы коэффициентов A, вектора правых частей b и вектора начального решения x были использованы одинаковые значения.

Вектор x инициализируется нулями.

```
double* InitPreSolution(int N) {
	double* vector = (double*)malloc(N * sizeof(double));
	for (int i = 0; i < N; ++i) {
		vector[i] = 0.0;
	}
	return vector;
}
```

Вектор b инициализируется следующим образом: b[i] = i + 1

```
double* InitVectorB(double* A, int N) {
	double* b = (double*)malloc(N * sizeof(double));
	for (int i = 0; i < N; ++i) {
		b[i] = i + 1;
	}
	return b;
}
```

Для генерации коэффициентов матрицы A и ее инициализации были написаны следующие функции:

```
void GenerateMatrixA(int N) {
	double* A = (double*)malloc(N * N * sizeof(double));
	for (int i = 0; i < N; ++i) {
		for (int j = 0; j < N; ++j) {
			double value = (double)rand() / RAND_MAX + (i == j ?
(double)SIZE / 100 : 0);
			A[i * N + j] = (i > j ? A[j * N + i] : value);
		}
	}

	std::ofstream fileIn;
	fileIn.open("A.txt");
	if (fileIn.is_open()) {
		for (int i = 0; i < N * N; ++i) {
			fileIn << A[i] << std::endl;
		}
		fileIn.close();
	}
}
```

```
double* InitMatrixA(int N) {
	std::ifstream file;
	file.open("A.txt");
```

```
    double* A = (double*)malloc(N * N * sizeof(double));
    for (int i = 0; i < N * N; ++i) {
          file >> A[i];
    }

    file.close();
    return A;
}
```

Последовательная программа (см. Приложение 1)

Функция, реализующая алгоритм вычисления приближенного решения СЛАУ с помощью метода сопряженных градиентов.

```
void ConjugateGradientMethod(double* A, double* x, double* b) {
     double* r = InitVectorR(A, x, b);
     double* r_next = (double*)malloc(SIZE * sizeof(double));
     double* z = (double*)malloc(SIZE * sizeof(double));
     CopyMatrix(r, z, SIZE);

     double alpha = 0.0;
     double beta = 0.0;

     int count = 0;

     while (!isSolutionReached(r, b) && (count < 50000)) {
          alpha = CalcNextAlpha(A, r, z);
          CalcNextX(x, z, alpha);
          CalcNextR(A, r, z, alpha, r_next);
          beta = CalcNextBeta(r_next, r);
          CalcNextZ(beta, r_next, z);
          CopyMatrix(r_next, r, SIZE);

          ++count;
     }
     printf("Number of iterations: %d\n", count);
}
```

Функция `int main()`. В ней поставлен таймер, определяющий, за какое время программа проинициализирует матрицу A, векторы b и x, а так же найдет приближенное значения вектора решений. Для этих целей был выбран таймер системного времени time(time_t *_Time).

```
int main() {
     GenerateMatrixA(SIZE);

     time_t startTime, endTime;
     time(&startTime);
     double* A = InitMatrixA(SIZE);
     double* x = InitPreSolution(SIZE);
     double* b = InitVectorB(A, SIZE);
```

```
	ConjugateGradientMethod(A, x, b);
	time(&endTime);

	printf("Time: %f sec\n\n", difftime(endTime, startTime));

	free(A);
	free(x);
	free(b);
}
```

Для того, чтобы программа работала больше 30 секунд, была выбрана матрица размерностью 1500x1500 и в качестве критерия завершения счета было выбрано значения эпсилона в 0.00001.

Результат работы программы (см. Приложение 3):

Количество итераций, совершенных при вычислении x	Время, сек
1804	106

Параллельная программа (см. Приложение 2)

Для реализации параллельного выполнения строки матрицы A разделяются между процессами. Таким образом, она делится на n частей, где n – это число запущенных MPI-процессов. Заметим, что строки матрицы A могут разделится на равномерно, если число строк не кратно числу процессов.

Функция вычисляющая, сколько элементов матрицы A должно быть отдано i-му процессу.

```
int* CalcSizesOfAp(int numProc, int totalLines, int totalColumns) {
	int* sizes = (int*)malloc(numProc * sizeof(int));
	for (int i = 0; i < numProc; ++i) {
		sizes[i] = totalLines / numProc;
		if (i < totalLines % numProc) ++sizes[i];
		sizes[i] *= totalColumns;
	}
	return sizes;
}
```

Вместе с ней была написана функция, вычисляющая, смещение в байтах в матрице A относительно первого элемента для каждого процесса. Это смещение необходимо для дальнейших коллективных операций и пользовательского кода.

```
int* CalcOffsetsOfAp(const int* sizes, int numProc) {
	int* offsets = (int*)malloc(numProc * sizeof(int));
	int offset = 0;
	for (int i = 0; i < numProc; ++i) {
		offsets[i] = offset;
		offset += sizes[i];
	}
	return offsets;
}
```

Для реализации параллельного вычисления нулевой процесс (rank == 0) инициализирует матрицу A и вектор b (вектор x инициализируется на всех процессах), аналогично последовательной программе, и далее отправляет каждому процессу свою часть матрицы A и весь вектор b с помощью коллективных операций MPI_Scatterv и MPI_Bcast соответственно.

```
	...
	if (rank == 0) {
		InitMatrixA(A, SIZE);
		InitVectorB(b, SIZE);
	}

	double* Ap = (double*)malloc(sizes[rank] * sizeof(double));

	MPI_Scatterv(A, sizes, offsets, MPI_DOUBLE, Ap, sizes[rank], MPI_DOUBLE, 0,
MPI_COMM_WORLD);
	MPI_Bcast(b, SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD);
	...
```

Сейчас массивы sizes и offsets хранят количество элементов матрицы A, отданному i-му процессу, и смещение относительно начала матрицы. В дальнейшем, нам понадобится лишь число строк у каждого процесса и смещение относительно них. Для этого мы переопределяем значения для данных массивов.

```
	...
	for (int i = 0; i < numProc; ++i) {
		sizes[i] /= SIZE;
		offsets[i] /= SIZE;
	}
	...
```

Функция, вычисляющая приближенное решение СЛАУ методом сопряженных градиентов. Здесь массивы tmp1 и tmp2 служат буферами для результатов промежуточных вычислений, необходимых для алгоритма. Реализация функций по расчету вспомогательных параметров находится в Приложении 2.

```
void ConjugateGradientMethod(double* Ap, double* b, double* x, int*
sizes, int* offsets, int rank) {
	double* tmp1 = (double*)malloc(SIZE * sizeof(double));
```

```
	double* tmp2 = (double*)malloc(SIZE * sizeof(double));
	double* r = InitVectorR(Ap, b, x, tmp1, sizes, offsets, rank);
	double dotR = 0;

	double* z = (double*)malloc(SIZE * sizeof(double));
	CopyMatrix(r, z, sizes[rank], offsets[rank]);

	MPI_Allgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, z, sizes,
offsets, MPI_DOUBLE, MPI_COMM_WORLD);

	double alpha = 0.0;
	double beta = 0.0;

	int count = 0;

	while (!IsSolutionReached(b, r, sizes, offsets, rank) && (count <
50000)) {
		CalcNextAlpha(&alpha, Ap, r, z, &dotR, sizes, offsets, rank,
tmp1);
		CalcNextXp(x, z, sizes, offsets, alpha, rank, tmp2);
		CalcNextRp(r, z, alpha, sizes, offsets, rank, tmp1, tmp2);
		CalcNextBeta(&beta, r, &dotR, sizes, offsets, rank);
		CalcNextZp(z, r, beta, sizes, offsets, rank);
		MPI_Allgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, z, sizes,
offsets, MPI_DOUBLE, MPI_COMM_WORLD);
		++count;
	}

	MPI_Allgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, x, sizes,
offsets, MPI_DOUBLE, MPI_COMM_WORLD);

	free(tmp1);
	free(tmp2);
}
```

Для подсчета времени был выбран таймер системного времени MPI_Wtime.

Функция, выводящая результат работы программы, а именно:
- количество MPI процессов (numProc)
- время работы программы (totalTime)
- ускорение (boost)
- эффективность (efficiency)

Здесь T1 = 106, время работы последовательной программы.

```
void PrintResult(float totalTime, int numProc) {
	float boost = T1 / totalTime;
	float efficiency = (boost / (float)numProc) * 100;
	printf("Number of processes: %d\n", numProc);
	printf("Total time: %f sec\n", totalTime);
	printf("Sp = %f\n", boost);
	printf("Ep = %f\n\n", efficiency);
}
```

Результат работы программы (см. Приложение 3)

Число процессов	Время, сек	Ускорение	Эффективность, %
2	27.037	3.9	196.02
4	14.851	7.137	178.42
8	10.18	10.412	130.15
16	7.744	13.687	85.54
24	7.253	14.614	60.892

График времени

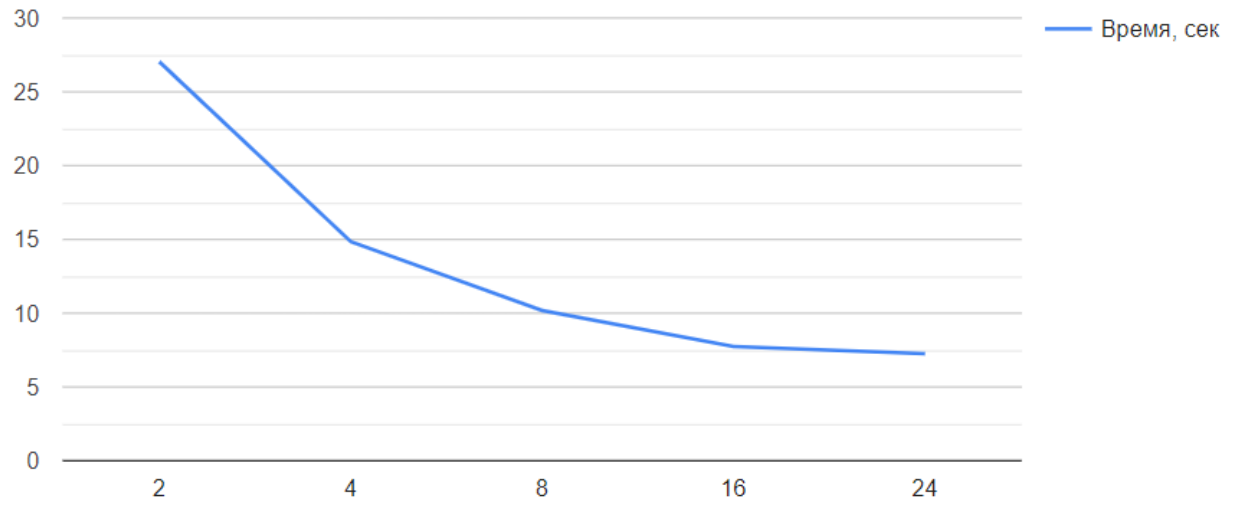


График ускорения

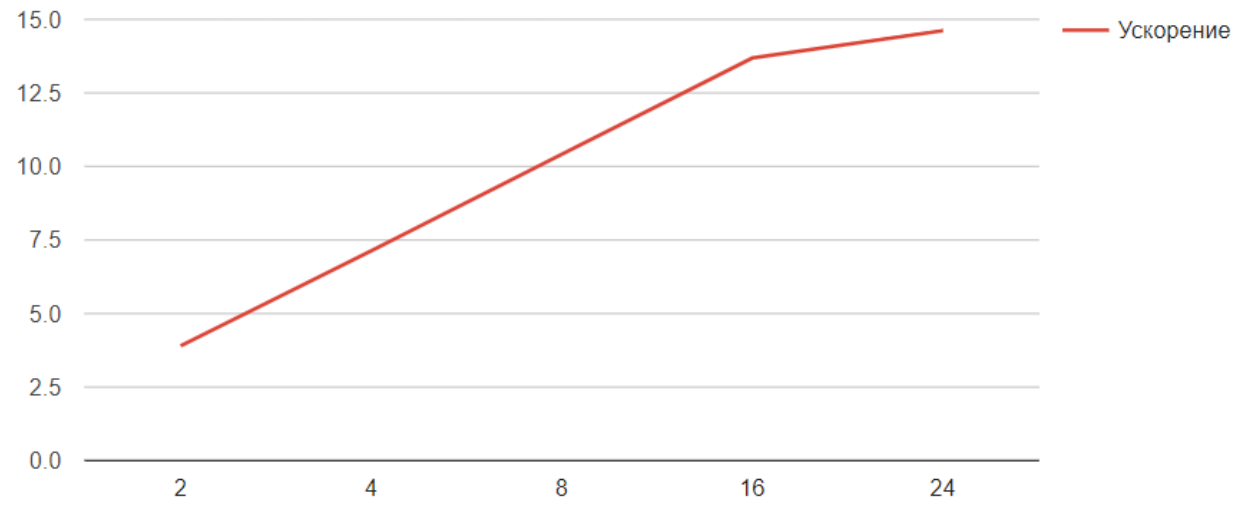
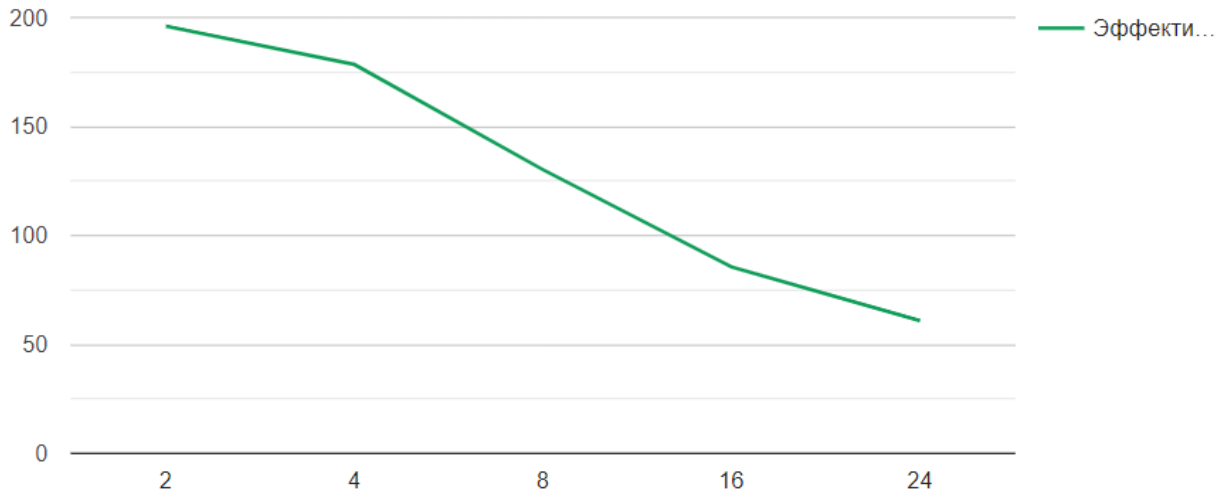


График эффективности



Профилирование

Профилирование было выполнено для параллельной программы с 24 процессами.

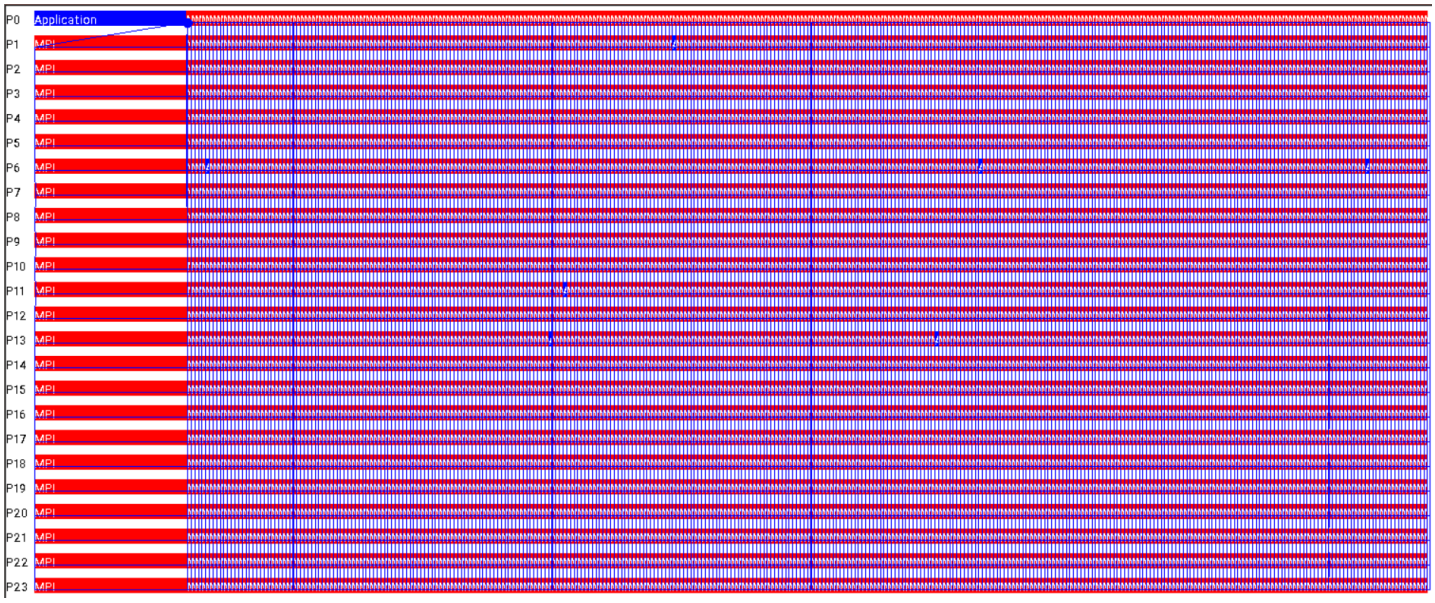


Рис.1

На Рис.2 видно, что пока нулевой процесс инициализирует матрицу A, оставшиеся процессы простаивают в ожидании его, пока он не передаст им часть матрицы A (MPI_Scatterv).

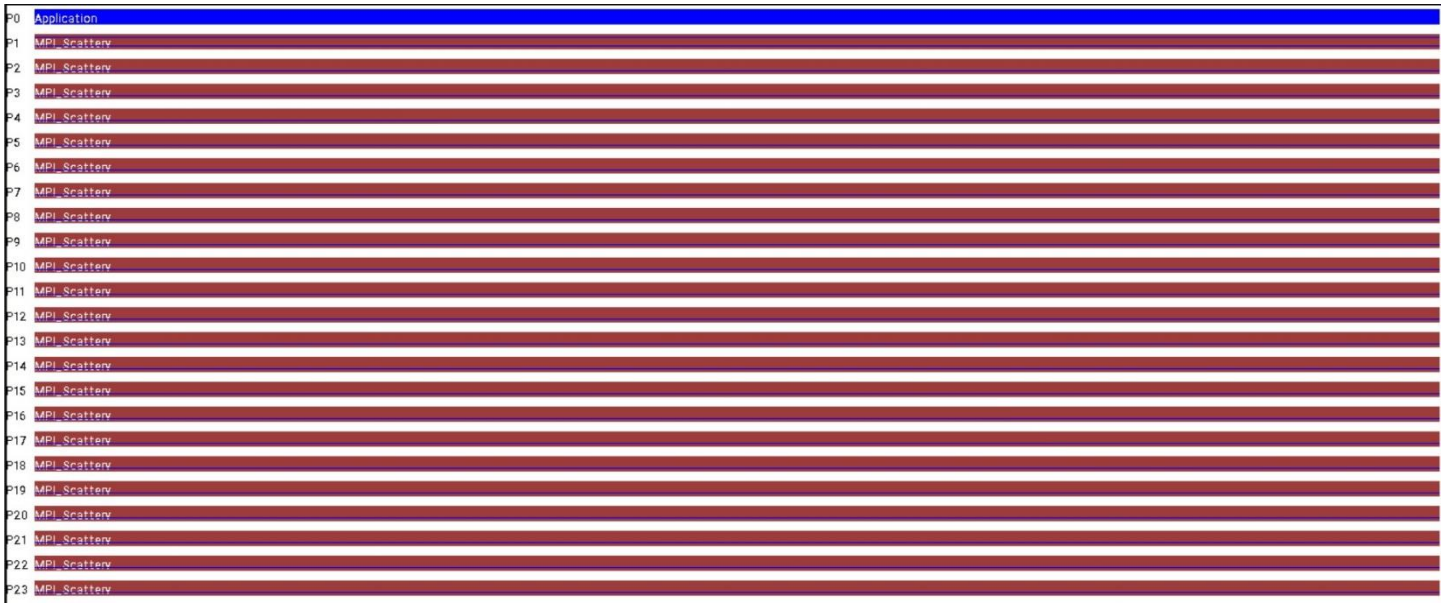


Рис.2

Дождавшись от нулевого процесса свою часть матрицы A, другие процессы начинают ждать от него целый вектор b (MPI_Bcast).

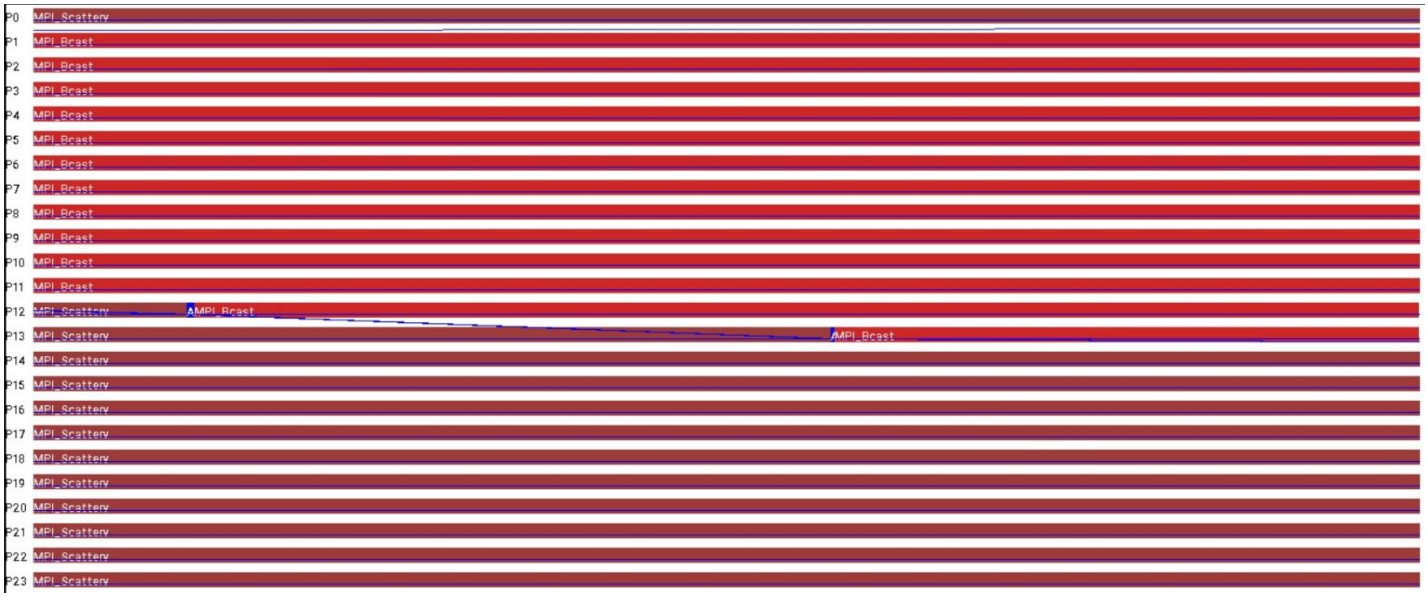


Рис.3

На Рис.4 изображена работа цикла в функции ConjugateGradientMethod. Как видно, пользовательский код и MPI функции чередуются вполне равномерно. Это достигается за счет функции MPI_Allreduce, так как из-за нее все процессы ждут друг друга.

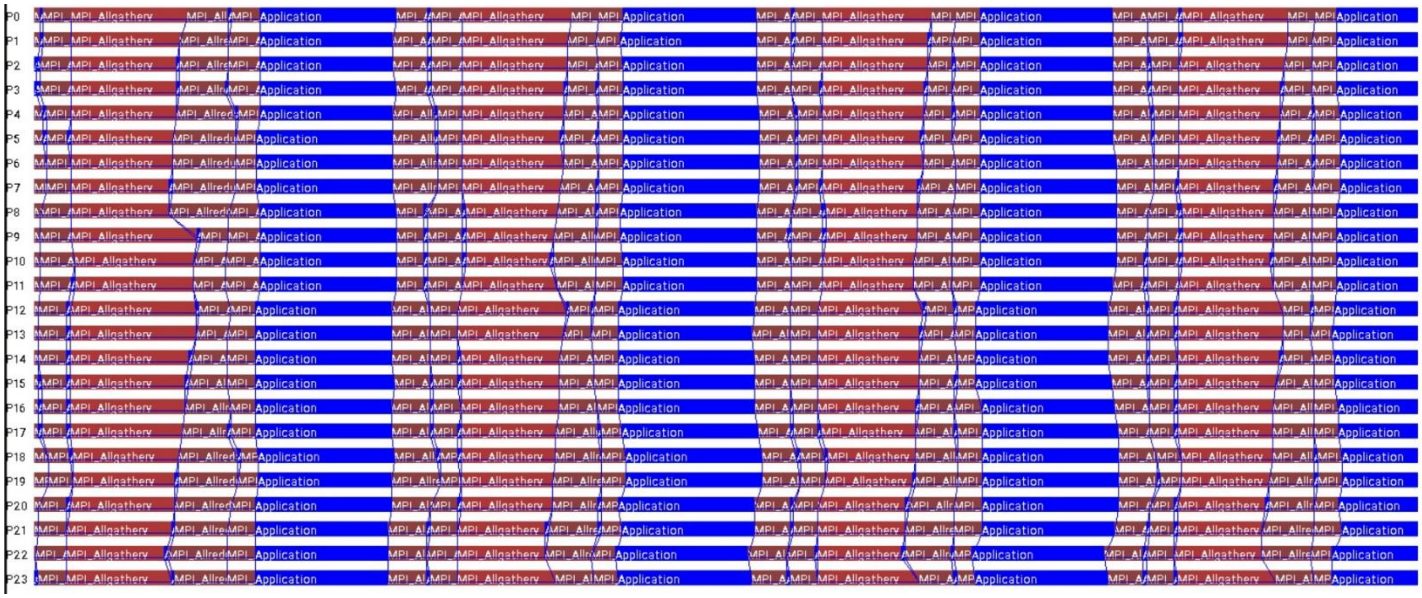


Рис.4

На Рис.5 видно, что после функции MPI_Allgetherv, которая собирает решение у всех процессов, нулевой процесс задерживается на функции PrintResult, в то время как другие процессы заканчивают свою работу (MPI_Finalize).

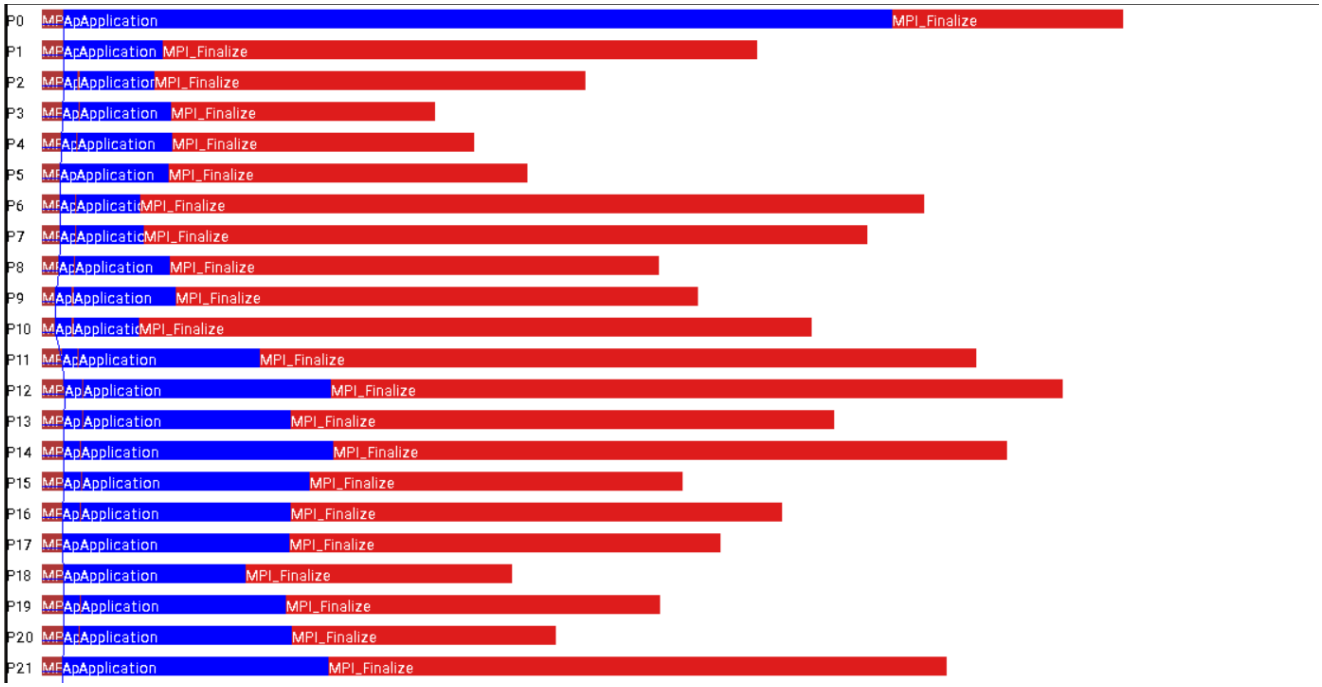


Рис.5

Общая статистика:

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	74.7608 s		174.199 s	24	3.11503 s
Group MPI	99.4386 s		99.4386 s	6000264	16.5724e-6 s

Рис.6

Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group All_Processes					
Group Application	74.7608 s		174.199 s	24	3.11503 s
MPI_Comm_size	16e-6 s		16e-6 s	24	666.667e-9 s
MPI_Comm_rank	16e-6 s		16e-6 s	24	666.667e-9 s
MPI_Finalize	12.855e-3 s		12.855e-3 s	24	535.625e-6 s
MPI_Bcast	59.832e-3 s		59.832e-3 s	24	2.493e-3 s
MPI_Wtime	73e-6 s		73e-6 s	48	1.52083e-6 s
MPI_Allgatherv	35.1916 s		35.1916 s	1200048	29.3252e-6 s
MPI_Scatterv	18.1992 s		18.1992 s	24	758.3e-3 s
MPI_Allreduce	45.975 s		45.975 s	4800048	9.57804e-6 s

Рис.7

ЗАКЛЮЧЕНИЕ

Выполненная практическая работа показала, как сильно можно ускорить выполнение программы – в данном случае математического алгоритма по решению СЛАУ – с помощью MPI инструментов. Время, проделанное параллельной программой, запущенной на 24 процессах, оказалось в ~14 раз меньше, чем у последовательной. Так же при разработки MPI программ необходимо учитывать тонкости при использовании коллективных операций, которые могут сильно затормозить процесс выполнения за счет того, что одни процессы будут «простаивать» в ожидании других процессов.

Приложение 1. Листинг последовательной программы

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

#define SIZE 1500
#define EPSILON 0.00001

void PrintVector(double* vector, int N) {
	for (int i = 0; i < N; ++i) {
		printf("%f ", vector[i]);
	}
	printf("\n");
}

void PrintMatrix(double* matrix, int N) {
	for (int i = 0; i < N; ++i) {
		for (int j = 0; j < N; ++j) {
			printf("%f ", matrix[i * N + j]);
		}
		printf("\n");
	}
	printf("\n");
}

void CopyMatrix(double* source, double* purpose, int size) {
	for (int i = 0; i < size; ++i) {
		purpose[i] = source[i];
	}

}

void MatrixMULT(double* A, double* B, double* C, int L, int M, int N) {
	for (int i = 0; i < L; ++i) {
		for (int j = 0; j < N; ++j) {
			C[i * N + j] = 0;
			for (int k = 0; k < M; ++k) {
				C[i * N + j] += A[i * M + k] * B[k * N + j];
			}
		}
	}
}

void MatrixSUB(double* A, double* B, double* C, int N) {
	for (int i = 0; i < N; ++i) {
		C[i] = A[i] - B[i];
	}
}

void MatrixADD(double* A, double* B, double* C, int N) {
	for (int i = 0; i < N; ++i) {
		C[i] = A[i] + B[i];
	}
}
```

```
void ScalarMULT(double scalar, double* matrix, double* result, int N) {
    for (int i = 0; i < N; ++i) {
        result[i] = scalar * matrix[i];
    }
}

double DotProduct(double* a, double* b) {
    double dot = 0;
    for (int i = 0; i < SIZE; ++i) {
        dot += (a[i] * b[i]);
    }
    return dot;
}

double Norm(double* vector) {
    double norm = 0;
    for (int i = 0; i < SIZE; ++i) {
        norm += (vector[i] * vector[i]);
    }
    return sqrt(norm);
}

double* InitVectorR(double* A, double* x, double* b) {
    double* r = (double*)malloc(SIZE * sizeof(double));
    double* tmp = (double*)malloc(SIZE * sizeof(double));
    MatrixMULT(A, x, tmp, SIZE, SIZE, 1);
    MatrixSUB(b, tmp, r, SIZE);
    free(tmp);
    return r;
}

void CalcNextX(double* x, double* z, double alpha) {
    double* tmp = (double*)malloc(SIZE * sizeof(double));
    ScalarMULT(alpha, z, tmp, SIZE);
    MatrixADD(x, tmp, x, SIZE);
    free(tmp);
}

void CalcNextR(double* A, double* r, double* z, double alpha, double*
r_next) {
    double* tmp = (double*)malloc(SIZE * sizeof(double));
    MatrixMULT(A, z, tmp, SIZE, SIZE, 1);
    ScalarMULT(alpha, tmp, tmp, SIZE);
    MatrixSUB(r, tmp, r_next, SIZE);
    free(tmp);
}

void CalcNextZ(double beta, double* r_next, double* z) {
    ScalarMULT(beta, z, z, SIZE);
    MatrixADD(r_next, z, z, SIZE);
}

double CalcNextAlpha(double* A, double* r, double* z) {
    double* tmp = (double*)malloc(SIZE * sizeof(double));
    MatrixMULT(A, z, tmp, SIZE, SIZE, 1);
    double alpha = DotProduct(r, r) / DotProduct(tmp, z);
```

```
    free(tmp);
    return alpha;
}

double CalcNextBeta(double* r_next, double* r) {
    return DotProduct(r_next, r_next) / DotProduct(r, r);
}

bool isSolutionReached(double* r, double* b) {
    return (Norm(r) / Norm(b)) < EPSILON;
}

void ConjugateGradientMethod(double* A, double* x, double* b) {
    double* r = InitVectorR(A, x, b);
    double* r_next = (double*)malloc(SIZE * sizeof(double));
    double* z = (double*)malloc(SIZE * sizeof(double));
    CopyMatrix(r, z, SIZE);

    double alpha = 0.0;
    double beta = 0.0;

    int count = 0;

    while (!isSolutionReached(r, b) && (count < 50000)) {
        alpha = CalcNextAlpha(A, r, z);
        CalcNextX(x, z, alpha);
        CalcNextR(A, r, z, alpha, r_next);
        beta = CalcNextBeta(r_next, r);
        CalcNextZ(beta, r_next, z);
        CopyMatrix(r_next, r, SIZE);
        ++count;
    }
    printf("Number of iterations: %d\n", count);
}

double* InitMatrixA(int N) {
    std::ifstream file;
    file.open("A.txt");

    double* A = (double*)malloc(N * N * sizeof(double));
    for (int i = 0; i < N * N; ++i) {
        file >> A[i];
    }

    file.close();
    return A;
}

double* InitVectorB(double* A, int N) {
    double* b = (double*)malloc(N * sizeof(double));
    for (int i = 0; i < N; ++i) {
        b[i] = i + 1;
    }
    return b;
}
```

```
double* InitPreSolution(int N) {
     double* x = (double*)malloc(N * sizeof(double));
     for (int i = 0; i < N; ++i) {
          x[i] = 0.0;
     }
     return x;
}

void GenerateMatrixA(int N) {
     double* A = (double*)malloc(N * N * sizeof(double));
     for (int i = 0; i < N; ++i) {
          for (int j = 0; j < N; ++j) {
               double value = (double)rand() / RAND_MAX + (i == j ?
(double)SIZE / 100 : 0);
               A[i * N + j] = (i > j ? A[j * N + i] : value);
          }
     }

     std::ofstream fileIn;
     fileIn.open("A.txt");
     if (fileIn.is_open()) {
          for (int i = 0; i < N * N; ++i) {
               fileIn << A[i] << std::endl;
          }
          fileIn.close();
     }
}


int main() {
     GenerateMatrixA(SIZE);

     time_t startTime, endTime;
     time(&startTime);
     double* A = InitMatrixA(SIZE);
     double* x = InitPreSolution(SIZE);
     double* b = InitVectorB(A, SIZE);
     ConjugateGradientMethod(A, x, b);
     time(&endTime);

     printf("Time: %f sec\n\n", difftime(endTime, startTime));

     free(A);
     free(x);
     free(b);
}
```

Приложение 2. Листинг параллельной программы

```
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <fstream>

#define SIZE 1500
#define EPSILON 0.00001
#define T1 106.0

void PrintMatrix(double* matrix, int size) {
	for (int i = 0; i < size; ++i) {
		for (int j = 0; j < size; ++j) {
			printf("%f  ", matrix[i * size + j]);
		}
		printf("\n");
	}
	printf("\n");
}

void PrintVectorInt(int* vector, int size) {
	for (int i = 0; i < size; ++i) {
		printf("%d ", vector[i]);
	}
	printf("\n\n");
}

void PrintVectorDouble(double* vector, int size) {
	for (int i = 0; i < size; ++i) {
		printf("%f  ", vector[i]);
	}
	printf("\n\n");
}

void InitMatrixA(double* A, int N) {
	std::ifstream file;
	file.open("A.txt");
	for (int i = 0; i < N * N; ++i) {
		file >> A[i];
	}
	file.close();
}

void InitVectorB(double* b, int N) {
	for (int i = 0; i < N; ++i) {
		b[i] = i + 1;
	}
}

void InitPreSolution(double* x, int N) {
	for (int i = 0; i < N; ++i) {
		x[i] = 0.0;
	}
}
```

```
int* CalcSizesOfAp(int numProc, int totalLines, int totalColumns) {
    int* sizes = (int*)malloc(numProc * sizeof(int));
    for (int i = 0; i < numProc; ++i) {
        sizes[i] = totalLines / numProc;
        if (i < totalLines % numProc) ++sizes[i];
        sizes[i] *= totalColumns;
    }
    return sizes;
}

int* CalcOffsetsOfAp(const int* sizes, int numProc) {
    int* offsets = (int*)malloc(numProc * sizeof(int));
    int offset = 0;
    for (int i = 0; i < numProc; ++i) {
        offsets[i] = offset;
        offset += sizes[i];
    }
    return offsets;
}

void CopyMatrix(double* source, double* purpose, int size, int begin) {
    for (int i = 0; i < size; ++i) {
        purpose[i + begin] = source[i + begin];
    }
}

void MatrixMULT(double* matrix, double* vector, double* result, int M, int
N, int begin) {
    for (int i = 0; i < M; ++i) {
        result[i + begin] = 0;
        for (int j = 0; j < N; ++j) {
            result[i + begin] += matrix[i * N + j] * vector[j];
        }
    }
}

void MatrixSUB(double* A, double* B, double* C, int N, int begin) {
    for (int i = 0; i < N; ++i) {
        C[i + begin] = A[i + begin] - B[i + begin];
    }
}

void MatrixADD(double* A, double* B, double* C, int size, int begin) {
    for (int i = 0; i < size; ++i) {
        C[begin + i] = A[begin + i] + B[begin + i];
    }
}

void ScalarMULT(double scalar, double* matrix, double* result, int size, int
begin) {
    for (int i = 0; i < size; ++i) {
        result[begin + i] = scalar * matrix[begin + i];
    }
}
```

```
double DotProduct(double* a, double* b, int size, int begin) {
     double dot = 0.0;
     for (int i = 0; i < size; ++i) {
          dot += (a[i + begin] * b[i + begin]);
     }
     return dot;
}

double FakeNorm(double* vector, int size, int begin) {
     double fakeNorm = 0;
     for (int i = begin; i < size + begin; ++i) {
          fakeNorm += (vector[i] * vector[i]);
     }
     return fakeNorm;
}

void CalcNextAlpha(double* alpha, double* Ap, double* r, double* z, double*
dotNomin, int* sizes, int* offsets, int rank, double* tmp) {
     double dotRp = DotProduct(r, r, sizes[rank], offsets[rank]);
     MPI_Allreduce(&dotRp, dotNomin, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

     MatrixMULT(Ap, z, tmp, sizes[rank], SIZE, offsets[rank]);
     double dotTmp = DotProduct(tmp, z, sizes[rank], offsets[rank]);
     double dotDenom = 0.0;
     MPI_Allreduce(&dotTmp, &dotDenom, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

     *alpha = (*dotNomin) / dotDenom;
}

void CalcNextBeta(double* beta, double* r, double* dotR, int* sizes, int*
offsets, int rank) {
     double tmp = DotProduct(r, r, sizes[rank], offsets[rank]);
     double dotRnext = 0;
     MPI_Allreduce(&tmp, &dotRnext, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
     *beta = dotRnext / (*dotR);
}

double* InitVectorR(double* Ap, double* b, double* x, double* tmp, int*
sizes, int* offsets, int rank) {
     double* r = (double*)malloc(SIZE * sizeof(double));
     MatrixMULT(Ap, x, tmp, sizes[rank], SIZE, offsets[rank]);
     MatrixSUB(b, tmp, r, sizes[rank], offsets[rank]);
     return r;
}

void CalcNextXp(double* x, double* z, int* sizes, int* offsets, double
alpha, int rank, double* tmp) {
     ScalarMULT(alpha, z, tmp, sizes[rank], offsets[rank]);
     MatrixADD(tmp, x, x, sizes[rank], offsets[rank]);
}

void CalcNextRp(double* r, double* z, double alpha, int* sizes, int*
offsets, int rank, double* tmp1, double* tmp2) {
     ScalarMULT(alpha, tmp1, tmp2, sizes[rank], offsets[rank]);
     MatrixSUB(r, tmp2, r, sizes[rank], offsets[rank]);
```

```
}

void CalcNextZp(double* z, double* r, double beta, int* sizes, int* offsets,
int rank) {
    ScalarMULT(beta, z, z, sizes[rank], offsets[rank]);
    MatrixADD(r, z, z, sizes[rank], offsets[rank]);
}

double CalcFakeNormB(double* b, int* sizes, int* offsets, int rank) {
    static const double fakeNormBp = FakeNorm(b, sizes[rank],
offsets[rank]);
    double fakeNormB = 0.0;
    MPI_Allreduce(&fakeNormBp, &fakeNormB, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);
    return fakeNormB;
}

bool IsSolutionReached(double* b, double* r, int* sizes, int* offsets, int
rank) {
    double fakeNormRp = FakeNorm(r, sizes[rank], offsets[rank]);
    double fakeNormR = 0.0;
    MPI_Allreduce(&fakeNormRp, &fakeNormR, 1, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

    static const double fakeNormB = CalcFakeNormB(b, sizes, offsets, rank);

    return (fakeNormR / fakeNormB) < (EPSILON * EPSILON);
}

void ConjugateGradientMethod(double* Ap, double* b, double* x, int* sizes,
int* offsets, int rank) {
    double* tmp1 = (double*)malloc(SIZE * sizeof(double));
    double* tmp2 = (double*)malloc(SIZE * sizeof(double));
    double* r = InitVectorR(Ap, b, x, tmp1, sizes, offsets, rank);
    double dotR = 0;

    double* z = (double*)malloc(SIZE * sizeof(double));
    CopyMatrix(r, z, sizes[rank], offsets[rank]);

    MPI_Allgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, z, sizes, offsets,
MPI_DOUBLE, MPI_COMM_WORLD);

    double alpha = 0.0;
    double beta = 0.0;

    int count = 0;

    while (!IsSolutionReached(b, r, sizes, offsets, rank) && (count <
50000)) {
        CalcNextAlpha(&alpha, Ap, r, z, &dotR, sizes, offsets, rank, tmp1);
        CalcNextXp(x, z, sizes, offsets, alpha, rank, tmp2);
        CalcNextRp(r, z, alpha, sizes, offsets, rank, tmp1, tmp2);
        CalcNextBeta(&beta, r, &dotR, sizes, offsets, rank);
        CalcNextZp(z, r, beta, sizes, offsets, rank);
        MPI_Allgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, z, sizes,
offsets, MPI_DOUBLE, MPI_COMM_WORLD);
```

```
            ++count;
        }

        MPI_Allgatherv(MPI_IN_PLACE, 0, MPI_DATATYPE_NULL, x, sizes, offsets,
MPI_DOUBLE, MPI_COMM_WORLD);

        free(tmp1);
        free(tmp2);
}

void FreeMemory(double* A, double* b, double* x, double* Ap, int* sizes,
int* offsets) {
        free(A);
        free(b);
        free(x);
        free(Ap);
        free(sizes);
        free(offsets);
}

void PrintResult(float totalTime, int numProc) {
        float boost = T1 / totalTime;
        float efficiency = (boost / (float)numProc) * 100;
        printf("Number of processes: %d\n", numProc);
        printf("Total time: %f sec\n", totalTime);
        printf("Sp = %f\n", boost);
        printf("Ep = %f\n\n", efficiency);
}

int main(int argc, char** argv) {
        int numProc, rank;
        MPI_Status status;

        MPI_Init(&argc, &argv);

        double startTime = MPI_Wtime();

        MPI_Comm_size(MPI_COMM_WORLD, &numProc);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        int* sizes = CalcSizesOfAp(numProc, SIZE, SIZE);
        int* offsets = CalcOffsetsOfAp(sizes, numProc);

        double* A = (double*)malloc(SIZE * SIZE * sizeof(double));
        double* b = (double*)malloc(SIZE * sizeof(double));
        double* x = (double*)malloc(SIZE * sizeof(double));
        InitPreSolution(x, SIZE);

        if (rank == 0) {
                InitMatrixA(A, SIZE);
                InitVectorB(b, SIZE);
        }

        double* Ap = (double*)malloc(sizes[rank] * sizeof(double));
```

```
    MPI_Scatterv(A, sizes, offsets, MPI_DOUBLE, Ap, sizes[rank], MPI_DOUBLE,
0, MPI_COMM_WORLD);
    MPI_Bcast(b, SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    for (int i = 0; i < numProc; ++i) {
        sizes[i] /= SIZE;
        offsets[i] /= SIZE;
    }

    ConjugateGradientMethod(Ap, b, x, sizes, offsets, rank);

    double endTime = MPI_Wtime();


    if (rank == 0) {
        PrintResult(endTime - startTime, numProc);
    }

    FreeMemory(A, b, x, Ap, sizes, offsets);
    MPI_Finalize();

    return 0;
}
```

Приложение 3. Результаты работы программы

```
hpcuser60@clu:~/lab1> cat run.sh.o5405842
The result of SEQUENTIAL program:
Number of iterations: 1804
Time: 106.000000 sec

The results of PARALLEL programs:
Number of processes: 2
Total time: 27.037199 sec
Sp = 3.920524
Ep = 196.026215

Number of processes: 4
Total time: 14.851889 sec
Sp = 7.137139
Ep = 178.428482

Number of processes: 8
Total time: 10.180353 sec
Sp = 10.412212
Ep = 130.152649

Number of processes: 16
Total time: 7.744564 sec
Sp = 13.687019
Ep = 85.543869

Number of processes: 24
Total time: 7.253199 sec
Sp = 14.614242
Ep = 60.892670

hpcuser60@clu:~/lab1>
```

Приложение 4. Скрипт run.sh

```
run.sh              [----] 16 L:[  1+ 2   3/ 21] *(29  / 566b) 0010 0x00A
#!/bin/bash

#PBS -q S5318391
#PBS -l walltime=00:20:00
#PBS -l select=2:ncpus=12:mpiprocs=12:mem=4000m,place=free
#PBS -m n

cd $PBS_O_WORKDIR

MPI_NP=$(wc -l $PBS_NODEFILE | awk '{ print $1 }')

echo "The result of SEQUENTIAL program:"
./default

echo "The results of PARALLEL programs:"

mpirun -machinefile $PBS_NODEFILE -np 2 ./parallel
mpirun -machinefile $PBS_NODEFILE -np 4 ./parallel
mpirun -machinefile $PBS_NODEFILE -np 8 ./parallel
mpirun -machinefile $PBS_NODEFILE -np 16 ./parallel
mpirun -trace -machinefile $PBS_NODEFILE -np 24 -perhost 12 ./parallel
```