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

«Два вектора»

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

1. Знакомство со стандартом MPI.
2. Сравнение времени выполнения расчета числа в двойном цикле путем последовательного и параллельного выполнения.

ЗАДАНИЕ

1. Написать 3 программы, каждая из которых рассчитывает число s по двум данным векторам a и b равной длины N в соответствии со следующим двойным циклом:

```
for (i = 0; i < N; i++)
        for(j = 0; j < N; j++)
                s += a[i] * b[j];
```

   a) последовательная программа

   b) параллельная, использующая коммуникации типа точка-точка (MPI_Send, MPI_Recv)

   c) параллельная, использующая коллективные коммуникации (MPI_Scatter, MPI_Reduce, MPI_Bcast)

2. Замерить время работы последовательной программы и параллельных на 2, 4, 8, 16, 24 процессах. Рекомендуется провести несколько замеров для каждого варианта запуска и выбрать минимальное время.

3. Построить графики времени, ускорения и эффективности.

4. Составить отчет, содержащий исходные коды разработанных программ и построенные графики.

ОПИСАНИЕ РАБОТЫ

Для чистоты эксперимента все программы были запущены на кафедральном сервере.

Последовательная программа (см. Приложение 1)

Функция `int* InitArray(int size)`. Реализует создание и инициализацию массива.

```
int* InitArray(int size) {
    int* arr = malloc(size * sizeof(int));
    for (int i = 0; i < size; ++i) {
        arr[i] = i;
    }
    return arr;
}
```

Функция `long long CalcS(int* a, int* b, int size)`. Реализует алгоритм расчета числа s из задания

```
long long CalcS(int* a, int* b, int size) {
    long long s = 0;
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            s += a[i] * b[j];
        }
    }
    return s;
}
```

Функция `int main()`. В ней поставлен таймер, определяющий, за какое время программа вычисляет число s. Для этих целей был выбран таймер системного времени time(time_t* Time). Размер массива SIZE равен 100 000. Такое значение позволяет вычислять число s больше 30 секунд.

```
int main() {
    int* array1 = InitArray(SIZE);
    int* array2 = InitArray(SIZE);

    time_t start, end;
    time(&start);
    long long s = CalcS(array1, array2, SIZE);
    time(&end);

    printf("%lld\n", s);
    printf("Time: %f sec\n", difftime(end, start));

    free(array1);
    free(array2);

    return 0;
}
```

Результат работы программы:

```
opp@comrade:~/204/osipov/lab0/default$ gcc main.c -o main
opp@comrade:~/204/osipov/lab0/default$ ./main
S = 2348225233701550336
Time: 38.000000 sec
opp@comrade:~/204/osipov/lab0/default$
```

Параллельная программа, использующая коммуникацию точка-точка (см. Приложение 2)

Для реализации параллельного выполнения вектор a (в коде array1) был разделен между процессами. Таким образом, он делится на n частей, где n – это число запущенных MPI-процессов. Заметим, что размер массива может делится на n с остатком. Чтобы не потерять данные, мы увеличиваем размер массива, чтобы он был равномерно поделен между процессами путем добавления фиктивных элементов, равных нулю.

Функция `int CalcCurrentArraySize(int numProc)`. Вычисляет оптимальный размер массива. Если его размер делится на число процессов без остатка, то она возвращает изначальный размер. В противном случае увеличивает размер.

```
int CalcCurrentArraySize(int numProc) {
	if (SIZE % numProc == 0) {
		return SIZE;
	}
	else {
		return SIZE + numProc - (SIZE % numProc);
	}
}
```

Функция `void InitArray(int* arr, int curArraySize, int numProc)`. Инициализирует необходимые элементы массива их индексом, а фиктивные элементы, получившиеся из-за увеличения массива, нулями.

```
void InitArray(int* arr, int curArraySize, int numProc) {
	for (int i = 0; i < curArraySize; ++i) {
		arr[i] = (i < SIZE ? i : 0);
	}
}
```

Функция `long long Mult(int* array1, int sizeArray1, int* array2, int sizeArray2)`. Выполняет «умножение» двух массивов согласно алгоритму из задания.

```
long long Mult(int* array1, int sizeArray1, int* array2, int sizeArray2) {
	long long result = 0;
	for (int i = 0; i < sizeArray1; ++i) {
		for (int j = 0; j < sizeArray2; ++j) {
			result += array1[i] * array2[j];
		}
	}
	return result;
}
```

Для реализации параллельного вычисления нулевой процесс (rank == 0) инициализирует два массива array1 и array2 и далее каждому процессу отправляет часть массива array1 размером bufferSize = curArraySize / numProc и весь массив array2.

```
if (rank == 0) {
	InitArray(array1, curArraySize, numProc);
	InitArray(array2, curArraySize, numProc);

	double startTime, endTime;
	startTime = MPI_Wtime();

	int shift = bufferSize;
	for (int i = 1; i < numProc; ++i) {
		MPI_Send(array1 + shift, bufferSize, MPI_INT, i, ID, MPI_COMM_WORLD);
		MPI_Send(array2, SIZE, MPI_INT, i, ID, MPI_COMM_WORLD);
		shift += bufferSize;
	}
	…
```

В это время оставшиеся процессы (rank != 0) ждут от нулевого процесса свою часть массива array1 и весь array2. Получив их, они вычисляют «свое» значение s и отправляют его нулевому процессу.

```
else {
	MPI_Recv(array1, bufferSize, MPI_INT, 0, ID, MPI_COMM_WORLD, &status);
	MPI_Recv(array2, SIZE, MPI_INT, 0, ID, MPI_COMM_WORLD, &status);
	s = Mult(array1, bufferSize, array2, SIZE);
	MPI_Send(&s, 1, MPI_LONG_LONG, 0, ID, MPI_COMM_WORLD);
}
```

Нулевой процесс, получив от каждого процесса «свое» значение s, суммирует их. Получившееся значение и будет искомым значением s

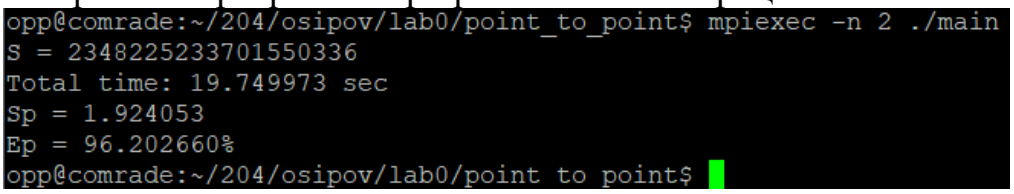
```
	…
	s = Mult(array1, bufferSize, array2, SIZE);
	long long stmp = 0;
	for (int i = 1; i < numProc; ++i) {
		MPI_Recv(&stmp, 1, MPI_LONG_LONG, MPI_ANY_SOURCE, ID, MPI_COMM_WORLD,
&status);
		s += stmp;
	}
	endTime = MPI_Wtime();
```

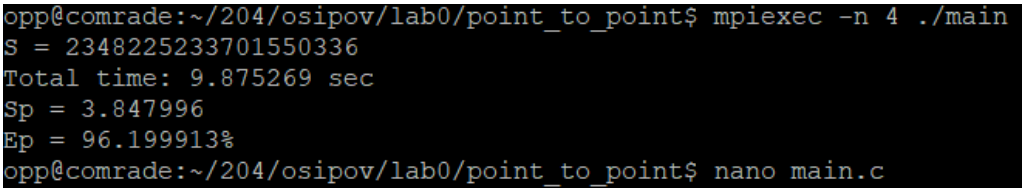
Для подсчета времени был выбран таймер системного времени MPI_Wtime().

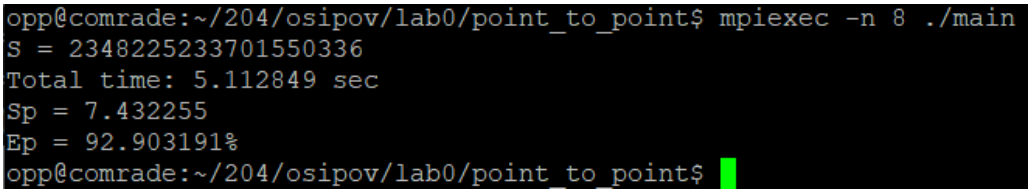
Функция `void PrintResult(long long* s, float totalTime, int numProc)`. Выводит результат работы программы: число s, время вычисления, ускорение (Sp) и эффективность (Ep).

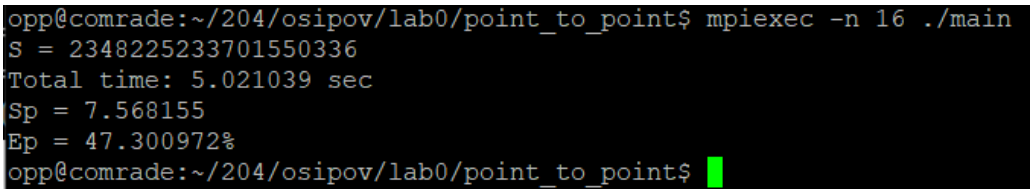
```
void PrintResult(long long* s, float totalTime, int numProc) {
	float boost = T1 / totalTime;
	float efficiency = (boost / (float)numProc) * 100;
	printf("S = %lld\n", *s);
	printf("Total time: %f\n", totalTime);
	printf("Sp = %f\n", boost);
	printf("Ep = %f%\n", efficiency);
}
```

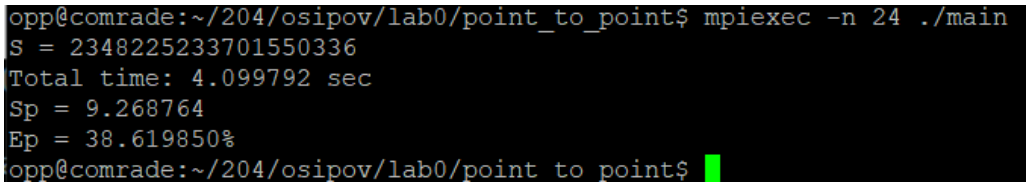
Результаты работы программы при разных числах процессов:











Параллельная программа, использующая
коллективную коммуникацию (см. Приложение 3)

Для данной реализации нулевой процесс сразу всем процессам отправляют часть массива array1 и так же отправляет целый массив array2 всем процессам (в отличие от предыдущей реализации, нулевой процесс тоже получает свою часть array1). Далее каждый процесс вычисляет свое значение stmp, и затем с помощью функции MPI_Reduce(…) все процессы отправляют stmp нулевому процессу, суммируя получившиеся значения. Это и есть искомое s.

```
…
MPI_Scatter(array1, bufferSize, MPI_INT, recvBuffer, bufferSize, MPI_INT,
0, MPI_COMM_WORLD);
MPI_Bcast(array2, SIZE, MPI_INT, 0, MPI_COMM_WORLD);
stmp = Mult(recvBuffer, bufferSize, array2, SIZE);
MPI_Reduce(&stmp, &s, 1, MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
…
```

Результат работы программы при разных числах процессов:

```
opp@comrade:~/204/osipov/lab0/collective$ mpiexec -n 2 ./main
S = 2348225233701550336
Total time: 19.722391 sec
Sp = 1.926744
Ep = 96.337204%
opp@comrade:~/204/osipov/lab0/collective$
```

```
opp@comrade:~/204/osipov/lab0/collective$ mpiexec -n 4 ./main
S = 2348225233701550336
Total time: 9.866122 sec
Sp = 3.851564
Ep = 96.289101%
opp@comrade:~/204/osipov/lab0/collective$
```

```
opp@comrade:~/204/osipov/lab0/collective$ mpiexec -n 8 ./main
S = 2348225233701550336
Total time: 5.107888 sec
Sp = 7.439475
Ep = 92.993431%
opp@comrade:~/204/osipov/lab0/collective$
```

```
opp@comrade:~/204/osipov/lab0/collective$ mpiexec -n 16 ./main
S = 2348225233701550336
Total time: 4.658295 sec
Sp = 8.157492
Ep = 50.984322%
opp@comrade:~/204/osipov/lab0/collective$
```

```
opp@comrade:~/204/osipov/lab0/collective$ mpiexec -n 24 ./main
S = 2348225233701550336
Total time: 3.952261 sec
Sp = 9.614749
Ep = 40.061455%
opp@comrade:~/204/osipov/lab0/collective$
```

График времени

Y1 – коммуникация точка-точка, сек.
Y2 – коллективная коммуникация, сек.

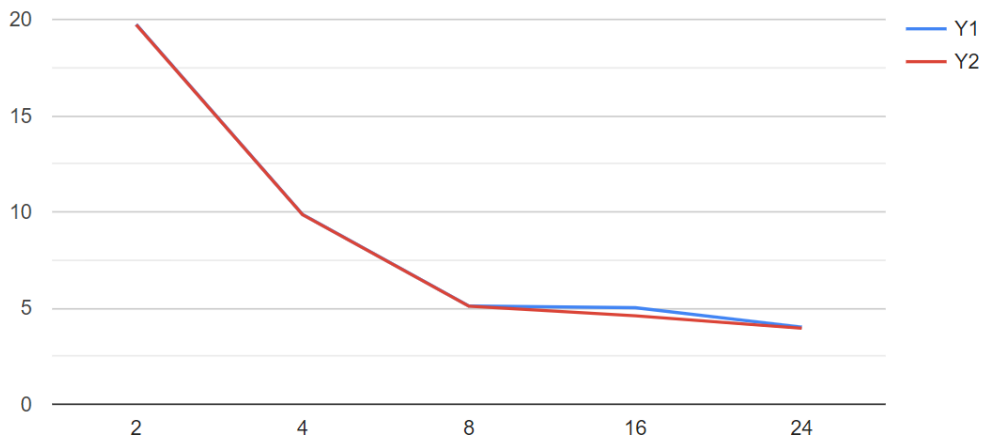


График ускорения

Y1 – коммуникация точка-точка, ускорение.
Y2 – коллективная коммуникация, ускорение.

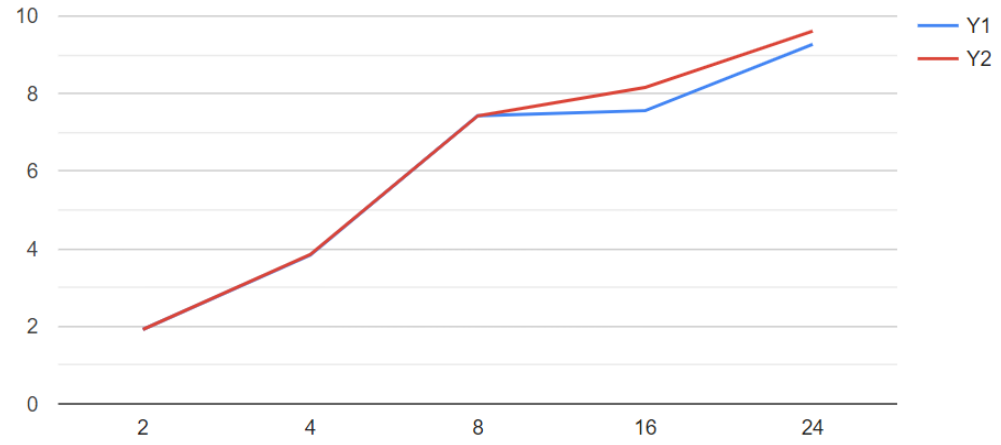
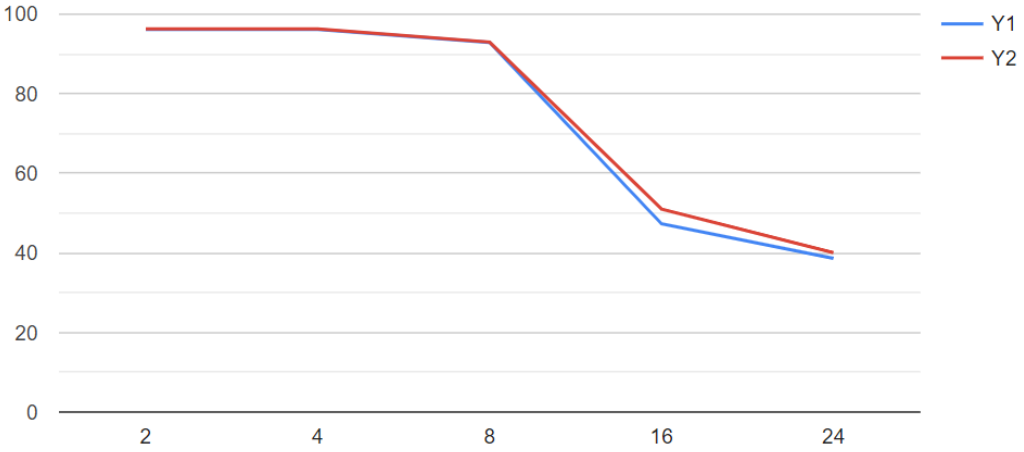


График эффективности

Y1 – коммуникация точка-точка, эффективность (%).
Y2 – коллективная коммуникация, эффективность (%).



ЗАКЛЮЧЕНИЕ

В рамках данной практической работы были изучен стандарт MPI и функции для реализации параллельного вычисления. Были написаны три программы: последовательная и две параллельные с разными коммуникациями (точка-точка и коллективная). Практическая работа показала, что параллельные программы, запущенные на нескольких процессах, работают гораздо быстрее последовательного аналога. Ускорение и эффективность зависят от числа запущенных MPI процессов. Было установлено, что параллельные программы, использующих коммуникацию точка-точка и коллективную соответственно, при равном количестве процессов показывают схожий результат. Из этого можно сделать вывод, что функции для коллективной коммуникации являются оберткой над функциями коммуникации точка-точка, предназначенные для более короткого и удобного написания параллельных программ.

Приложение 1. Листинг последовательной программы

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 100000

int* InitArray(int size) {
    int* arr = malloc(size * sizeof(int));
    for (int i = 0; i < size; ++i) {
        arr[i] = i;
    }
    return arr;
}

long long CalcS(int* array1, int* array2, int size) {
    long long s = 0;
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            s += array1[i] * array2[j];
        }
    }
    return s;
}

int main() {
    int* array1 = InitArray(SIZE);
    int* array2 = InitArray(SIZE);

    time_t startTime, endTime;
    time(&startTime);
    long long s = CalcS(array1, array2, SIZE);
    time(&endTime);

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

    free(array1);
    free(array2);

    return 0;

}
```

Приложение 2. Листинг параллельной программы с коммуникацией точка-точка

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"
#define SIZE 100000
#define ID 1
#define T1 38.0

int CalcCurrentArraySize(int numProc) {
      if (SIZE % numProc == 0) {
            return SIZE;
      }
      else {
            return SIZE + numProc - (SIZE % numProc);
      }
}
```

```
void InitArray(int* arr, int curArraySize, int numProc) {
	for (int i = 0; i < curArraySize; ++i) {
		arr[i] = (i < SIZE ? i : 0);
	}
}

long long Mult(int* array1, int sizeArray1, int* array2, int sizeArray2) {
	long long result = 0;
	for (int i = 0; i < sizeArray1; ++i) {
		for (int j = 0; j < sizeArray2; ++j) {
			result += array1[i] * array2[j];
		}
	}
	return result;
}

void PrintResult(long long* s, float totalTime, int numProc) {
	float boost = T1 / totalTime;
	float efficiency = (boost / (float)numProc) * 100;
	printf("S = %lld\n", *s);
	printf("Total time: %f\n", totalTime);
	printf("Sp = %f\n", boost);
	printf("Ep = %f%\n", efficiency);
}

void FreeArrays(int* array1, int* array2) {
	free(array1);
	free(array2);
}

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

	const int curArraySize = CalcCurrentArraySize(numProc);
	int* array1 = malloc(curArraySize * sizeof(int));
	int* array2 = malloc(curArraySize * sizeof(int));

	long long s = 0;
	const int bufferSize = curArraySize / numProc;

	if (rank == 0) {
		InitArray(array1, curArraySize, numProc);
		InitArray(array2, curArraySize, numProc);

		double startTime, endTime;
		startTime = MPI_Wtime();

		int shift = bufferSize;
		for (int i = 1; i < numProc; ++i) {
			MPI_Send(array1 + shift, bufferSize, MPI_INT, i, ID,
MPI_COMM_WORLD);
			MPI_Send(array2, SIZE, MPI_INT, i, ID, MPI_COMM_WORLD);
			shift += bufferSize;
		}

		s = Mult(array1, bufferSize, array2, SIZE);
		long long stmp = 0;
		for (int i = 1; i < numProc; ++i) {
			MPI_Recv(&stmp, 1, MPI_LONG_LONG, MPI_ANY_SOURCE, ID,
MPI_COMM_WORLD, &status);
```

```
                        s += stmp;
                }
                endTime = MPI_Wtime();

                PrintResult(&s, endTime - startTime, numProc);
        }
        else {
                MPI_Recv(array1, bufferSize, MPI_INT, 0, ID, MPI_COMM_WORLD, &status);
                MPI_Recv(array2, SIZE, MPI_INT, 0, ID, MPI_COMM_WORLD, &status);
                s = Mult(array1, bufferSize, array2, SIZE);
                MPI_Send(&s, 1, MPI_LONG_LONG, 0, ID, MPI_COMM_WORLD);
        }
        FreeArrays(array1, array2);
        MPI_Finalize();
        return 0;
}
```

Приложение 3. Листинг параллельной программы с коллективной коммуникацией

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"
#define SIZE 100000
#define ID 1
#define T1 38.0

int CalcCurrentArraySize(int numProc) {
        if (SIZE % (numProc) == 0)
                return SIZE;
        else
                return SIZE + (numProc)-(SIZE % (numProc));
}

void InitArray(int* arr, int curArraySize) {
        for (int i = 0; i < curArraySize; ++i) {
                arr[i] = (i < SIZE ? i : 0);
        }
}

long long Mult(int* a, int sizeA, int* b, int sizeB) {
        long long result = 0;
        for (int i = 0; i < sizeA; ++i) {
                for (int j = 0; j < sizeB; ++j) {
                        result += a[i] * b[j];
                }
        }
        return result;
}

void PrintResult(long long* s, float totalTime, int numProc) {
        float boost = T1 / totalTime;
        float efficiency = (boost / (float)numProc) * 100;
        printf("S = %lld\n", *s);
        printf("Total time: %f sec\n", totalTime);
        printf("Sp = %f\n", boost);
        printf("Ep = %f%\n", efficiency);
}

void FreeArrays(int* array1, int* array2, int* recvBuffer) {
        free(array1);
```

```
	free(array2);
	free(recvBuffer);
}

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

	const int curArraySize = CalcCurrentArraySize(numProc);
	int* array1 = malloc(curArraySize * sizeof(int));
	int* array2 = malloc(curArraySize * sizeof(int));

	const int bufferSize = curArraySize / (numProc);
	int* recvBuffer = malloc(bufferSize * sizeof(int));

	double timeStart, timeEnd;
	if (rank == 0) {
		InitArray(array1, curArraySize);
		InitArray(array2, curArraySize);
		timeStart = MPI_Wtime();
	}
	long long s = 0, stmp = 0;

	MPI_Scatter(array1, bufferSize, MPI_INT, recvBuffer, bufferSize, MPI_INT, 0,
MPI_COMM_WORLD);
	MPI_Bcast(array2, SIZE, MPI_INT, 0, MPI_COMM_WORLD);
	stmp = Mult(recvBuffer, bufferSize, array2, SIZE);
	MPI_Reduce(&stmp, &s, 1, MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);

	if (rank == 0) {
		timeEnd = MPI_Wtime();
		PrintResult(&s, timeEnd - timeStart, numProc);
	}
	FreeArrays(array1, array2, recvBuffer);
	MPI_Finalize();

	return 0;
}
```