Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**ЛАБОРАТОРНАЯ РАБОТА №4**

**Дисциплина: Распределенные задачи и алгоритмы**

Работу выполнила: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д. Н. Баева

Направление подготовки: 02.03.02 Фундаментальная информатика и информационные технологии

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. И. Шиян

**Тема работы:** Двухточечные обмены.

**Ход работы:**

В задании лабораторной работы дана двумерная матрица размера NxN и вектор размерности N. Необходимо было написать параллельную MPI-программу вычисления скалярного произведения матрицы и вектора (представлен одномерным массивом) используя двухточечный блокирующий, синхронный и коллективный обмены. Программа должна быть организована по схеме master-slave. Полученные результаты необходимо сравнить, используя засечение времени с помощью класса Stopwatch из пакета com.google.common.base и оформить в виде графиков зависимости времени выполнения от числа процессов и ускорения от числа процессов.

Каждый способ обмена представляет собой отдельную программу, написанную на языке программирования Java.

Первым был реализован двухточечный блокирующий способ обмена. Двухточечный обмен — это один из методов передачи данных между двумя процессами. Он заключается в том, что каждый процесс отправляет данные другому процессу, а затем принимает данные от него. Обмен блокирующий, то есть процесс останавливается и ждет, пока данные не будут получены.

Для реализации двухточечного обмена в данном коде используются следующие методы:

- MPI.Init(args) - инициализация MPI;

- MPI.COMM\_WORLD.Rank() - определение ранга процесса;

- MPI.COMM\_WORLD.Size() - определение размера коммуникатора;

- MPI.COMM\_WORLD.Send() - отправка данных;

- MPI.COMM\_WORLD.Recv() - получение данных.

Названные методы работают по следующей логике: сперва инициализируется MPI и определяются ранг и размер коммуникатора. Если ранг равен 0, то определяется матрица и вектор, заполняются случайными значениями и отправляются в slaves. Вычисляется локальное скалярное произведение и получается из slaves. Аналогичный порядок методов и действия и в случае, если ненулевой ранг. Фрагмент кода, выполняющий это, представлен на рисунке 1.

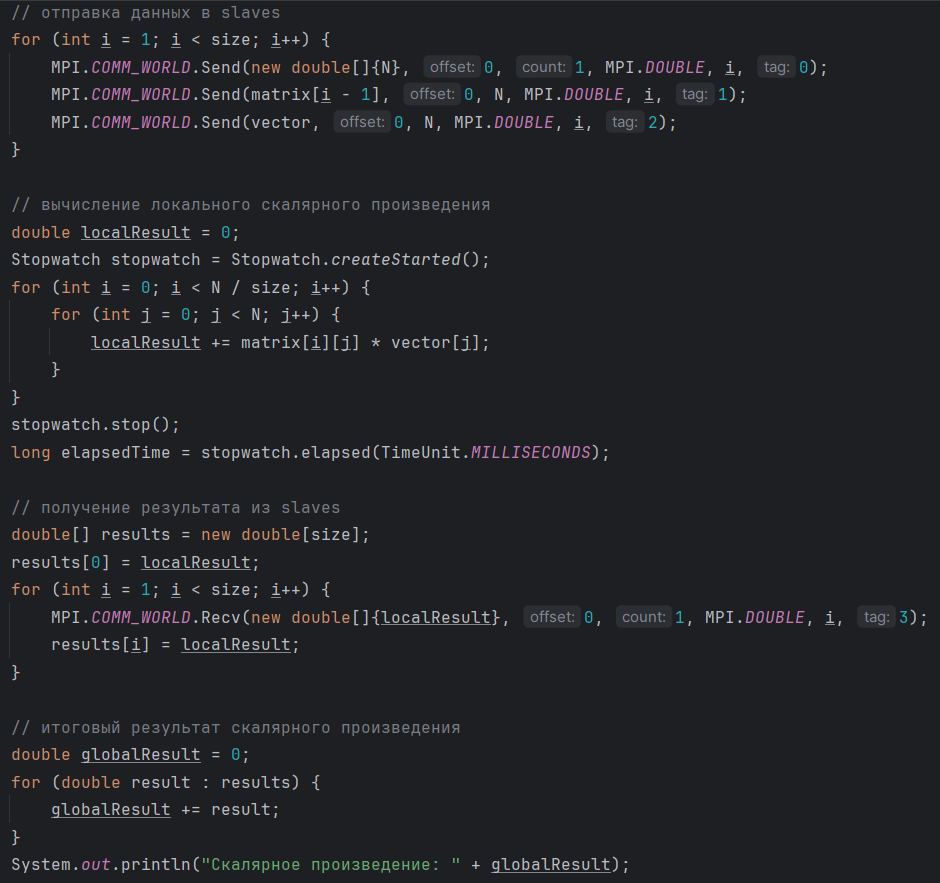


Рисунок 1 – Отправка и получение данных в двухточечном блокирующем обмене.

Для удобства определения зависимости времени выполнения программы от количества процессов результаты для разных случаев были представлены в виде таблиц (рисунок 2).

Рисунок 2 – Результаты работы метода на разном количестве процессов и количестве элементов для метода MPI.COMM\_WORLD.Send().

Следующим был реализован двухточечный синхронный обмен. Отличие от предыдущего состоит в том, что так как обмен синхронный, то это значит, что процесс не останавливается и продолжает работу, пока данные не будут получены. Для этого используется MPI.COMM\_WORLD.Irsend() — это функция MPI, которая используется для отправки данных в режиме готовности. Она является синхронной, но не блокирующей, то есть процесс не останавливается и продолжает работу, пока данные не будут получены. Фрагмент кода этого обмена представлен на рисунке 3.



Рисунок 3 – Отправка и получение данных в двухточечном синхронном режиме.

Аналогично предыдущему случаю, результаты работы представлены в виде таблицы. Она отображена на рисунке 4.

Рисунок 4 – Результаты работы метода на разном количестве процессов и количестве элементов для метода MPI.COMM\_WORLD.Irsend().

Финальным был реализован метод обмена данных в коллективном режиме. Набор операций типа точка-точка является достаточным для программирования любых алгоритмов, однако одной из наиболее привлекательных сторон MPI является наличие широкого набора коллективных операций, которые берут на себя выполнение наиболее часто встречающихся при программировании действий. Например, часто возникает потребность разослать некоторую переменную или массив из одного процессора всем остальным. Каждый программист может написать такую процедуру с использованием операций Send/Recv, однако гораздо удобнее воспользоваться коллективными операциями. Причем гарантировано, что эта операция будет выполняться гораздо эффективнее, поскольку MPI-функция реализована с использованием внутренних возможностей коммуникационной среды.

Главное отличие коллективных операций от операций типа точка-точка состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором. Несоблюдение этого правила приводит либо к аварийному завершению задачи, либо к еще более неприятному зависанию задачи.

Для реализации коллективного обмена использовались следующие методы:

- MPI.Init(args) - инициализация MPI;

- MPI.COMM\_WORLD.Rank() - определение ранга процесса;

- MPI.COMM\_WORLD.Size() - определение размера коммуникатора;

- MPI.COMM\_WORLD.Barrier() - блокировка выполнения программы до тех пор, пока все процессы не достигнут этой точки;

- MPI.COMM\_WORLD.Scatter() - рассылка данных на все процессы;

- MPI.COMM\_WORLD.Gather() - сбор данных на корневом процессе.

В коде происходит следующее: сперва аналогично предыдущим случая инициализируется MPI и определяются ранг и размер коммуникатора, далее определяется матрица и вектор, заполняются случайными значениями. После матрица переопределяется в массив и вычисляется локальное произведение матрицы и вектора. Результаты собираются на корневом процессе. Фрагмент кода, выполняющий это, представлен на рисунке 5.

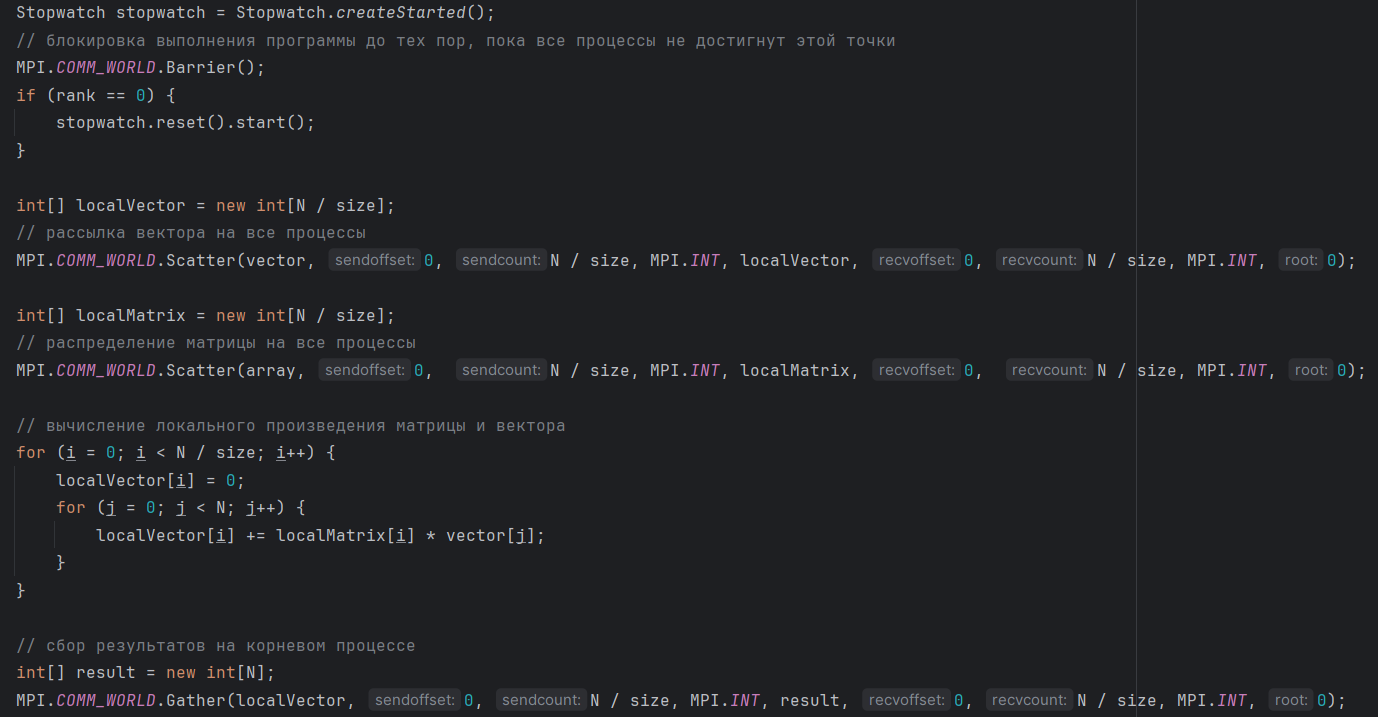


Рисунок 5 – Обмен данными в коллективном режиме.

Результаты работы представлены в виде таблицы. Она отображена на рисунке 6.



Рисунок 6 – Результаты работы метода на разном количестве процессов и количестве элементов для методов коллективного обмена.

На основе всех получившихся значений были построены графики зависимости времени выполнения от числа процессов и ускорения от числа процессов (рисунок 7).

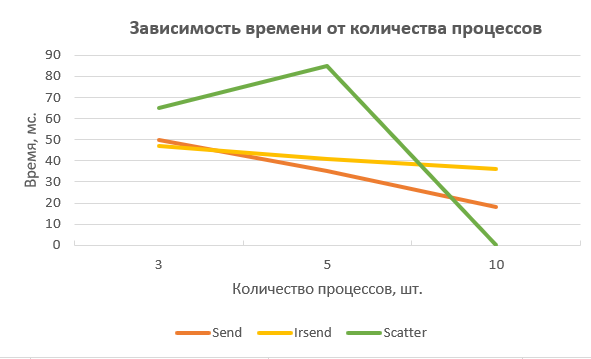


Рисунок 7 – Графики зависимости времени от количества процессов.

**Листинг программ**

Файл LW\_4\_1.java

package LW\_4;  
  
import com.google.common.base.Stopwatch;  
import mpi.\*;  
  
import java.util.Random;  
import java.util.concurrent.TimeUnit;  
  
// двухточечный обмен, блокирующий  
public class LW\_4\_1 {  
 public static void main(String[] args) throws MPIException {  
 MPI.*Init*(args);  
 int rank = MPI.*COMM\_WORLD*.Rank();  
 int size = MPI.*COMM\_WORLD*.Size();  
  
 if (rank == 0) {  
 // определение матрицы и вектора  
 int N = 100;  
 double[][] matrix = new double[N][N];  
 double[] vector = new double[N];  
 Random rand = new Random();  
 // заполнение матрицы и вектора случайными значениями  
 for (int i = 0; i < N; i++) {  
 for (int j = 0; j < N; j++) {  
 matrix[i][j] = rand.nextDouble(100);  
 }  
 vector[i] = rand.nextDouble(100);  
 }  
  
 // отправка данных в slaves  
 for (int i = 1; i < size; i++) {  
 MPI.*COMM\_WORLD*.Send(new double[]{N}, 0, 1, MPI.*DOUBLE*, i, 0);  
 MPI.*COMM\_WORLD*.Send(matrix[i - 1], 0, N, MPI.*DOUBLE*, i, 1);  
 MPI.*COMM\_WORLD*.Send(vector, 0, N, MPI.*DOUBLE*, i, 2);  
 }  
  
 // вычисление локального скалярного произведения  
 double localResult = 0;  
 Stopwatch stopwatch = Stopwatch.*createStarted*();  
 for (int i = 0; i < N / size; i++) {  
 for (int j = 0; j < N; j++) {  
 localResult += matrix[i][j] \* vector[j];  
 }  
 }  
 stopwatch.stop();  
 long elapsedTime = stopwatch.elapsed(TimeUnit.*MILLISECONDS*);  
  
 // получение результата из slaves  
 double[] results = new double[size];  
 results[0] = localResult;  
 for (int i = 1; i < size; i++) {  
 MPI.*COMM\_WORLD*.Recv(new double[]{localResult}, 0, 1, MPI.*DOUBLE*, i, 3);  
 results[i] = localResult;  
 }  
  
 // итоговый результат скалярного произведения  
 double globalResult = 0;  
 for (double result : results) {  
 globalResult += result;  
 }  
 System.*out*.println("Скалярное произведение: " + globalResult);  
 System.*out*.println("Затраченное время: " + elapsedTime + " milliseconds");  
 } else {  
 // получение данных из master  
 double[] data = new double[1001];  
 MPI.*COMM\_WORLD*.Recv(data, 0, data.length, MPI.*DOUBLE*, 0, MPI.*ANY\_TAG*);  
  
 // вычисление локального скалярного произведения  
 int N = (int) data[0];  
 double[] matrixRow = new double[N];  
 double[] vector = new double[N];  
 MPI.*COMM\_WORLD*.Recv(matrixRow, 0, N, MPI.*DOUBLE*, 0, MPI.*ANY\_TAG*);  
 MPI.*COMM\_WORLD*.Recv(vector, 0, N, MPI.*DOUBLE*, 0, MPI.*ANY\_TAG*);  
 double localResult = 0;  
 for (int i = 0; i < N; i++) {  
 localResult += matrixRow[i] \* vector[i];  
 }  
  
 // отправка результатов в master  
 MPI.*COMM\_WORLD*.Send(new double[]{localResult}, 0, 1, MPI.*DOUBLE*, 0, 3);  
 }  
 // освобождение ресурсов  
 MPI.*Finalize*();  
 }  
}

LW\_4\_2.java

package LW\_4;  
  
import com.google.common.base.Stopwatch;  
import mpi.\*;  
  
import java.util.Random;  
import java.util.concurrent.TimeUnit;  
  
// двухточечный обмен, синхронный, не блокирующий  
public class LW\_4\_2 {  
 public static void main(String[] args) throws MPIException {  
 MPI.*Init*(args);  
 int rank = MPI.*COMM\_WORLD*.Rank();  
 int size = MPI.*COMM\_WORLD*.Size();  
  
 if (rank == 0) {  
 // определение матрицы и вектора  
 int N = 10000;  
 double[][] matrix = new double[N][N];  
 double[] vector = new double[N];  
 Random rand = new Random();  
 // заполнение матрицы и вектора случайными значениями  
 int i, j;  
 for (i = 0; i < N; i++) {  
 for (j = 0; j < N; j++) {  
 matrix[i][j] = rand.nextDouble(100);  
 }  
 vector[i] = rand.nextDouble(100);  
 }  
  
 // отправка данных в slaves  
 for (i = 1; i < size; i++) {  
 MPI.*COMM\_WORLD*.Irsend(new double[]{N}, 0, 1, MPI.*DOUBLE*, i, 0);  
 MPI.*COMM\_WORLD*.Irsend(matrix[i - 1], 0, N, MPI.*DOUBLE*, i, 1);  
 MPI.*COMM\_WORLD*.Irsend(vector, 0, N, MPI.*DOUBLE*, i, 2);  
 }  
  
 // вычисление локального скалярного произведения  
 double localResult = 0;  
 Stopwatch stopwatch = Stopwatch.*createStarted*();  
 for (i = 0; i < N / size; i++) {  
 for (j = 0; j < N; j++) {  
 localResult += matrix[i][j] \* vector[j];  
 }  
 }  
 stopwatch.stop();  
 long elapsedTime = stopwatch.elapsed(TimeUnit.*MILLISECONDS*);  
  
 // получение результата из slaves  
 double[] results = new double[size];  
 results[0] = localResult;  
 for (i = 1; i < size; i++) {  
 MPI.*COMM\_WORLD*.Recv(new double[]{localResult}, 0, 1, MPI.*DOUBLE*, i, 3);  
 results[i] = localResult;  
 }  
  
 // итоговый результат глобального скалярного произведения  
 double globalResult = 0;  
 for (double result : results) {  
 globalResult += result;  
 }  
 System.*out*.println("Скалярное произведение: " + globalResult);  
 System.*out*.println("Затраченное время: " + elapsedTime + " milliseconds");  
 } else {  
 // получение данных из master  
 double[] data = new double[1001];  
 MPI.*COMM\_WORLD*.Recv(data, 0, data.length, MPI.*DOUBLE*, 0, MPI.*ANY\_TAG*);  
  
 // вычисление локального скалярного произведения  
 int N = (int) data[0];  
 double[] matrixRow = new double[N];  
 double[] vector = new double[N];  
 MPI.*COMM\_WORLD*.Recv(matrixRow, 0, N, MPI.*DOUBLE*, 0, MPI.*ANY\_TAG*);  
 MPI.*COMM\_WORLD*.Recv(vector, 0, N, MPI.*DOUBLE*, 0, MPI.*ANY\_TAG*);  
 double localResult = 0;  
 for (int i = 0; i < N; i++) {  
 localResult += matrixRow[i] \* vector[i];  
 }  
  
 // отправка в master  
 MPI.*COMM\_WORLD*.Irsend(new double[]{localResult}, 0, 1, MPI.*DOUBLE*, 0, 3);  
 }  
 // освобождение ресурсов  
 MPI.*Finalize*();  
 }  
}

Файл LW\_4\_3.java

package LW\_4;  
  
import com.google.common.base.Stopwatch;  
import mpi.\*;  
  
import java.util.Random;  
import java.util.concurrent.TimeUnit;  
  
// двухточечный обмен, синхронный, не блокирующий  
public class LW\_4\_2 {  
 public static void main(String[] args) throws MPIException {  
 MPI.*Init*(args);  
 int rank = MPI.*COMM\_WORLD*.Rank();  
 int size = MPI.*COMM\_WORLD*.Size();  
  
 if (rank == 0) {  
 // определение матрицы и вектора  
 int N = 10000;  
 double[][] matrix = new double[N][N];  
 double[] vector = new double[N];  
 Random rand = new Random();  
 // заполнение матрицы и вектора случайными значениями  
 int i, j;  
 for (i = 0; i < N; i++) {  
 for (j = 0; j < N; j++) {  
 matrix[i][j] = rand.nextDouble(100);  
 }  
 vector[i] = rand.nextDouble(100);  
 }  
  
 // отправка данных в slaves  
 for (i = 1; i < size; i++) {  
 MPI.*COMM\_WORLD*.Irsend(new double[]{N}, 0, 1, MPI.*DOUBLE*, i, 0);  
 MPI.*COMM\_WORLD*.Irsend(matrix[i - 1], 0, N, MPI.*DOUBLE*, i, 1);  
 MPI.*COMM\_WORLD*.Irsend(vector, 0, N, MPI.*DOUBLE*, i, 2);  
 }  
  
 // вычисление локального скалярного произведения  
 double localResult = 0;  
 Stopwatch stopwatch = Stopwatch.*createStarted*();  
 for (i = 0; i < N / size; i++) {  
 for (j = 0; j < N; j++) {  
 localResult += matrix[i][j] \* vector[j];  
 }  
 }  
 stopwatch.stop();  
 long elapsedTime = stopwatch.elapsed(TimeUnit.*MILLISECONDS*);  
  
 // получение результата из slaves  
 double[] results = new double[size];  
 results[0] = localResult;  
 for (i = 1; i < size; i++) {  
 MPI.*COMM\_WORLD*.Recv(new double[]{localResult}, 0, 1, MPI.*DOUBLE*, i, 3);  
 results[i] = localResult;  
 }  
  
 // итоговый результат глобального скалярного произведения  
 double globalResult = 0;  
 for (double result : results) {  
 globalResult += result;  
 }  
 System.*out*.println("Скалярное произведение: " + globalResult);  
 System.*out*.println("Затраченное время: " + elapsedTime + " milliseconds");  
 } else {  
 // получение данных из master  
 double[] data = new double[1001];  
 MPI.*COMM\_WORLD*.Recv(data, 0, data.length, MPI.*DOUBLE*, 0, MPI.*ANY\_TAG*);  
  
 // вычисление локального скалярного произведения  
 int N = (int) data[0];  
 double[] matrixRow = new double[N];  
 double[] vector = new double[N];  
 MPI.*COMM\_WORLD*.Recv(matrixRow, 0, N, MPI.*DOUBLE*, 0, MPI.*ANY\_TAG*);  
 MPI.*COMM\_WORLD*.Recv(vector, 0, N, MPI.*DOUBLE*, 0, MPI.*ANY\_TAG*);  
 double localResult = 0;  
 for (int i = 0; i < N; i++) {  
 localResult += matrixRow[i] \* vector[i];  
 }  
  
 // отправка в master  
 MPI.*COMM\_WORLD*.Irsend(new double[]{localResult}, 0, 1, MPI.*DOUBLE*, 0, 3);  
 }  
 // освобождение ресурсов  
 MPI.*Finalize*();  
 }  
}