Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

Дисциплина «Параллельные вычисления»

«Разработка программ с использованием threads, MPI»

Работу выполнили студенты группы №13541/2 Чеботарев Г.М._____

Работу принял преподаватель Стручков И. В._____

Санкт-Петербург

2017

Программа работы:

Согласно выданному варианту (умножение матриц), необходимо выполнить следующие:

1. Разработать программу с последовательным алгоритмом
2. Выполнить распараллеливание алгоритма с помощью класса threads
3. Выполнить распараллеливание алгоритма с помощью библиотеки MPI

В выводах провести сравнительный анализ всех трех подходов.

Ход работы:

Введение

При компиляции, код написанный на Java преобразовывается в байт-код, который в свою очередь исполняется. Виртуальная машина java способна оптимизировать байт-код. В случае запуска небольшого объема кода, уже после нескольких запусков ВМ упростит байт-код, как следствие алгоритм (в данном случае умножение матриц) начнет выполняться быстрее. Т.е. выполниться «прогрев» виртуальной машины. С целью повышения точности исследования, тестирования алгоритмов были использованы с помощью Benchmark (библиотека jmh).

Точное кол-во запусков для «прогрева» ВМ угадать сложно. Т.к. объем кода небольшой, кол-во запусков для прогрева ВМ тоже небольшое. Вполне достаточно отвести 5 запусков на прогрев.

Прогрев выполняется для каждой из размерностей матриц (т.е. после каждого изменения размерности матрицы – будет выполнен прогрев). После прогрева, выполняются 5 идентичных экспериментов. С одинаковой размерностью. На основе пяти результатов, рассчитывается среднее время выполнения алгоритма и вычисляется погрешность. Затем размерность увеличивается, вновь выполняется прогрев, вновь выполняется замер средней скорости работы. Итоговый результат приводится в виде таблицы.

Этап 1. Разработка алгоритма с последовательным исполнением.

Код первого алгоритма умножения приведен в листинге 1.

Листинг 1. Последовательный алгоритм умножения.

```
public int[][] simple_multiplication_v1(int a[][], int b[][],int c[][]){
    for(int i = 0; i < a.length; ++i)
        for(int j = 0; j < a.length; ++j){
            c[i][j] = 0;
            for(int k = 0; k < a.length; ++k)
                c[i][j] = a[i][k]*b[k][j];
        }
    return c;
}
```

Результат измерений:

```
# Run complete. Total time: 00:29:10

Benchmark     (SIZE)  Mode  Cnt             Score             Error  Units
Main.checkIt       1  avgt    5             9,098 ±           1,468  ns/op
Main.checkIt      10  avgt    5          1479,253 ±          50,910  ns/op
Main.checkIt     100  avgt    5       1579051,941 ±      158426,488  ns/op
Main.checkIt     500  avgt    5     395707887,823 ±   371634950,451  ns/op
Main.checkIt    1000  avgt    5    7181196011,400 ±  1735507036,309  ns/op
Main.checkIt    2000  avgt    5  141432394832,600 ± 22340853190,710  ns/op
```

Рис. 1. Результат алгоритма 1 последовательного умножения.

Общее время выполнения тестов (с прогревом) составило 29 минут 10 секунд. Алгоритм имеет кубическую сложность, как следствие увеличение размерности матриц ведет к резкому ухудшению производительности.

Этап 2. Разработка алгоритма с использованием класса threads.

Алгоритм заключается в распределении умножения столбцов матриц по разным потокам (Код приведен в приложении 2). Создается N потоков, каждый из которых занимается вычислением назначенных ему ячеек. Результаты приведены ниже:

```
# Run complete. Total time: 00:18:40

Benchmark     (SIZE)  Mode  Cnt             Score             Error  Units
Main.checkIt       1  avgt    5       408046,118 ±       30076,579  ns/op
Main.checkIt      10  avgt    5       408594,978 ±       33807,161  ns/op
Main.checkIt     100  avgt    5      1309388,400 ±     1072725,542  ns/op
Main.checkIt     500  avgt    5    204358184,633 ±    75001570,133  ns/op
Main.checkIt    1000  avgt    5   5484885320,500 ±   649004872,136  ns/op
Main.checkIt    2000  avgt    5  99860742596,600 ± 4575735625,176  ns/op
```

Рис. 5. Результат алгоритма 1 последовательного умножения.

Без дополнительной оптимизации умножение матриц выполняется примерно столько же, сколько и при последовательном исполнении. Возможно это связано с тем, что JM эффективно использует все ядра в машине, даже при исполнении последовательной программы.

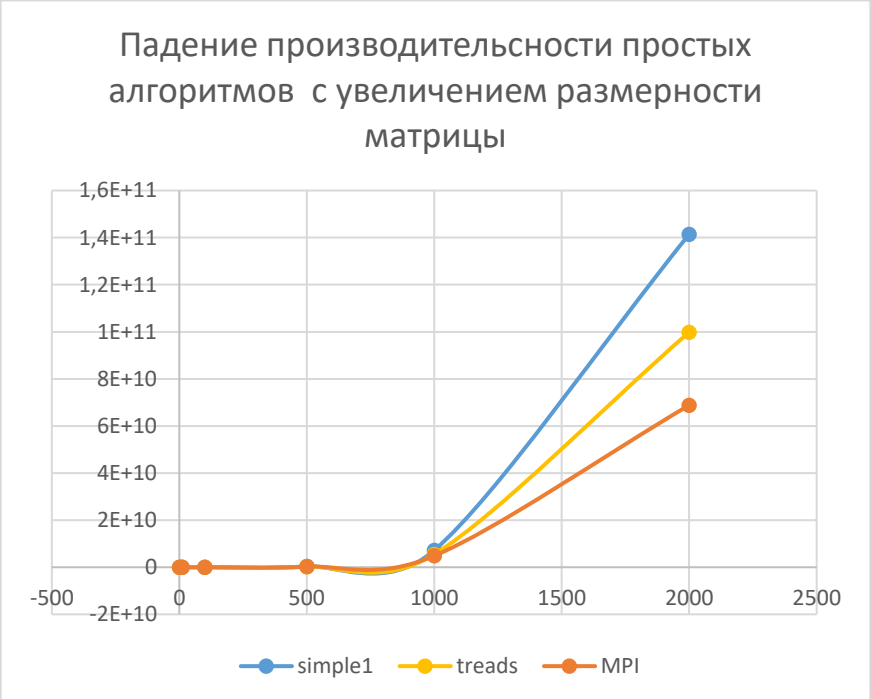
Этап 3. Разработка алгоритма с использованием MPJ (MPI).

Как и в предыдущем пункте, умножение матрицы происходит в многопоточном режиме. Компьютер имеет 2 физических и 2 виртуальных ядра. Вычисления распределяются между ядрами. Одно из ядер является главным, и распределяем данных между остальными. После оно принимает их результат и собирает результирующую матрицу. К сожалению при использовании библиотеки MPI пришлось отказаться от JMH. В результате измерения проводились без разогрева (просто несколько запусков, вычисление среднего значения времени исполнения). Результат приведен ниже:

Count	Time
1	10 msec
10	18 msec
100	30 msec
1000	4845 msec
2000	68833 ms

Код приведен в приложении 3.

Итоги работы:



На итоговом графике показаны результаты для всех трех способов умножения матриц. Как и ожидалось, MPI быстрее Threads, Threads быстрее последовательного способа исполнения программы. Такой результат был ожидаем, т.к. вычисления стали происходить более эффективно. Временные затраты на распределение данных по потокам снижались, эффективность использования ядер росла. Однако, «сверх-прироста» не наблюдается. MPI всего в 2,2 раза лучше последовательного исполнения. Очевидно это связано с прослойкой в виде JM.

Приложение 1. Последовательное исполнение программы (JMH).

```
package multmatrix;

import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Fork;
import org.openjdk.jmh.annotations.Measurement;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.OutputTimeUnit;
import org.openjdk.jmh.annotations.Param;
import org.openjdk.jmh.annotations.Scope;
import org.openjdk.jmh.annotations.Setup;
import org.openjdk.jmh.annotations.State;
import org.openjdk.jmh.annotations.Warmup;
import org.openjdk.jmh.infra.Blackhole;
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.RunnerException;
import org.openjdk.jmh.runner.options.Options;
import org.openjdk.jmh.runner.options.OptionsBuilder;

import java.util.Random;
import java.util.concurrent.TimeUnit;

@State(Scope.Thread)
@Warmup(iterations = 10, time = 1, timeUnit = TimeUnit.SECONDS)
@Measurement(iterations = 10, time = 1, timeUnit = TimeUnit.SECONDS)
@Fork(1)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class Main {
    private static final Random r =  new Random();

    // so simple
    public int[][] simple_multiplication_v1(int a[][], int b[][],int c[][]){
        for(int i = 0; i < a.length; ++i)
            for(int j = 0; j < a.length; ++j){
                c[i][j] = 0;
                for(int k = 0; k < a.length; ++k)
                    c[i][j] = a[i][k]*b[k][j];
            }
        return c;
    }

    // multiplication with transpose
    public int[][] simple_multiplication_v2(int a[][], int b[][],int c[][]){
        double bt[][] = new double[a.length][a.length];
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a.length; j++) {
                bt[j][i] = b[i][j];
            }
        }

        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < a.length; j++) {
                int summand = 0;
                for (int k = 0; k < a.length; k++) {
                    summand += a[i][k] * bt[j][k];
                }
                c[i][j] = summand;
            }
        }
        return c;
    }

    // combination cycles
    public int[][] simple_multiplication_v3(int a[][], int b[][],int c[][]){
        double thatColumn[] = new double[a.length];
        for (int j = 0; j < a.length; j++) {
```

```
            for (int k = 0; k < a.length; k++) {
                thatColumn[k] = B[k][j];
            }

            for (int i = 0; i < a.length; i++) {
                int[] thisRow = A[i];
                double summand = 0;
                for (int k = 0; k < a.length; k++) {
                    summand += thisRow[k] * thatColumn[k];
                }
                c[i][j] = (int) summand;
            }
        }
        return c;
    }

    // multiply Matrix
    public int[][] multiplyMatrixMT(int a[][], int b[][],int c[][]) {
        multiplyMatrixMT m = new multiplyMatrixMT();
        c = m.multiplyMatrixMT(a, b, Runtime.getRuntime().availableProcessors());
        return c;
    }


    @Param({"1", "10", "100", "500", "1000", "2000"})
    private static int SIZE;

    int[][] A, B, C;

    public static void swap(int[] a, int i, int j) {
        int x = a[i];
        a[i] = a[j];
        a[j] = x;
    }

    @Setup
    public void setup() {
        A = new int[SIZE][SIZE];
        B = new int[SIZE][SIZE];
        C = new int[SIZE][SIZE];

        A = fillMatrix(A);
        B = fillMatrix(B);
    }

    @Benchmark
    public void checkIt(Blackhole bh) {
        bh.consume(multiplyMatrixMT(A,B,C));
    }

    /* Fill matrix*/
    private static int[][] fillMatrix(int A[][]){
        for (int i=0;i < A.length;i++){
            for (int j=0;j < A[i].length;j++){
                A[i][j]=(int)(Math.random()*10);
            }
        }
        return A;
    }

    /* Print matrix */
    private static void printMatrix(int A[][]){
        for (int i=0;i < A.length;i++,System.out.println()){
            for (int j=0;j < A[i].length;j++){
                System.out.print(A[i][j]+" ");
            }
        }
        System.out.println("");
    }
```

```
    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(Main.class.getSimpleName())
                .warmupIterations(5)
                .measurementIterations(5)
                .forks(1)
                .build();

        new Runner(opt).run();
    }

}
```

Приложение 2. Реализация с использованием threads.

```
package multmatrix;

/**
 * Created by Георгий on 02.04.2017.
 */

class MultiplierThread extends Thread
{
    private final int[][] A;
    private final int[][] B;
    private final int[][] C;

    private final int firstIndex;
    private final int lastIndex;
    private final int sumLength;


    public MultiplierThread(final int[][] A, final int[][] B, final int[][] C,
                            final int firstIndex, final int lastIndex){
        this.A  = A;
        this.B = B;
        this.C = C;
        this.firstIndex   = firstIndex;
        this.lastIndex    = lastIndex;

        sumLength = B.length;
    }

    /**
     * calculation value in one cell
     * */
    private void calcValue(final int row, final int col)
    {
        int sum = 0;
        for (int i = 0; i < sumLength; ++i)
            sum += A[row][i] * B[i][col];
        C[row][col] = sum;
    }


    @Override
    public void run()
    {
        final int colCount = B[0].length;  // count row in result matrix
        for (int index = firstIndex; index < lastIndex; ++index)
            calcValue(index / colCount, index % colCount);
    }
}

public class multiplyMatrixMT
{
    public static int[][] multiplyMatrixMT(final int[][] A, final int[][] B, int threadCount)
    {
        assert threadCount > 0;

        final int rowCount = A.length;
        final int colCount = B[0].length;
        final int[][] result = new int[rowCount][colCount];

        final int cellsForThread = (rowCount * colCount) / threadCount;
        int firstIndex = 0;
        final MultiplierThread[] multiplierThreads = new MultiplierThread[threadCount];


        for (int threadIndex = threadCount - 1; threadIndex >= 0; --threadIndex) {
            int lastIndex = firstIndex + cellsForThread;  // index of the last of the cell
            if (threadIndex == 0) {
                // остаток
                lastIndex = rowCount * colCount;
            }
            multiplierThreads[threadIndex] = new MultiplierThread(A, B, result, firstIndex,
lastIndex);
            multiplierThreads[threadIndex].start();
            firstIndex = lastIndex;
        }
```

```
        // Ожидание завершения потоков.
        try {
            for (final MultiplierThread multiplierThread : multiplierThreads)
                multiplierThread.join();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }

        return result;
    }
}
```

Приложение 3. Реализация с использованием MPI.

```
import mpi.*;           // for mpiJava
import java.net.*;      // for InetAddress
import java.util.*;     // for Date

public class Main {

    private int myrank = 0;
    private int nprocs = 0;

    // matrices
    private double A[], B[], C[];

    // messages
    int averows;                // average #rows allocated to each rank
    int extra;                  // extra #rows allocated to some ranks
    int offset[] = new int[1]; // offset in row
    int rows[] = new int[1];   // the actual # rows allocated to each rank
    int mtype;                  // message type (tagFromMaster or tagFromWorker )

    final static int tagFromMaster = 1;
    final static int tagFromWorker = 2;
    final static int master = 0;

    // print option
    boolean isPrint = false;


    /**
     *  Matrix initialization
     */
    private void init( int size ) {
        // Initialize matrices

        for ( int i = 0; i < size; i++ )
            for ( int j = 0; j < size; j++ )
                A[i * size + j] = (int)(Math.random()*10);

        for ( int i = 0; i < size; i++ )
            for ( int j = 0; j < size; j++ )
                B[i * size + j] = (int)(Math.random()*10);
    }

    /**
     * Computes a multiplication for rows
     */
    private void compute( int size ) {

        for ( int k = 0; k < size; k++ )
            for ( int i = 0; i <= rows[0]; i++ )
                for ( int j = 0; j < size; j++ )
                    C[i * size + k] += A[i * size + j] * B[j *size + k];

    }

    /**
     * Printing matrices
     */
    private void print( double array[] ) {
        if ( myrank == 0 && isPrint == true ) {
            int size = ( int )Math.sqrt( ( double )array.length );
            for ( int i = 0; i < size; i++ ){
                for ( int j = 0; j < size; j++ ) {
                    System.out.print( array[i * size + j] + " ");
                }
                System.out.println();
            }
            System.out.println();
```

```
        }
    }


    public Main( int size, boolean option ) throws MPIException {
        myrank = MPI.COMM_WORLD.Rank( );
        nprocs = MPI.COMM_WORLD.Size( );

        A = new double[size * size];
        B = new double[size * size];
        C = new double[size * size];

        isPrint = option;

        if ( myrank == 0 ) {
            // Initialize matrices.
            init( size );
            System.out.println( "array a:" );
            print( A );
            System.out.println( "array b:" );
            print( B );

            // Construct message components.
            averows = size / nprocs;
            extra = size % nprocs;
            offset[0] = 0;
            mtype = tagFromMaster;

            // Start timer.
            Date startTime = new Date( );

            // Send matrices to each worker.
            for ( int rank = 0; rank < nprocs; rank++ ) {
                if(rank < extra){
                    rows[0] = averows + 1;
                }else{
                    rows[0] = averows;
                }

                System.out.println( "sending " + rows[0] + " rows to rank " +
                        rank );
                if ( rank != 0 ) {
                    MPI.COMM_WORLD.Send( offset, 0, 1, MPI.INT, rank, mtype );
                    MPI.COMM_WORLD.Send( rows, 0, 1, MPI.INT, rank, mtype );
                    MPI.COMM_WORLD.Send( A, offset[0] * size, rows[0] * size,
                            MPI.DOUBLE, rank, mtype );
                    MPI.COMM_WORLD.Send( B, 0, size * size, MPI.DOUBLE, rank,
                            mtype );
                }
                offset[0] += rows[0];
            }

            // Perform matrix multiplication.
            compute( size );

            // Collect results from each worker.
            int mytpe = tagFromWorker;
            for ( int source = 1; source < nprocs; source++ ) {
                MPI.COMM_WORLD.Recv( offset, 0, 1, MPI.INT, source, mtype );
                MPI.COMM_WORLD.Recv( rows, 0, 1, MPI.INT, source, mtype );
                MPI.COMM_WORLD.Recv( C, offset[0] * size, rows[0] * size,
                        MPI.DOUBLE, source, mtype );
            }

            // Stop timer.
            Date endTime = new Date( );

            // Print out results
            System.out.println( "result c:" );
            print( C );
```

```
            System.out.println( "time elapsed = " +
                    ( endTime.getTime( ) - startTime.getTime( ) ) +
                    " msec" );
        }
        else {
            // I'm a worker.

            // Receive matrices.
            int mtype = tagFromMaster;
            MPI.COMM_WORLD.Recv( offset, 0, 1, MPI.INT, master, mtype );
            MPI.COMM_WORLD.Recv( rows, 0, 1, MPI.INT, master, mtype );
            MPI.COMM_WORLD.Recv( A, 0, rows[0] * size, MPI.DOUBLE, master,
                    mtype );
            MPI.COMM_WORLD.Recv( B, 0, size * size, MPI.DOUBLE, master,
                    mtype );

            // Perform matrix multiplication.
            compute( size );

            // Send results to the master.
            MPI.COMM_WORLD.Send( offset, 0, 1, MPI.INT, master, mtype );
            MPI.COMM_WORLD.Send( rows, 0, 1, MPI.INT, master, mtype );
            MPI.COMM_WORLD.Send( C, 0, rows[0] * size, MPI.DOUBLE, master,
                    mtype );
        }

        try {
            // Print out a complication message.
            InetAddress inetaddr = InetAddress.getLocalHost( );
            String ipname = inetaddr.getHostName( );
            System.out.println( "rank[" + myrank + "] at " + ipname +
                    ": multiplication completed" );
        } catch ( UnknownHostException e ) {
            System.err.println( e );
        }
    }

    /**
     *
     * @param args Receive the matrix size and the print option in args[0] and
     *             args[1]
     */
    public static void main( String[] args ) {
        // Check # args.

        // Start the MPI library.
        MPI.Init( args );

        // Will initialize size[0] with args[1] and option with args[2] (y | n)
        int size[] = new int[1];
        size[0] = 500;
        boolean option[] = new boolean[1];
        option[0] =  false;

        // Broadcast size and option to all workers.
        MPI.COMM_WORLD.Bcast( size, 0, 1, MPI.INT, master );
        MPI.COMM_WORLD.Bcast( option, 0, 1, MPI.BOOLEAN, master );

        // Compute matrix multiplication in both master and workers.
        new Main( size[0], option[0] );

        // Terminate the MPI library.
        MPI.Finalize( );
    }
}
```