Assignment 6
Using the Seeds Pattern Programming Framework - Workpool Pattern

B. Wilkinson and C. Ferner: Modification date: October 27, 2014
(corrected in red October 31, 2014, see course home page announcements)

Preliminaries

The purpose of this assignment is to become familiar with the Java-based Seeds pattern programming framework. The Seeds framework will be used on your own computer (or a lab computer). The provided Ubuntu virtual machine is already loaded with the Seeds libraries and sample files, and Eclipse. Since Seeds only requires Java and will run on any platform, you can also do this assignment on a native Windows or Linux installation and upload the files for the course home page. The code will be executed using Eclipse with the Java perspective.

Seeds Framework. Various patterns are available in Seeds including workpool, pipeline and synchronous stencil, and others can be created. We shall use the workpool pattern in this assignment as it is very general and applicable to many problems. In the workpool pattern, a master node sends out tasks to slave workers. The slaves perform computations and return results to the master, which then produces the final results. To achieve dynamic load balancing, the master keeps a queue of tasks. When slave returns a result of task, it is given another task from the queue until all talks are completed.

To use the Seeds workpool, the programmer must implement a Java interface with three principal methods:

- The diffuse method – used by the master to distribute pieces of the data to the slaves.
- The compute method – used by the slaves for the actual computation
- The gather method – used by the master to gather the results

An additional "data count" method is required to tell the framework how many pieces of data will be computed. The programmer might implement a few other methods depending upon the application, notably an initialization method and a method to compute the final result. No message passing routines are needed by the programmer - the diffuse method creates a **DataMap** object with data to be passed to the slaves (as a **Data** object), and similarly the compute method creates a **DataMap** object with data to the passed back to the master (as a **Data** object). The framework takes care of the message passing and self deploys on a local computer, a cluster, or a geographically distributed Grid platform when the application is launched. Deployment is done using a "bootstrapping" class with a main method. This class is mostly written for each pattern and the programmer simply fills in site-specific details (computer names, paths, etc.) prior to running.

Seeds Framework Versions. There are three versions of the Java-based Seeds framework currently implemented:

- Full JXTA P2P networking version suitable for a fully distributed network of computers and requiring an Internet connection even if just running on a single computer
- A simplified JXTA P2P version called the "NoNetwork" version for testing on a single computer and not requiring an Internet connection, and finally
- Multicore version implemented with threads for more efficient execution on single multicore computer or shared memory multiprocessor system. It does not require an Internet connection.

The two JXTA versions can use the same module source code and bootstrap run the module code, and executes in the same fashion with similar logging output. The multicore version also uses the same module source code but the bootstrap run module code is slightly different (thread-based). Here we will use the multicore version. The Appendix gives details on using the P2P version.

Software

In this assignment, simple workpool applications (the Monte Carlo π calculation and matrix addition and matrix multiplication) will be executed using the Eclipse IDE. The provided virtual machine has the Seeds programs and libraries in **~/ParallelProg/Seeds/workspaceMulticore**.[1] If you are using your own native Windows/Linux installation, you will need to download the Seeds libraries and programs from the link "VM software" on the course home page and make sure you have the specified directory structure. (You will also have to include the Seeds libraries in the Eclipse project as will be described.)

The following projects and libraries are within the Seeds workspace:

Monte Carlo pi ("PiApprox"")
MatrixAddition ("MatrixAdd")
Matrix Multiplication ("MatrixMult")
Numerical Integration ("NumIntegration")
SeedsTemplate ("SeedsTemplate" for code development)
Seeds libraries ("seedsMulticore")

The directory structure and important files to know are given overleaf.

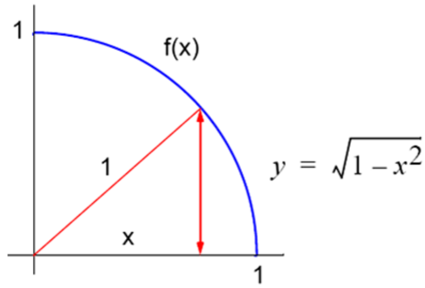
[1] **~/ParallelProg/Seeds/workspaceNetwork** hold the two JXTA P2P versions.

```
~/ParallelProg/Seeds/workspaceMulticore               // used to hold Seeds projects
    /seedsMulticore                                   // Seeds framework
        /lib                                          // Seeds libraries
        Availableservers.txt                          // holds computer names (not used here)
        …

    /PiApprox                                         // Monte Carlo pi project
        /bin/edu/uncc/grid/example/workpool           // Class files, empty until code compiled
        /src/edu/uncc/grid/example/workpool/          // Java source files
                MonteCarloPiModule.java
                RunMonteCarloPiModule.java

    /MatrixAdd                                        // Matrix add project
        /bin/edu/uncc/grid/example/workpool/          // Class files, empty until code compiled
        / src/edu/uncc/grid/example/workpool/         // Java source files
                MatrixAddModule.java
                RunMatrixAddModule.java

    /MatrixMult                                       // Matrix multiply project
        /bin/edu/uncc/grid/example/workpool/          // Class files, empty until code compiled
        / src/edu/uncc/grid/example/workpool/         // Java source files
                MatrixMultModule.java
                RunMatrixMultModule.java

    /NumericalIntegration                             // Numerical Integration project
        /bin/edu/uncc/grid/example/workpool/          // Class files, empty until code compiled
        / src/edu/uncc/grid/example/workpool/         // Java source files
                NumericalIntegrationModule.java
                RunNumericalIntegrationModule.java

    /SeedsTemplate                                    // For code development
        …
```

Software directory structure

Part 1 Monte Carlo π Code (25%)

The Monte Carlo algorithm for computing π is well known and given in many parallel programming texts. It is a so-called embarrassingly parallel application particularly amenable to parallel implementation *but used more for demonstration purposes than as a good way to compute* π. However, it can lead to more important Monte Carlo applications. A circle is formed within a square. The circle has unit radius and the square has sides 2 x 2. The ratio of the area of the circle to the area of the square is given by $\pi(1^2)/(2 \times 2) = \pi/4$. Points within the square are chosen randomly and a score is kept of how many points happen to lie within the circle. The fraction of points within the circle will be $\pi/4$, given a sufficient number of randomly selected points. Usually only one quadrant is used as shown below:



so that the randomly selected points generated, x_r, y_r, are each between 0 and 1, and are counted as in the circle, if $y_r \leq \sqrt{1 - x_r^2}$, i. e. $y_r^2 + x_r^2 \leq 1$. Note the integral $\int_0^1 \sqrt{1 - x^2}\, dx = \pi/4$ is being evaluated.

Implementing the pattern in the Seeds framework requires two Java classes - a module class called here **MonteCarloPiModule.java** implements a pattern interface and the bootstrapping class called here **RunMonteCarloPiModule.java**. The bootstrapping class is used to run the application. In the workpool here, the master process sends a different random number to each of the slaves. Each slave uses that number as the starting seed for their random number generator. The Java Random class nextDouble method returns a number uniformly distributed between 0 and 1.0.[2] Each slave then gets the next two random numbers as the coordinates of a point (x,y) using nextDouble. If the point is within the circle (i.e. $x^2 + y^2 <= 1$), it increments a counter that is counting the number of points within the circle. This is repeated for 1000 points. Each slave returns its accumulated count. The GatherData method performed by the master accumulates the slave results. A separate method, getPi, executed within the bootstrap module, computes the final approximation for π using the accumulated total.

MonteCarloPiModule.java. MonteCarloPiModule.java implements the interface for the workpool

```
package edu.uncc.grid.example.workpool;
import java.util.Random;
import java.util.logging.Level;
import edu.uncc.grid.pgaf.datamodules.Data;
import edu.uncc.grid.pgaf.datamodules.DataMap;
import edu.uncc.grid.pgaf.interfaces.basic.Workpool;
```

[2] Excluding 0 and 1 but we are ignoring that.

```java
import edu.uncc.grid.pgaf.p2p.Node;

public class MonteCarloPiModule extends Workpool {
    private static final long serialVersionUID = 1L;
    private static final int DoubleDataSize = 1000;
    double total;
    int random_samples;
    Random R;
    public MonteCarloPiModule() {
        R = new Random();
    }
    public void initializeModule(String[] args) {
        total = 0;
        Node.getLog().setLevel(Level.WARNING); // reduce verbosity for logging
        random_samples = 3000; // set number of random samples
    }
    public Data Compute (Data data) {
        DataMap<String, Object> input = (DataMap<String,Object>)data; // data from DiffuseData()
        DataMap<String, Object> output = new DataMap<String, Object>(); // output from Compute
        Long seed = (Long) input.get("seed"); // get random seed
        Random r = new Random();
        r.setSeed(seed);
        Long inside = 0L;
        for (int i = 0; i < DoubleDataSize ; i++) {
            double x = r.nextDouble();
            double y = r.nextDouble();
            double dist = x * x + y * y;
            if (dist <= 1.0) {
              ++inside;
            }
        }
        output.put("inside", inside);     // store partial answer to return to GatherData()
        return output;
    }
    public Data DiffuseData (int segment) {
        DataMap<String, Object> d =new DataMap<String, Object>();
        d.put("seed", R.nextLong());
        return d; // returns a random seed for each job unit
    }
    public void GatherData (int segment, Data dat) {
        DataMap<String,Object> out = (DataMap<String,Object>) dat;
        Long inside = (Long) out.get("inside");
        total += inside; // aggregate answer from all the worker nodes.
    }
    public double getPi() { // returns value of pi based on the job done by all the workers
        double pi = (total / (random_samples * DoubleDataSize)) * 4;
        return pi;
    }
    public int getDataCount() {
        return random_samples;
    }
}
```

MonteCarloPiModule.java

In **MonteCarloPiModule.java**, two important classes are imported called **Data** and **DataMap**. **Data** is used to pass data between the master and slaves. **DataMap** is used within DiffuseData, Compute, and GatherData methods for specifying the data being passed and uses two parameters, a string and an object (generic typing). The first parameter can be any programmer chosen string and used to identify the second stored item.[3] DiffuseData method (executed by the master) creates a DataMap object and returns it with random seed for each job. The Compute method (executed by slaves) picks up the data from DiffuseData and creates a **DataMap** object for holding its partial results.

RunMonteCarloPiModule.java. RunMonteCarloPiModule.java deploys the Seeds pattern and runs the workpool. Below is the code for the multicore version of the framework

```
package edu.uncc.grid.example.workpool;
import java.io.IOException;
import net.jxta.pipe.PipeID;
import edu.uncc.grid.pgaf.Anchor;
import edu.uncc.grid.pgaf.Operand;
import edu.uncc.grid.pgaf.Seeds;
import edu.uncc.grid.pgaf.p2p.Types;

public class RunMonteCarloPiModule {
    public static void main(String[ ] args) {
        try {
            long start = System.currentTimeMillis();

            MonteCarloPiModule pi = new MonteCarloPiModule();
            Thread id = Seeds.startPatternMulticore( new Operand( (String[])null,
                                new Anchor( args[0],Types.DataFlowRole.SINK_SOURCE), pi ), 4 );
            id.join();

            System.out.println( "The result is: " + pi.getPi() ) ;

            long stop = System.currentTimeMillis();
            double time = (double) (stop - start) / 1000.0;
            System.out.println("Execution time = " + time);

        } catch (SecurityException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

RunMonteCarloPiModule.java –Thread-based multicore version

Several classes are imported, PipeID, seeds-specific Anthor, Operand, Seeds, and Types. An instance of MonteCarloPiModule is first created. The Thread object is the thread managing

---

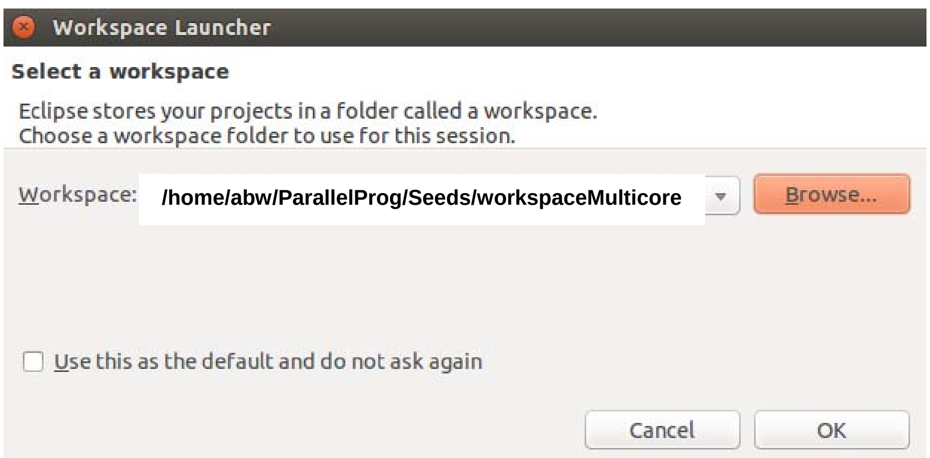
[3] **DataMap** extends Java **HashMap**.

the source and sink threads for the pattern. The programmer can monitor when the pattern is done computing by checking id.isAlive() or can just wait for the pattern to complete using id.join(). Args[0] should be the local host name.

The Seeds method startPattern starts the workpool pattern on the computers. It requires as a single argument an Operand object. Creating an Object object requires three arguments. The first is a String list of argument that will be submitted to the host. The second is an Anchor object specifying the nodes that should have source and sink nodes (the master in this case) which in this provided as the string argument of main (first command line argument, args[0]). The third argument is an instance of MonteCarloPiModule previously created. As mentioned above, to run this code, we will need to provide one command line argument, the name of the local host.

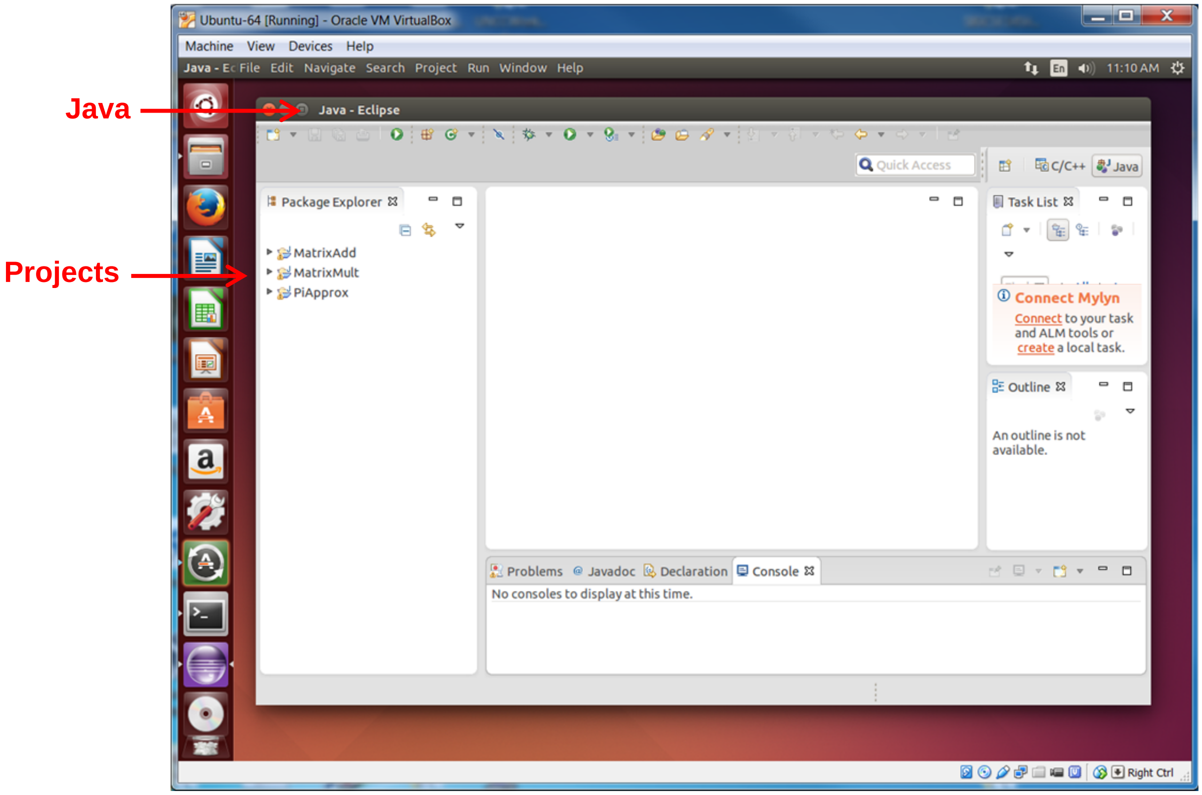
Task 1 Executing the Monte Carlo π program. The program can be executed on the command line or through an IDE. We choose to use Eclipse here.

Step 1. Open Eclipse Start Eclipse on the command line (**eclipse**). Browse and select the workspace ~/**ParallelProg/Seeds/workspaceMulticore**



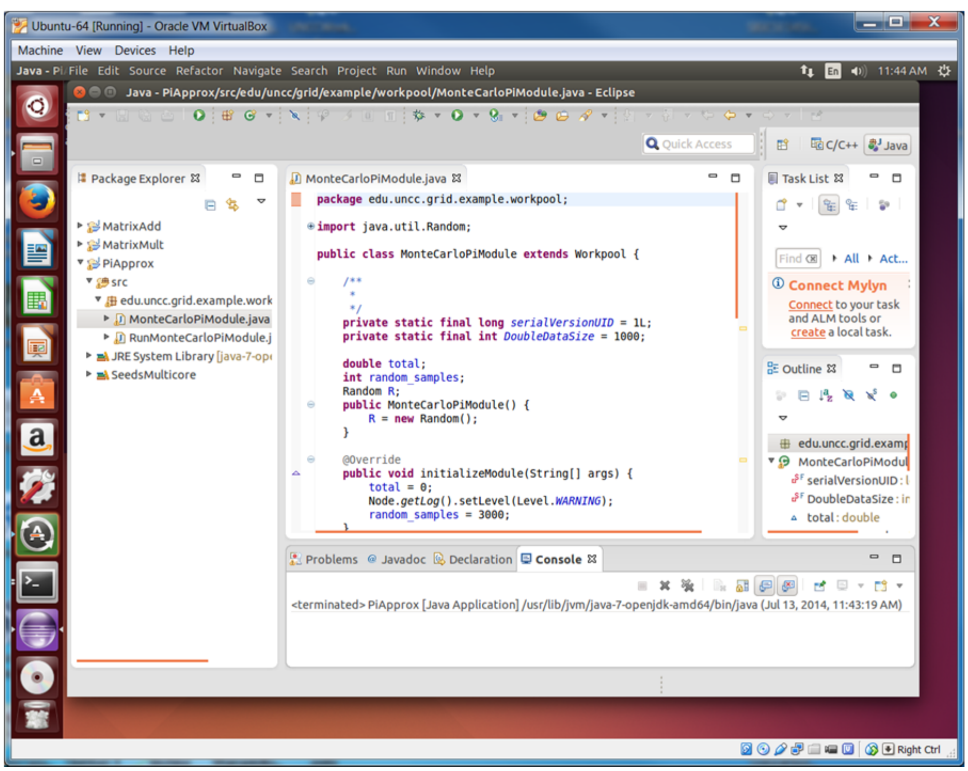
Eclipse might open in the C/C++ perspective rather than the Java perspective as its last use. If so, go to **Window** > **Open perspective** > **Other** > **Java** to open in the Java perspective

You should see the Monte Carlo π, MatrixAdd and MatrixMult projects already loaded:

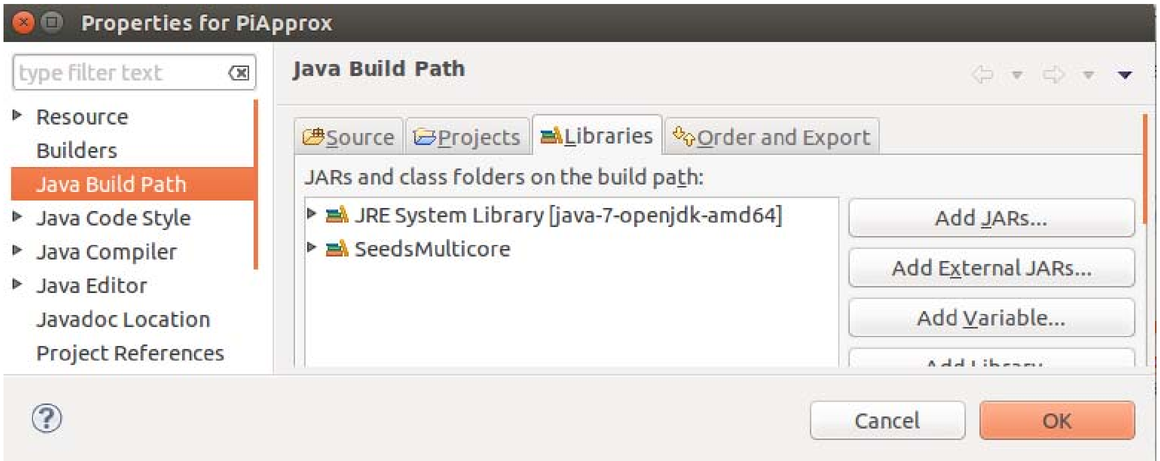


If you do not see the projects, import the projects with **File > Import > General > Existing Projects** into Workspace. You may also have to click on the Workspace" icon at the right.

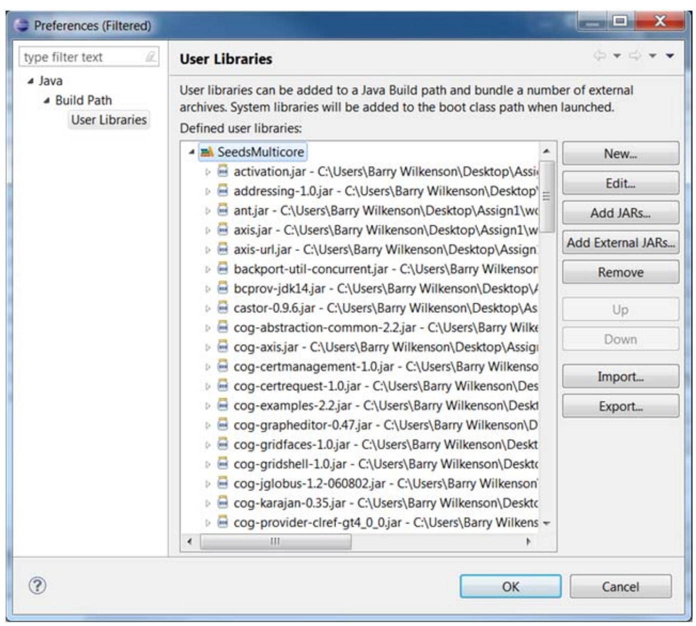
You may want to look at the **PiApprox** project we will be executing:



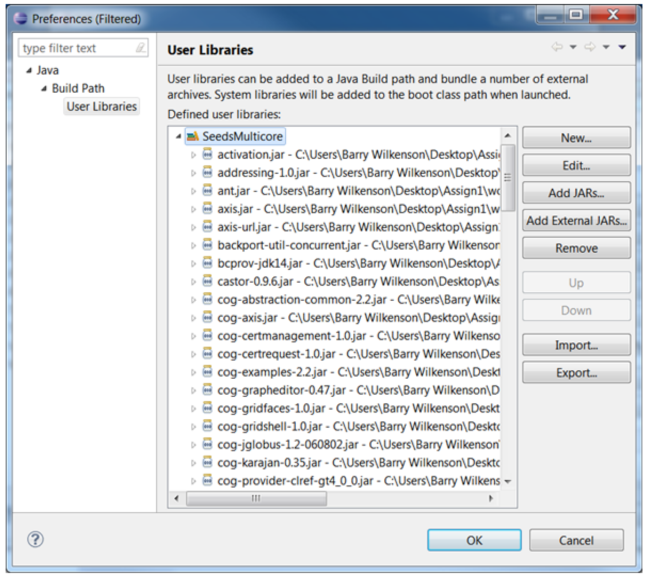
Step 2 Add Build path to Seeds Libraries. The build path to the Seeds library should already set up in the VM with a named user library, “**SeedsMulticore**”: -- but the paths may be incorrect so you may need to delete the paths and to re-create them.



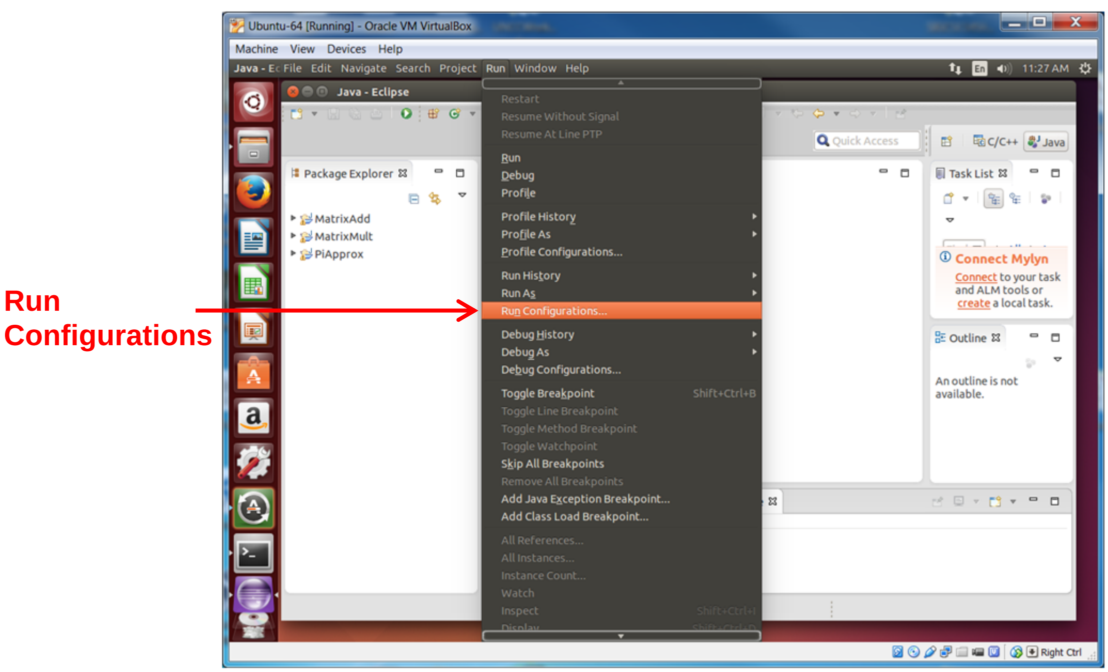
In general, to add libraries with a group name (which makes it easier to use the libraries in other projects), right click the project folder and select **Properties**. Select **Java Build Path** > **Libraries tab** > **Add library** > **User Library** and **Next**. Provide a name for the libraries, in this case say “SeedsMulticore” and click **OK**. Click on **Add External Jars** and navigate through your file system to the Seeds library folder, **~/ParallelProg/Seeds/workspaceMulticore/seedsMulticore/lib**. Select all the jars inside lib (select one jar and then **control-A** to select all the jars). Click **Open**. Finally you should see something like:

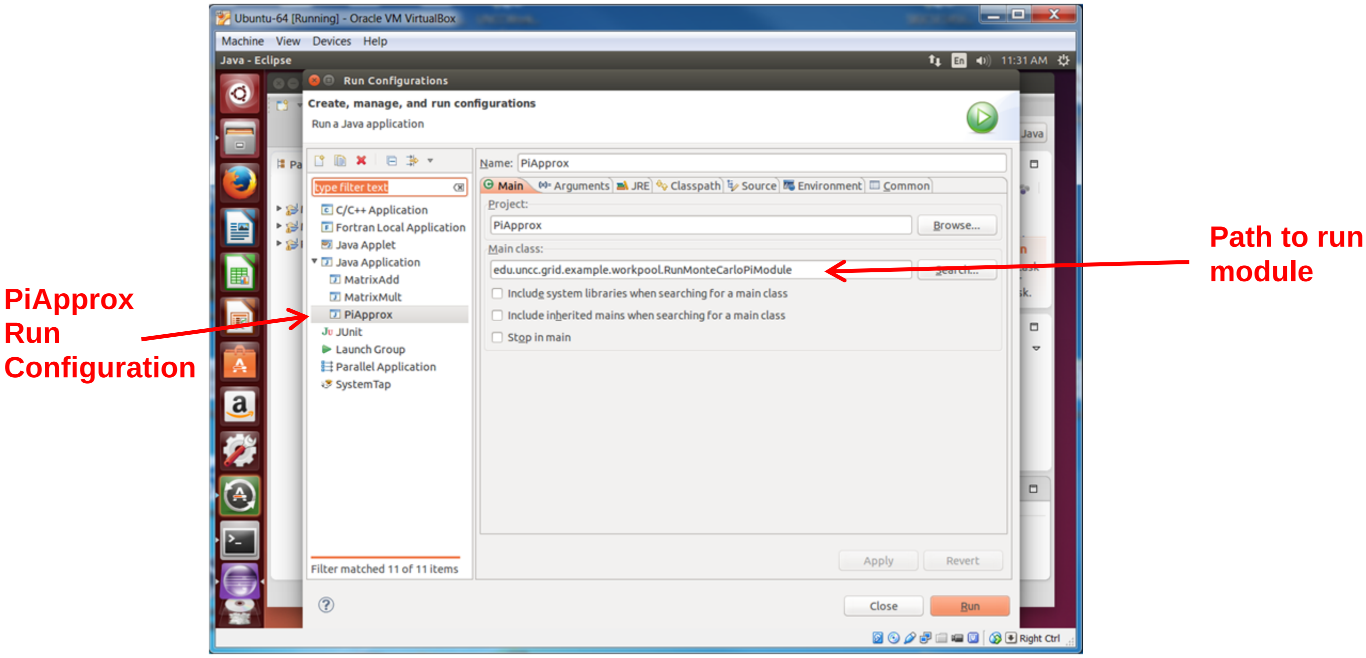


Click **OK**, and get back to the workbench (**Finish** > **OK**). At this point, all unresolved references should vanish. (*Note each of the three versions of Seeds have different Seeds libraries and should be named differently.*)



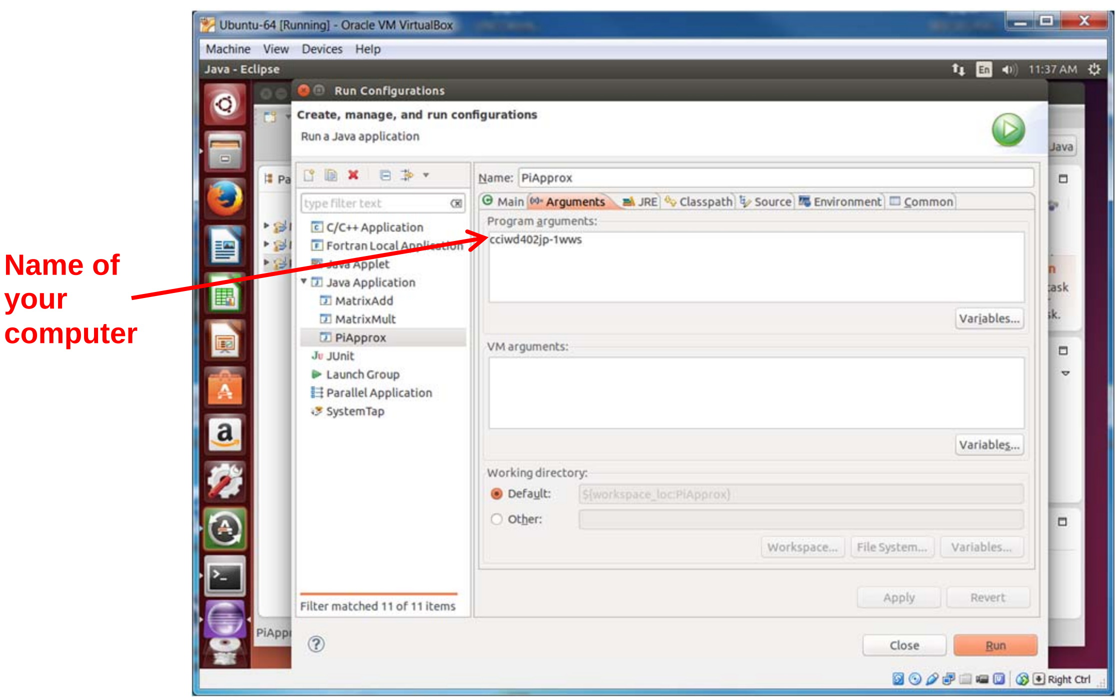
Step 3 Command Line Arguments Before you can run the program, you will need to add a command line argument that provides the computer name, which is read by the bootstrap class **RunMonteCarloPiModule**. This is done in the **Run Configurations**. Go to **Run** > **Run Configurations** … and select the **PiApprox**:





If you are using the prepared VM, **Run Configurations** may already be set up with the correct path to the main class (**edu.uncc.grid.example.workpool.RunMonteCarloPiModule**).

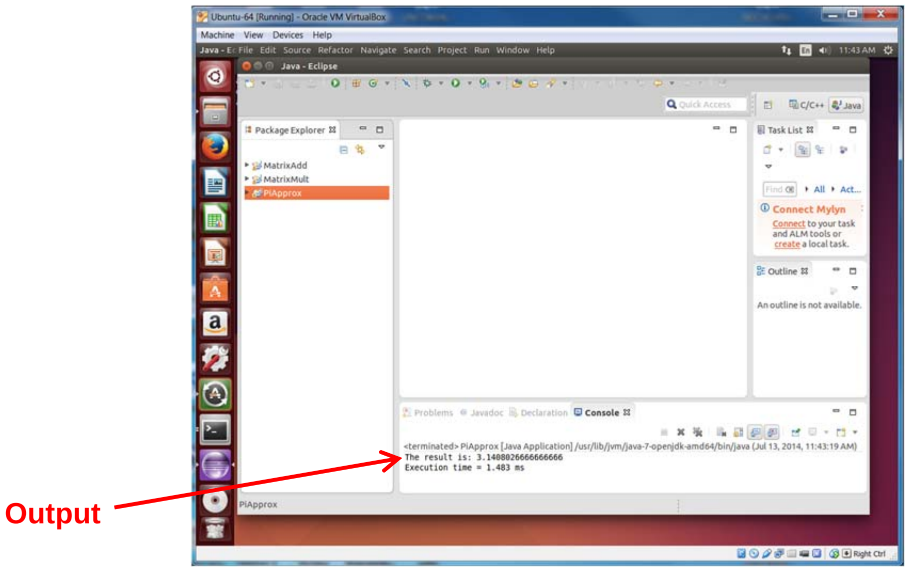
Click the tab named **(x)=Arguments.** For the multicore version of Seeds, the bootstrap class is written to accept one argument, the name of your computer:



If you are using the provided virtual machine, a computer name (**cciwd402jp-1wws**) may exist as an argument. Check whether this name is correct and if not, replace the computer name with the name of your computer. The name of your computer can be found by typing hostname on the command line. *Do NOT use the computer name that you will see from Windows "View system information" or similar, which can have additional characters added to the name. You MUST use the name returned by the hostname command. Include double quotes to make a string if there are one or more spaces in the name.*

Step 4 Run program

Click “Run” to run the project. You should see the project run immediately with output in the console window:



Question 1: How many random numbers were tried by the π approximation program? Explain.

Task 2 Correcting a Flaw in Monte Carlo π Code

There is a potential flaw in the Monte Carlo π code. Although it produces the correct answer, the use of a random number to start each random number sequence in each slave using the same random function possibly causes each sequence to be interrelated. Investigate whether in Java this is a problem and report. In any event, modify the code to avoid the issue altogether and execute the code. Provide the code, a full explanation, and results in your write up.

What to submit for Part 1

Your submission document should include the following:

Task 1

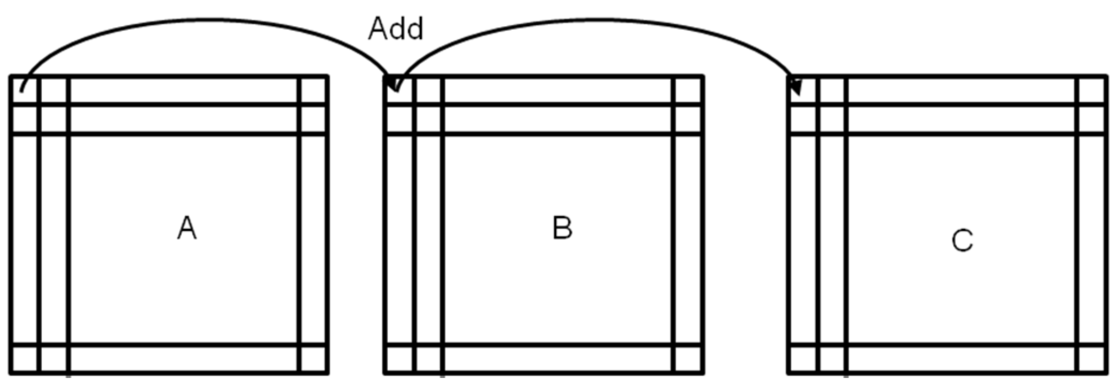
a) Screenshot from compiling and running the **PiApprox** program on your computer and an explanation of output.
b) Answer Question 1

Task 2

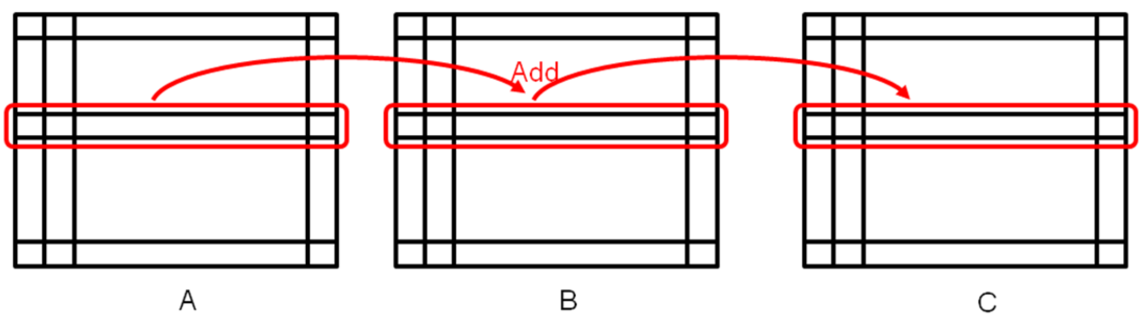
a) A discussion on the potential random number flaw in the code
b) Code listing to avoid the potential random number issue
c) A screenshot showing the code functioning

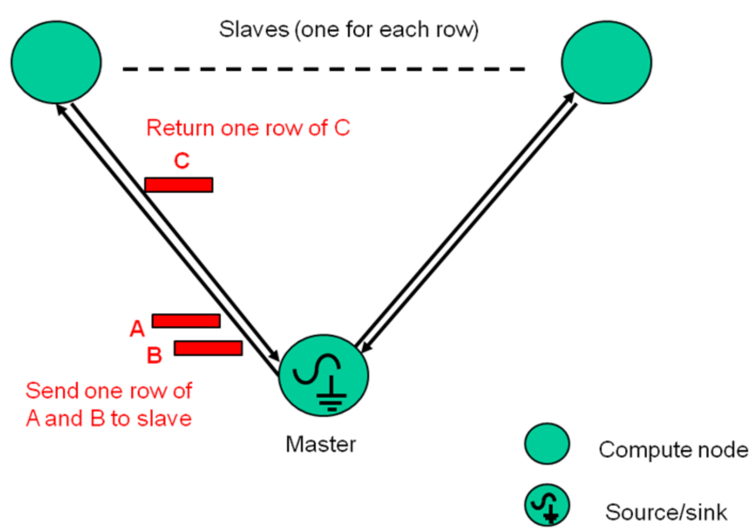
Part 2 Matrix Addition (25%)

Matrix addition, **C** = **A** + **B**, adds corresponding elements of each matrix to form elements of result matrix. Given elements of **A** as $a_{i,j}$ and elements of **B** as $b_{i,j}$, each element of **C** computed as $c_{i,j} = a_{i,j} + b_{i,j}$ ($0 <= i < n$, $0 <= j < m$ for n x m matrices).



Workpool Implementation. Each slave adds one row of **A** with one row of **B** to create one row of **C** (rather than each slave adding single elements):





Matrix addition workpool

There are two Java source programs:

- **MatrixAddModule.java** – the module class that implements the interface for the workpool
- **RunMatrixAddModule.java** – the bootstrap class to deploy the framework and run the code.

MatrixAddModule.java. This code uses 3 x 3 matrices and 3 slaves and is given below:

```
package edu.uncc.grid.example.workpool;
import java.util.Random;
import java.util.logging.Level;
import edu.uncc.grid.pgaf.datamodules.Data;
import edu.uncc.grid.pgaf.datamodules.DataMap;
import edu.uncc.grid.pgaf.interfaces.basic.Workpool;
import edu.uncc.grid.pgaf.p2p.Node;
public class MatrixAddModule extends Workpool {
private static final long serialVersionUID = 1L;
    int[][] matrixA;
    int[][] matrixB;
    int[][] matrixC;
    public MatrixAddModule() {
        matrixC = new int[3][3];
    }

    public void initMatrices(){
        matrixA = new int[][]{{2,5,8},{3,4,9},{1,5,2}};
        matrixB = new int[][]{{2,5,8},{3,4,9},{1,5,2}};
    }
    public int getDataCount() {
    return 3;
    }
    public void initializeModule(String[] args) {
        Node.getLog().setLevel(Level.WARNING);
    }

    public Data DiffuseData(int segment) {

        int[] rowA = new int[3];
        int[] rowB = new int[3];

        DataMap<String, int[]> d =new DataMap<String, int[]>();

        int k = segment;
        for (int i=0;i<3;i++) {
            rowA[i] = matrixA[k][i];
            rowB[i] = matrixA[k][i];
        }
        d.put("rowA",rowA);
        d.put("rowB",rowB);
        return d;
    }
```

```
    public Data Compute(Data data) {

        int[] rowC = new int[3];
        DataMap<String, int[]> input = (DataMap<String,int[]>)data;
        DataMap<String, int[]> output = new DataMap<String, int[]>();

        int[] rowA = (int[]) input.get("rowA");
        int[] rowB = (int[]) input.get("rowB");

        for (int i=0;i<3;i++) {
            rowC[i] = rowA[i] + rowB[i];
        }

        output.put("rowC",rowC);
        return output;
    }

    public void GatherData(int segment, Data dat) {

        DataMap<String,int[]> out = (DataMap<String,int[]>) dat;

        int[] rowC = (int[]) out.get("rowC");

        for (int i=0;i<3;i++) {
            matrixC[segment][i]= rowC[i];
        }

    }

    public void printResult() {

        for (int i=0;i<3;i++) {
            System.out.println();
            for (int j=0;j<3;j++) {
                System.out.print(matrixC[i][j] + "  ");
            }
        }
    }
}           // end of MatrixAddModule
```

MatrixAddModule.java

As in other workpool framework projects, two important classes are imported, called Data and DataMap. Data is used to pass data between the master and slaves and DataMap is used within DiffuseData, Compute, and GatherData methods.

RunMatrixAddModule.java. RunMatrixAddModule.java deploys the Seeds pattern and runs the workpool has the same structure as previously:

```
package edu.uncc.grid.example.workpool;
import java.io.IOException;
import net.jxta.pipe.PipeID;
import edu.uncc.grid.pgaf.Anchor;
import edu.uncc.grid.pgaf.Operand;
```

```
import edu.uncc.grid.pgaf.Seeds;
import edu.uncc.grid.pgaf.p2p.Types;

public class RunMatrixAddModule {
    public static void main(String[] args) {
        try {
            long start = System.currentTimeMillis();

            MatrixAddModule  m = new MatrixAddModule();
            m.initMatrices();

            Thread id = Seeds.startPatternMulticore( new Operand( (String[])null, new Anchor(
        args[0],
                Types.DataFlowRole.SINK_SOURCE), m ), 4 );
            id.join();

            m.printResult();

            long stop = System.currentTimeMillis();
            double time = (double) (stop - start) / 1000.0;
            System.out.println("Execution time = " + time);

        } catch (SecurityException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

RunMatrixAddModule.java –Thread-based multicore version

Task 1 Executing the matrix addition program

Execute the matrix addition code through Eclipse as described for the Monte Carlo pi program. Test the code with the following matrices:

Matrix A
1 2 3
4 5 6
7 8 9
Matrix B
9 8 7
6 5 4
3 2 1

Task 2 Correcting a coding mistake

There is a mistake in the code. Although it produces an answer, it is incorrect. Find this mistake and correct it. Re-execute the code and provide a screenshot of the output.

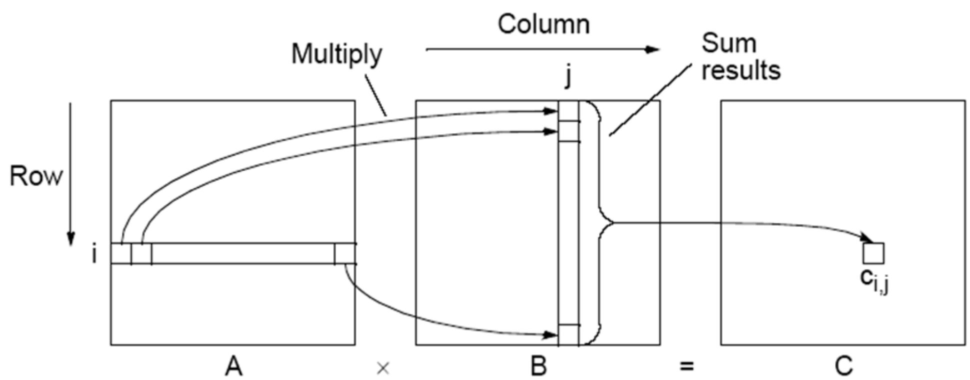
What to submit for Part 2

Your submission document should include the following:

1) Screenshot from compiling and running the **MatrixAdd** program on your computer with specified input data and an explanation of output.
2) Identify the mistake in the code.

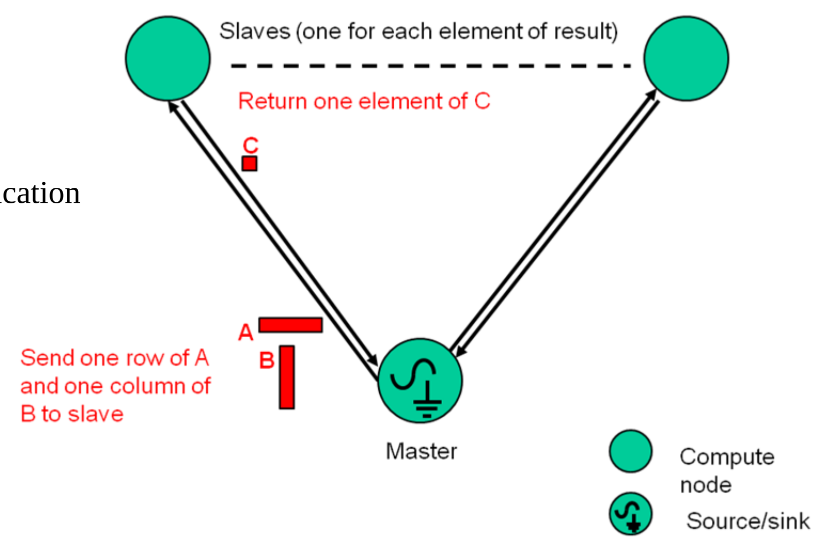
Part 3 Matrix Multiplication, C = A * B (25%)

Multiplication of two matrices, **A** and **B**, to produce matrix **C** is shown below:



In the workpool implementation, one slave computes one $c_{i,j}$ element of the result as shown below:

Matrix multiplication workpool



*(Note rows of **A** and columns of **B** are send to slaves; **B** is not broadcast as in previous programs.)*

There are two Java source programs:

- **MatrixMultModule.java** – the module class that implements the interface for the workpool
- **RunMatrixMultModule.java** – the bootstrap class to deploy the framework and run the code.

MatrixAddModule.java. This code uses 3 x 3 matrices and 9 slaves and is given below:

```
package edu.uncc.grid.example.workpool;
import java.util.logging.Level;
import edu.uncc.grid.pgaf.datamodules.Data;
import edu.uncc.grid.pgaf.datamodules.DataMap;
import edu.uncc.grid.pgaf.interfaces.basic.Workpool;
import edu.uncc.grid.pgaf.p2p.Node;

public class MatrixMultModule extends Workpool {

    private static final long serialVersionUID = 1L;

    int[][] matrixA;
    int[][] matrixB;
    int[][] matrixC;

    public MatrixMultModule() {
        matrixC = new int[3][3];
    }

    public void initMatrices(){
        matrixA = new int[][]{   {3,6,7},
                    {7,4,9},
                    {9,5,7}};
        matrixB = new int[][]{   {3,6,7},
                    {7,4,9},
                    {9,5,7}};
        }
        public int getDataCount() {
            return 9;
        }

        public void initializeModule(String[] args) {

            Node.getLog().setLevel(Level.WARNING);
        }

        public Data DiffuseData(int segment) {

            int[] rowA = new int[3];
            int[] colB = new int[3];

            DataMap<String, int[]> d =new DataMap<String, int[]>();

            int a=segment/3, b = segment%3;
            for (int i=0;i<3;i++){          //Copy one row of A and one column of B into d
                rowA[i] = matrixA[a][i];
                colB[i] = matrixA[i][b];
```

```
        }

        d.put("rowA",rowA);
        d.put("colB",colB);

        return d;
    }

    public Data Compute(Data data) {

        DataMap<String, int[]> input = (DataMap<String,int[]>)data;
        DataMap<String, Integer> output = new DataMap<String, Integer>();

        int[] rowA = (int[]) input.get("rowA");
        int[] colB = (int[]) input.get("colB");

        int out = 0;                    //computation
        for (int i=0;i<3;i++) {
            out += rowA[i] * colB[i];
        }

        output.put("out",out);

        return output;
    }

    public void GatherData(int segment, Data dat) {

        DataMap<String,Integer> out = (DataMap<String,Integer>) dat;

        int answer = out.get("out");
        int a=segment/3, b=segment%3;
        matrixC[a][b]= answer;

    }

    public void printResult(){

        for (int i=0;i<3;i++){
            System.out.println();
            for(int j=0;j<3;j++){
                System.out.print(matrixC[i][j] + "  ");
            }
        }
        System.out.println();
    }
}   // end of MatrixMultiplyModule
```

MatrixMultModule.java

RunMatrixMultModule.java. RunMatrixMultModule.java deploys the Seeds pattern and runs the workpool and is similar to the other Bootstrap class. Just the name of the module class in the code is changed to suit.

Task 1 Executing the matrix multiplication program.

Execute the matrix multiplication code through Eclipse as described for the Monte Carlo pi program. Test the code with the following matrices:

Matrix A	Matrix B
1 2 3	**9 8 7**
4 5 6	**6 5 4**
7 8 9	**3 2 1**

Make sure you show the numeric results that you get in your write up as a screenshot.

Task 2 Correcting a coding mistake

There is a mistake in the code. Although it produces an answer, it is incorrect. Find this mistake and correct it. Re-execute the code and provide a screenshot of the output.

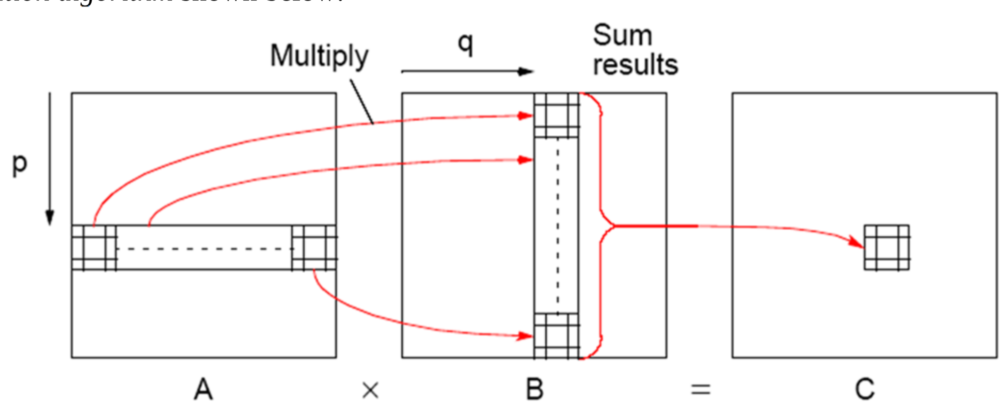
What to submit for Part 3

Your submission document should include the following:

1) Screenshot from compiling and running the **MatrixMult** program on your computer withe the specified input data and an explanation of output.
2) Identify the mistake in the code.

Part 4 Writing Your Own Code – Block Matrix Multiplication (25%)

Modify the sample matrix multiplication program to multiple two N x N matrices using the *block matrix multiplication algorithm* shown below:



Block matrix multiplication

Each slave is given ***s*** rows and ***s*** columns to produce an ***s*** x ***s*** sub matrix answer. Choose N = 8 and s = 2. With s = 2, 16 slaves are needed. Test your program with the following 8 x 8 matrices:

```
        Matrix A                         Matrix B
 1  2  3  4  5  6  7  8     64 63 62 61 60 58 58 57
 9 10 11 12 13 14 15 16     56 55 54 53 52 51 50 49
17 18 19 20 21 22 23 24     48 47 46 45 44 43 42 41
25 26 27 28 29 30 31 32     40 39 38 37 36 35 34 33
33 34 35 36 37 38 39 40     32 31 30 29 28 27 26 25
41 42 43 44 45 46 47 48     24 23 22 21 20 19 18 17
49 50 51 52 53 54 55 56     16 15 14 13 12 11 10  9
57 58 59 60 61 62 63 64      8  7  6  5  4  3  2  1
```

Make sure you show the numeric results that you get in your write up as a screenshot.

Graduate student (5% Extra credit for undergraduate students): Test your program with the two matrices given in the file input2x512x512Doubles (as used in Assignment 3). Use 16 slaves. Compare your results with the posted results. You will need to write code to read the files.

What to submit for Part 4

Your submission document should include the following:

1) Screenshot from compiling and running your block matrix multiplication program on your computer withe specified input data and an explanation of output.

Assignment Submission

Produce a ***single pdf*** document that show that you successfully followed the instructions. Submit by the due date as described on the course home page. *Specify whether you are a graduate or undergraduate student.* Your submission document should include insightful conclusions.

Every part and task specified will be allocated a score so make sure you ***clearly identify*** each part and task you did.

The following is provided for information but not used in this assignment.

Appendix
NETWORK VERSION OF SEEDS FRAMEWORK

The full network version requires an Internet connection. The "NoNetwork" version is a similar JXTA P2P implementation but runs on a single computer without an Internet connection. In each version of the framework, only one Seeds library is different - seeds.jar, seedsNoNetwork.jar, and seedsMulticore.jar. The full network and no-network version of the Seeds libraries can be found in the directory **~/ParallelProg/Seeds /workspaceNetwork**, together with the sources files for the Monte Carlo π calculation and matrix addition/multiplication but you will need to create the Eclipse projects and set up the Seeds library build path.

Mostly, the setup corresponds to the multicore version except it is now necessary to specify the servers, even you only use a single computer.

Specifying the computers to use

The **AvailableServers.txt** file found inside the **seeds** folder within the workspace folder needs to hold the name of the computers being used and other information can be included. For this session, we will only use a local computer and just need to provide its name of the computer. Lines starting with a # are commented out lines. Modify the one uncommented line:

<computerName> local - - - 1 10 GridTwo

replacing <computerName> (or whatever name is there) with the name of your computer and set the number of processors from 1 to however many processors you have (normally just one) and set the number of cores from 10 to the number of cores in each processor on your computer.

The name of your computer can be found by typing hostname on the command line.

Do NOT use the computer name that you will see from "View system information" or similar, which can have additional characters added to the name. You MUST use the name returned by the hostname command.

RunMonteCarloPiModule.java. RunMonteCarloPiModule.java deploys the Seeds pattern and runs the workpool. Below is the code for the network version of the framework:

```
package edu.uncc.grid.example.workpool;
import java.io.IOException;
import net.jxta.pipe.PipeID;
import edu.uncc.grid.pgaf.Anchor;
import edu.uncc.grid.pgaf.Operand;
import edu.uncc.grid.pgaf.Seeds;
import edu.uncc.grid.pgaf.p2p.Types;

public class RunMonteCarloPiModule {

    public static void main(String[] args) {
        try {
            MonteCarloPiModule pi = new MonteCarloPiModule();
```

```
            Seeds.start( args[0] , false);

            PipeID id = Seeds.startPattern(
                new Operand( (String[])null, new Anchor( args[1] ,
Types.DataFlowRoll.SINK_SOURCE), pi ) );
            System.out.println(id.toString() );
            Seeds.waitOnPattern(id);
            System.out.println( "The result is: " + pi.getPi() ) ;

            Seeds.stop();

        } catch (SecurityException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

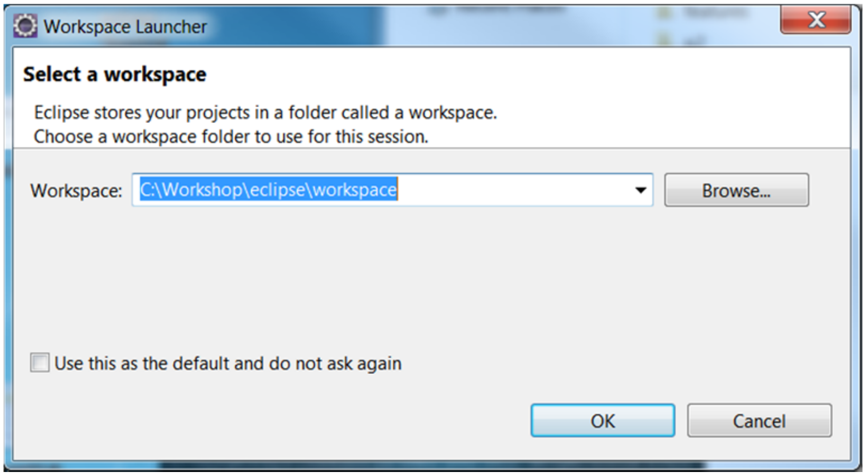
RunMonteCarloPiModule.java – Network version

Seeds is started and deployed on the list servers using the Seeds method start, which takes as its first argument the path to the seeds folder on the local computer. In the code given, the path is provided as the string argument of main (first command line argument, args[0]). The Seeds method startPattern starts the workpool pattern on those computers. It requires as a single argument an Operand object. Creating an Object object requires three arguments. The first is a String list of argument that will be submitted to the remote hosts. The second is an Anchor object specifying the nodes that should have source and sink nodes (the master in this case) which in this provided as the string argument of main (second command line argument, args[1]). The third argument is an instance of MonteCarloPiModule previously created.

To run this code, we will need to provide two command line arguments, the local path to the Seeds folder and the name of the local host. Both could have been hardcoded.

Executing the Monte Carlo π program.

Step 1. Open Eclipse and select the workspace:



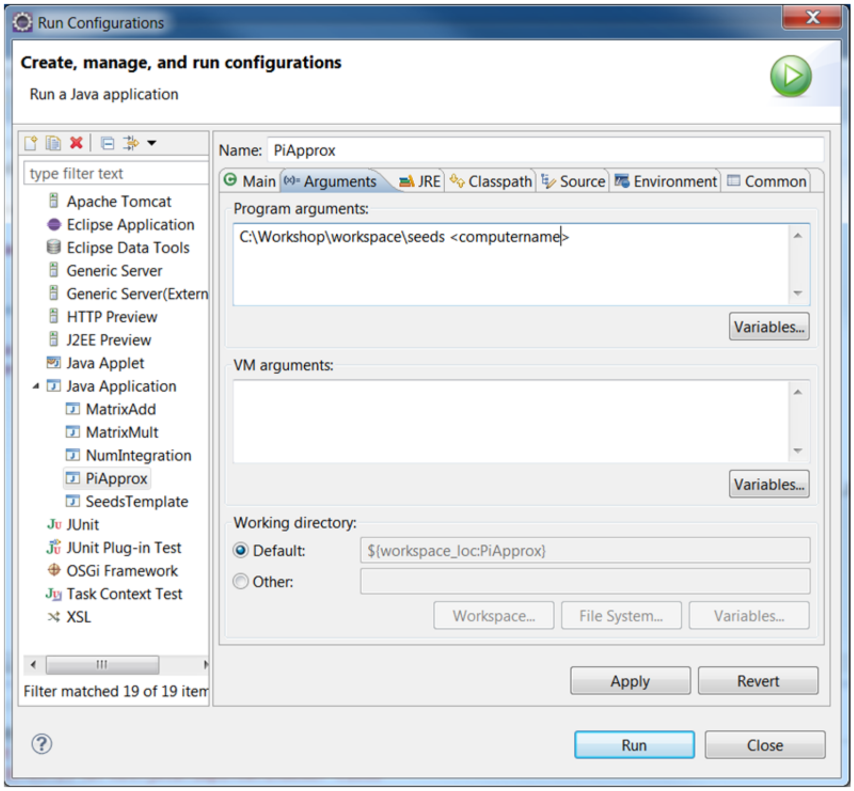
Step 2 Build paths. Make sure the build paths are correct.

Step 3 Command Line Arguments

For the full network version and “NoNetwork version of Seeds, the bootstrap class is written to accept two arguments:

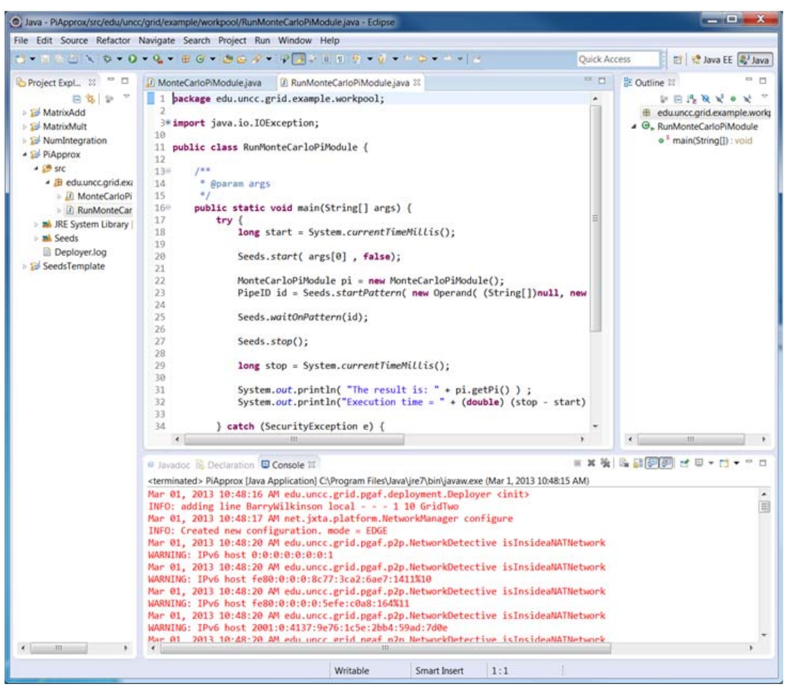
1st argument: Path to where AvailableServers.txt is located
2nd argument: Name of your computer

Enter the two arguments. Includes double quotes to make a string if there are one or more spaces in the path. The name of your computer should be the same as you put in AvailableServers.txt.

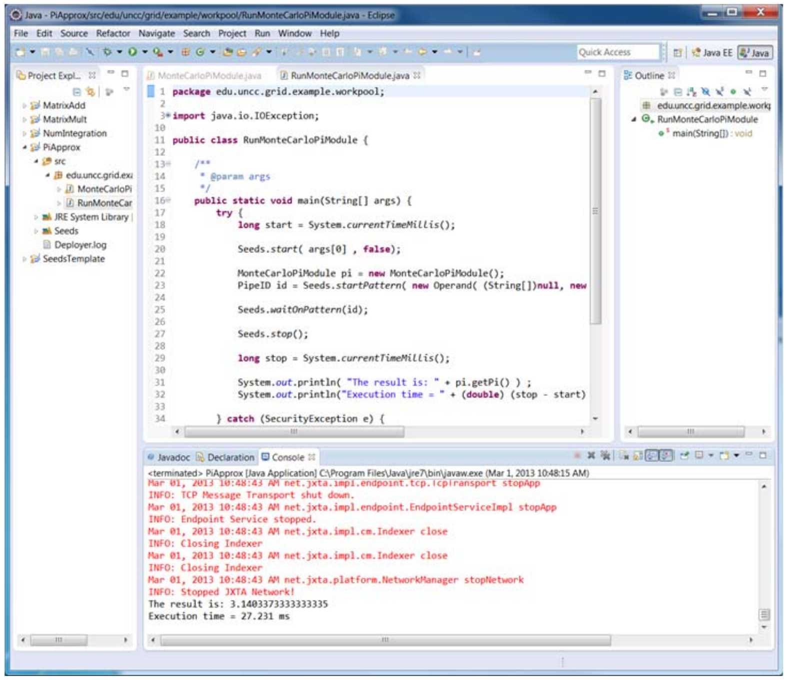


Step 4 Run program

Click "Run" to run the project. The console output will begin with logging messages such as:



with the final result in black at the end:



Issues running program: If you do not get the expected output, see posted FAQs for known issues.