A2: Sharp Edges

TITLE OF MODULE: MCOMD3HPC – High Performance Computing	MODULE COMPONENT: 50% of Module
MODULE TEAM: Dr Vijay Sahota	ASSIGNMENT CONTACT: Dr Vijay Sahota Vijay.Sahota@canterbury.ac.uk
ASSIGNMENT DEADLINE: 7th Jan 2024 14:00	EXPECTED FEEDBACK DATE: 31st Jan Location of Feedback: VIA TURNITIN ON BLACKBOARD
ASSESSMENT TYPE	Report/ Digital artefact
WHERE TO SUBMIT: BLACKBOARD TURNITIN SUBMISSION TOOL If you experience any problems with this system then please contact the Computing Administration Team (computing@canterbury.ac.uk) WHAT TO SUBMIT: **A 2000 (Max) word report of high professional standard & Source code+ data.**	
TITLE OF ASSIGNMENT: A2: Sharp Edges	

Scenario:

You have been employed by a medical imaging company to investigate a possible solution for their technology to work over a very unreliable network of compute nodes. To start things off, you have been tasked to investigate and propose solution(s) to a simple sharpen followed by an edge detect filter – as a means to provide a proof of concept.

It is envisaged that your final implementation should draw from your own programming experience, and from researching the topic further- such that a reliable, scalable, and robust (fault tolerant) solution is needed.

Greater details on solutions and tests are provided in the tasks section.

Contents

Task 1

Note: Please refer to the code comments in the code screenshot snippets for part of the code explanation, safety and justifications.

Code explanation

The author has done the single-threaded implementation as seen in the screenshot:

```java
public class Task1 {
    public static void main(String[] args) {
        //: Setting up by instantiating the classes
        GoldenStandard goldenStandard = new GoldenStandard();
        //^ to compare with tasks 2, 3, and 4
        Utilities utilities = new Utilities();
        //^ for printing a matrix and a generating matrix

        int[][] matrix = utilities.matrixGen();
        //^ generate the 10,000 by 10,000 random matrix
        utilities.printMatrix(matrix, "Randomly generated matrix:");
        //^ prints the initial matrix

        System.out.println("Program is now doing the golden standard solution...");
        int[][] matrixResult = goldenStandard.apply(matrix);
        //^ applying one kernel filter one after the other (sharpen then edge-detection)
        utilities.printMatrix(matrixResult, "Applying Sharpen kernel then Edge-Detection kernel:");
        //^ prints the resulting matrix
        System.out.println("...The golden standard solution has finished");
    }
}
```

The “goldenStandard” instance is what applies the kernel filters where it gets called by the “apply” method. The second kernel (Edge-detection) filter is applied on the resulting matrix that had the first kernel (Sharpen) filter as seen:

```java
    public int[][] apply(int[][] matrixInput){
        matrixInput = applyKernel(1 , matrixInput);
        //^ Apply the "Sharpen" kernel to the generated matrix
        matrixInput = applyKernel(2 , matrixInput);
        //^ Apply the "Edge-Detection" kernel to the resulting
        //^ matrix of the "Sharpen" kernel application.
        return matrixInput;
    }
```

Both kernel filter application uses the “applyKernel” private method as seen:

```java
    public int[][] applyKernel(int kernelType, int[][] matrixInput) {
        int[][] matrixResult = new int[matrixInput.length][matrixInput[0].length];
        //^ set-up
        //: apply kernel to each of the matrix elements
        for (int yAxis = 0; yAxis < matrixInput.length; yAxis++) {
            for (int xAxis = 0; xAxis < matrixInput[0].length; xAxis++) {
                switch(kernelType) {
                case 1: matrixResult[yAxis][xAxis] = this.element(kernelSharpen, matrixInput, yAxis, xAxis); break;
                default: matrixResult[yAxis][xAxis] = this.element(kernelEdgeDetection, matrixInput, yAxis, xAxis); break;
                //^ when kernelType == 2
                }
            }
        }
        return matrixResult;
    }
```

This method uses the “element” private method to apply the kernel filter to each individual element’s value of the matrix. The nested loop goes through each matrix value/element, and the switch statement allows both kernels to be used in the same method. It uses the kernel depending on if “kernelType” is 1 or 2 for Sharpen or Edge-detection kernels respectively which are defined as private fields as seen:

```java
    //: set-up.
    //: kernels declared in this way to allow other developers to easily manipulate its elements' value
    private int[][] kernelSharpen = {
        {0, -1, 0},
        {-1, 5, -1},
        {0, -1, 0}
    };
    private int[][] kernelEdgeDetection = {
        //^ also called as "outline" kernel
        {-1, -1, -1},
        {-1, 8, -1},
        {-1, -1, -1}
    };
```

Source of both kernels' compositions are from Powell (2024). Both kernels' values are programable. In the "element" private method, it applies by cumulating the multiplication of the matrix's elements to each of their corresponding kernel elements as seen:

```java
    private int element(int[][] kernel, int[][] matrixInput, int yAxis, int xAxis) {
        int total = 0;
        //^ set-up
        //: apply kernel to the element with the element's neighbors
        for (int yAxisKernel = -1; yAxisKernel < kernel.length-1; yAxisKernel++) {
            for (int xAxisKernel = -1; xAxisKernel < kernel.length-1; xAxisKernel++) {
                try { total += kernel[yAxisKernel+1][xAxisKernel+1] * matrixInput[yAxis+yAxisKernel][xAxis+xAxisKernel]; }
                catch(ArrayIndexOutOfBoundsException error) { }
                //^ works as zero-padding by not adding operations that include out-of-bounds
                //^ indexes as that will be that same as adding zero.
            }
        }
        return total;
    }
}
```

The method uses zero-padding to deal with matrix elements on the edges or corners by the 'try' and 'catch' statements in the method screenshot. This treats out-of-bounds elements as adding zero which is the same as ignoring them (Powell, 2024).

Code's output

Here is example of implementation but matrix size is 10x10 in the demonstration to fit the console output in this document:

```
Randomly generated matrix:
[5, 9, 6, 2, 2, 0, 0, 0, 9, 9]
[6, 1, 1, 6, 5, 4, 1, 1, 1, 3]
[5, 7, 9, 2, 0, 5, 1, 9, 6, 2]
[9, 9, 1, 9, 5, 6, 9, 5, 8, 7]
[8, 6, 7, 6, 7, 2, 3, 2, 8, 3]
[6, 6, 8, 9, 4, 7, 0, 6, 0, 5]
[3, 1, 2, 3, 3, 3, 6, 4, 4, 9]
[3, 2, 3, 0, 5, 3, 5, 2, 6, 8]
[4, 9, 1, 1, 4, 0, 3, 2, 7, 5]
[6, 8, 1, 3, 7, 5, 4, 3, 6, 4]
# END MATRIX PRINT #
Program is now doing the golden standard solution...
Applying Sharpen kernel then Edge-Detection kernel:
[46, 252, 130, -69, -8, -71, 6, -93, 274, 240]
[113, -255, -216, 144, 99, 86, -21, -80, -203, -34]
[-33, 51, 266, -150, -221, 85, -236, 245, 38, -83]
[129, 101, -328, 241, -2, 54, 230, -108, 47, 136]
[85, -87, 41, -87, 72, -164, -23, -145, 186, -93]
[79, 18, 95, 156, -98, 215, -219, 199, -262, 76]
[26, -112, -74, -34, -46, -76, 112, 5, -58, 204]
[25, -79, 92, -112, 144, -20, 93, -112, 11, 100]
[-56, 202, -114, -57, 26, -205, 20, -101, 77, -22]
[88, 158, -104, 37, 175, 84, 71, -16, 99, 34]
# END MATRIX PRINT #
...The golden standard solution has finished
```

Speed

Here is the average speed based on the mean of 5 runs:

	Trail #1	**Trail #2**	**Trail #3**	**Trail #4**	**Trail #5**	**Mean**
Execution time in milliseconds	4284	4176	4256	4287	4123	4225.2

Task 2

Code explanation

The author has done the muti-threaded implementation of a Shared Memory Model (Delporte-Gallet et al., 2003) as seen in the main method:

```java
    public static void main(String[] args) {
        int[][] matrix = Task2.utilities.matrixGen();
        //^ generate the 10,000 by 10,000 random matrix
        utilities.printMatrix(matrix, "Randomly generated matrix:");
        //^ prints the initial matrix

        //: set up the atomic two-dimentional int array fields
        Task2.matrixResult2 = new int[matrix.length][matrix[0].length];
        Task2.matrixResult2Sharpen = new int[matrix.length][matrix[0].length];

        int[][] matrixResult = Task2.singleThreadedSolution(matrix, Task2.utilities);
        //^ doing the golden standard (with execution duration) for comparison with the multi-threaded solution

        //: threaded (task 2 solution)
        System.out.println("Program is now doing the threaded solution...");
        int threadCount = Task2.utilities.getProcessingEleCount();
        //^ get the processor count form user (must be in range of 0 to maximum available processors)
        long startTime = System.nanoTime();
        //^ start execution time
        Task2.multiThreadedSolution(matrix, threadCount);
        //^ applying one kernel filter after the other (sharpen then edge-detection) but with parallel threads
        Task2.utilities.executionTime(startTime);
        //^ prints execution time of applying the kernel filters
        utilities.printMatrix(Task2.matrixResult2, "multi threaded solution result:");
        //^ prints the resulting matrix
        System.out.println("...The threaded solution has finished");

        System.out.println(Task2.utilities.compareMatrixies(matrixResult, Task2.matrixResult2));
    }
```

Because of the Shared Memory Model, class fields are used as seen:

```java
public class Task2 {
    public static int[][] matrixResult2Sharpen;
    //^ the result of deep-copying "matrixResult2"'s values
    //^ before applying the edge-detection kernel filter.
    //^ The threads can read or write to this concurrently.
    public static boolean toFormat = false;
    //^ tells the threads if you apply kernel filter or
    //^ to deep-copy.
    public static int[][] matrixResult2;
    //^ for the threads to write to, for the matrix result
    //^ of both kernels in the multi-threaded solution, concurrently.
    private static Task2WorkerThread[] threads;
    //^ set-up the thread pool to store the threads
    private static Utilities utilities = new Utilities();
    //^ for printing matrixes and generating matrixes
```

The class's “multiTheadedSolution” method does the multi-threaded solution as seen:

```
    private static void multiThreadedSolution(int[][] matrix, int threadCount){
        //: set-up
        int totalSize = matrix.length * matrix[0].length;
        int sectorSize = totalSize / threadCount;
        //^ calculate element count for each job distribution
        Task2.threads = new Task2WorkerThread[threadCount];
        //^ set-up the thread pool

        for (int index = 0; index < Task2.threads.length; index++) {
            Task2.threads[index] = new Task2WorkerThread();
            Task2.threads[index].start();
        }

        multiThreadedJob(1, matrix, threadCount, sectorSize, totalSize);
        //^ apply the sharpen kernel

        //: deep-copy the result to pass the values, instead of reference, to the next job
        Task2.toFormat = true;
        multiThreadedJob(0, new int [][]{{}}, threadCount, sectorSize, totalSize);
        //^ values of first 2 arguments do not matter as they will not be used
        Task2.toFormat = false;

        multiThreadedJob(2, Task2.matrixResult2Sharpen, threadCount, sectorSize, totalSize);
        //^ apply the edge-detection kernel

        for (int index = 0; index < Task2.threads.length; index++) { Task2.threads[index].done(); }
        //^ acts like a Poison Pill
        for (int index = 0; index < Task2.threads.length; index++) {
            try { Task2.threads[index].join(); } catch (InterruptedException e) { e.printStackTrace(); }
        }
    }
```

It calls the "multiThreadedJob" method to pass the jobs as seen:

```
    private static void multiThreadedSolution(int[][] matrix, int threadCount){
        //: set-up
        int totalSize = matrix.length * matrix[0].length;
        int sectorSize = totalSize / threadCount;
        //^ calculate element count for each job distribution
        Task2.threads = new Task2WorkerThread[threadCount];
        //^ set-up the thread pool

        for (int index = 0; index < Task2.threads.length; index++) {
            Task2.threads[index] = new Task2WorkerThread();
            Task2.threads[index].start();
        }

        multiThreadedJob(1, matrix, threadCount, sectorSize, totalSize);
        //^ apply the sharpen kernel

        //: deep-copy the result to pass the values, instead of reference, to the next job
        Task2.toFormat = true;
        multiThreadedJob(0, new int [][]{{}}, threadCount, sectorSize, totalSize);
        //^ values of first 2 arguments do not matter as they will not be used
        Task2.toFormat = false;

        multiThreadedJob(2, Task2.matrixResult2Sharpen, threadCount, sectorSize, totalSize);
        //^ apply the edge-detection kernel

        for (int index = 0; index < Task2.threads.length; index++) { Task2.threads[index].done(); }
        for (int index = 0; index < Task2.threads.length; index++) {
            try { Task2.threads[index].join(); } catch (InterruptedException e) { e.printStackTrace(); }
        }
    }
```

Each job instance set-ups according to its arguments as seen (kernel values taken from Powell (2024)):

```java
public class Task2Job{
    //: set-up.
    //: kernels declared in this way to allow other developers to easily manipulate its elements' value
    private int[][] kernelSharpen = {
        {0, -1, 0},
        {-1, 5, -1},
        {0, -1, 0}
    };
    private int[][] kernelEdgeDetection = {
        {-1, -1, -1},
        {-1, 8, -1},
        {-1, -1, -1}
    };

    private boolean done = false;
    //^ notify main thread when the job has finished
    private int kernelType;
    //^ determines which kernel filter (Sharpen or Edge-Detection) will be used
    private int[][] matrixInput;
    //^ the matrix to apply the specified kernel filter on
    private int start;
    //^ to calculate the position of the first matrix element to apply the operation on.
    private int end;
    //^ to calculate the position of the last matrix element to apply the operation on.
    private int loops;
    //^ tells the nested loop how many matrix elements to apply the operation on (is calculated)
    private int xAxis;
    //^ the index of the array of int sub-arrays - calculated by "start" and matrix dimensions
    private int yAxis;
    //^ the index of the selected sub-array - calculated by "start" and dimensions

    public Task2Job(int kernelType, int[][] matrixInput, int start, int end) {
        this.kernelType = kernelType;
        this.matrixInput = matrixInput;
        this.start = start;
        this.end = end;
        this.loops = this.end - this.start;
        this.yAxis = start / Task2.matrixResult2.length;
        this.xAxis = start % Task2.matrixResult2.length;
    }
```

Jobs are stored and executed in the worker threads as seen:

```java
public class Task2WorkerThread extends Thread{
    private Task2Job job;
    private boolean jobDone;
    private boolean running;

    public void setJob(Task2Job job) {
        //^ called by main thread when a new job needs doing
        this.job = job;
        this.jobDone = false;
    }
    public void done() { this.running = false;  }
    //^ when thread is no longer needed - gets called by main thread
    public boolean isJobDone() { return this.jobDone; }
    //^ when thread finished current job - gets called by main thread
    public void run() {
        this.running = true;
        while (this.running) {
            if (job != null && !job.isDone()) {
                //^ execute when there is an existing job that is not completed
                this.job.perform();
                //^ where the job actually gets done
                this.jobDone = true;
                this.job = null;
            }
            try { Thread.sleep(10); } catch (InterruptedException e) { }
            //^ to avoid busy-waiting
        }
    }
}
```

The job is similar to the golden solution but can do deep-copy instead when called by "Task2. toFormat" as seen:

```java
    public void perform() {
        if(Task2.toFormat) { this.deepCopyMatrix(Task2.matrixResult2, Task2.matrixResult2Sharpen); return; }
        //: set-up
        int matrixElement;
        int loops = this.loops;
        boolean first = true;
        //: set up the starting coords of the matrix
        int yAxis = this.yAxis;
        int xAxis = this.xAxis;

        //: apply kernel to each of the matrix elements
        for (int y = yAxis; y < this.matrixInput.length; y++) {
            if (loops < 0)/* thread finish its part of the matrix*/{ break; }
            if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis = 0;}
            first = !first;

            for (int x = xAxis; x < this.matrixInput[0].length; x++) {
                loops--;
                switch(this.kernelType) {
                case 1: matrixElement = this.element(kernelSharpen, this.matrixInput, y, x); break;
                default: matrixElement = this.element(kernelEdgeDetection, this.matrixInput, y, x); break;
                //^ when kernelType == 2

                }

                Task2.matrixResult2[y][x] = matrixElement;
                //^ immediately write the new element value to the array of int arrays (concurrently)

                if (loops < 0){ break; }
                //^ if job has finished creating its part of the result
            }

        }
```

Job's "element" method is identical to that of the golden solution as seen:

```java
    private int element(int[][] kernel, int[][] matrixInput, int yAxis, int xAxis) {
        int total = 0;
        //^ set-up
        //: apply kernel to the element with the element's neighbors
        for (int yAxisKernel = -1; yAxisKernel < kernel.length-1; yAxisKernel++) {
            for (int xAxisKernel = -1; xAxisKernel < kernel.length-1; xAxisKernel++) {
                try { total += kernel[yAxisKernel+1][xAxisKernel+1] * matrixInput[yAxis+yAxisKernel][xAxis+xAxisKernel]; }
                catch(ArrayIndexOutOfBoundsException error) { }
                //^ works as zero-padding by not adding operations that include out-of-bounds
                //^ indexes as that will be that same as adding zero.
            }
        }
        return total;
    }
```

When deep-copy is need, the "deepCopyMatrix" method is called:

```java
    private void deepCopyMatrix(int[][] matrixCopy, int[][] matrixPaste){
        //* to prevent the reference passed instead of the values (deep-copy)
        //:set-up
        int loops = this.loops;
        boolean first = true;
        //: set up the starting coords of the matrix
        int yAxis = this.yAxis;
        int xAxis = this.xAxis;

        //: deep-copy each of the matrix elements within specified range
        for (int y = yAxis; y < matrixCopy.length; y++) {
            if (loops < 0)/* thread finish its part of the matrix*/{ break; }
            if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis = 0;}
            first = !first;

            for (int x = xAxis; x < matrixCopy[0].length; x++) {
                loops--;
                matrixPaste[y][x] = matrixCopy[y][x];
                //^ DEEP-copy the element's value to the array of atomic int arrays (concurrently)

                if (loops < 0){ break; }
                //^ if job has finished creating its part of the result
            }
        }
        this.done = true;
    }
}
```

In the main (Task2) class it does the golden solution in the "singleThreadedSolution" method as seen:

```java
    private static int[][] singleThreadedSolution(int[][] matrix, Utilities utilities)/*golden standard*/{
        GoldenStandard goldenStandard = new GoldenStandard();
        //^ instantiating the class that does the golden standard solution

        System.out.println("Program is now doing the golden standard solution...");
        long startTime = System.nanoTime();
        //^ start execution time
        int[][] matrixResult = goldenStandard.apply(matrix);
        //^ applying one kernel filter after the other (sharpen then edge-detection)
        utilities.executionTime(startTime);
        //^ prints execution time of applying the kernel filters
        //utilities.printMatrix(matrixResult, "golden standard solution result:");
        //^ prints the resulting matrix
        System.out.println("...The golden standard solution has finished");

        return matrixResult;
    }
}
```

Matrixes are compared by calling "untilities.compareMatrixies" as seen:

```java
    public String compareMatrixies(int[][] matrix1, int[][] matrix2) {
        int[] result = sameMatrixes( matrix1, matrix2);
        //^ set-up
        if (result[0] == -2) {return "non-identical matrixes - matrix dimensions are different";}
        else if (result[0] == -1) {return "both matrixes are identical in both: dimensions and elements' value";}
        else {return String.format("non-identical matrixes - atleast one element's value is unequal. First unequal at [%d][%d]",result[0],result[1]);}
    }
```

Where it calls the "sameMatrixies" private method to do simple subtraction of both results to prove both solution's results identical to each other as seen:

```
    private int[] sameMatrixes(int[][] matrix1, int[][] matrix2) {
        int [][] matrixResult = new int[matrix1.length][matrix1[0].length];
        //^ set-up for the subtraction proof
        //* {-1,0} means same, otherwise matrixes are not the same.
        //* {-2,0} means different dimensions.
        //: subtraction proof
        if ( (matrix1.length != matrix2.length) || (matrix1[0].length != matrix2[0].length) ) { return new int[] {-2,0}; }
        //^ check dimensions
        for (int yAxis = 0; yAxis < matrix1.length; yAxis++) {
            for (int xAxis = 0; xAxis < matrix1.length; xAxis++) {
                matrixResult[yAxis][xAxis] = matrix1[yAxis][xAxis] - matrix2[yAxis][xAxis];
                //^ subtract each individual element value from the corresponding one from the other matrix at a time
            }
        }
        printMatrix(matrixResult, "subtraction proof (a simple subtraction between both matrixes):");
        //^ print the subtraction proof

        //: check the subtraction proof
        for (int yAxis = 0; yAxis < matrix1.length; yAxis++) {
            for (int xAxis = 0; xAxis < matrix1.length; xAxis++) {
                if (matrixResult[yAxis][xAxis] != 0) { return new int[] {yAxis,xAxis}; }
                //^ return location of first non-identical element value
            }
        }
        return new int[] {-1,0};
        //^ executes if matrixes are identical
    }
```

Thread safety

Threads have no race condition for “toFormat”, “matrixResult2Sharpen”, nor “matrixResult2”. This is Because the threads only read “toFormat”. Meanwhile, for “matrixResult2Sharpen” and “matrixResult2”, each thread writes to their corresponding section of the two-dimensional array exclusively – making them atomic in nature. The threads automatically close when they done their job that was started by “.start()”, hence there are no zombie threads. There are brief “.sleep()” sleep statements to avoid busy-waiting (threads consuming CPU usage, preventing the CPU to switch to other threads). The “done” public method of the worker thread acts as a poison pill.

Code’s output

Here is example of implementation (with 3 threads selected for the multi-threaded solution) but matrix size is 10x10 in the demonstration to fit the console output in this document:

```
Randomly generated matrix:
[8, 6, 3, 5, 2, 5, 1, 9, 2, 0]
[3, 0, 7, 4, 5, 0, 7, 2, 9, 4]
[6, 1, 8, 3, 9, 5, 4, 5, 0, 3]
[9, 4, 4, 5, 0, 1, 5, 1, 2, 3]
[1, 3, 6, 7, 7, 6, 3, 8, 9, 6]
[6, 3, 1, 9, 1, 6, 4, 1, 9, 2]
[9, 4, 0, 5, 5, 1, 8, 8, 9, 7]
[1, 0, 4, 1, 4, 5, 0, 8, 2, 2]
[7, 1, 9, 9, 3, 7, 8, 8, 5, 9]
[1, 2, 2, 8, 1, 9, 1, 5, 5, 4]
# END MATRIX PRINT #
Program is now doing the golden standard solution...
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
1080700, 1, 0
golden standard solution result:
[245, 120, -62, 106, -66, 181, -176, 299, -123, -85]
[-29, -233, 144, -84, 37, -256, 193, -238, 275, 52]
[128, -184, 210, -161, 290, 59, -21, 122, -189, 37]
[277, -39, -63, 23, -250, -149, 96, -168, -82, 10]
[-160, -33, 81, 46, 142, 110, -89, 196, 144, 95]
[105, -27, -164, 229, -272, 116, -31, -316, 111, -197]
[267, 50, -132, 85, 92, -181, 219, 85, 128, 167]
[-139, -155, 63, -209, 19, 77, -337, 120, -230, -126]
[288, -147, 213, 166, -133, 68, 132, 103, -68, 276]
[-57, 13, -146, 201, -212, 292, -231, 55, 22, 2]
# END MATRIX PRINT #
...The golden standard solution has finished
Program is now doing the threaded solution...
Java runtime have 11 available effective processing elements (minus the main thread)
How many processing elements do you want for this operation?3
Human counterpart compliant
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
15479300, 15, 0
multi threaded solution result:
[245, 120, -62, 106, -66, 181, -176, 299, -123, -85]
[-29, -233, 144, -84, 37, -256, 193, -238, 275, 52]
[128, -184, 210, -161, 290, 59, -21, 122, -189, 37]
[277, -39, -63, 23, -250, -149, 96, -168, -82, 10]
[-160, -33, 81, 46, 142, 110, -89, 196, 144, 95]
[105, -27, -164, 229, -272, 116, -31, -316, 111, -197]
[267, 50, -132, 85, 92, -181, 219, 85, 128, 167]
[-139, -155, 63, -209, 19, 77, -337, 120, -230, -126]
[288, -147, 213, 166, -133, 68, 132, 103, -68, 276]
[-57, 13, -146, 201, -212, 292, -231, 55, 22, 2]
# END MATRIX PRINT #
...The threaded solution has finished
subtraction proof (a simple subtraction between both matrixes):
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
# END MATRIX PRINT #
both matrixes are identical in both: dimensions and elements' value
```

Please note that due to the small matrix scale, the multi-thread's quicker parallel advantage diminishes.

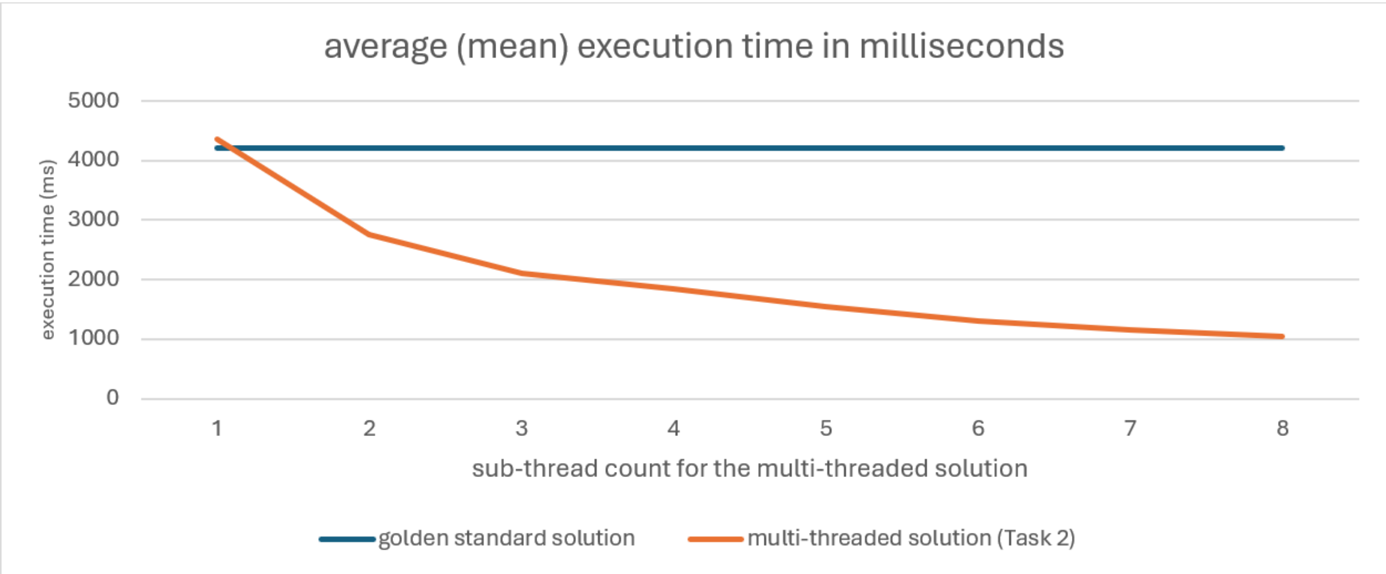
Speed comparison

Here is the comparison table of the golden solution and the multi-threaded solution with different thread counts (in milliseconds):

(ms)	golden standard	1 sub-thread	2 sub-threads	3 sub-threads	4 sub-threads	5 sub-threads	6 sub-threads	7 sub-threads	8 sub-threads
trail #1	4306	4229	2727	2172	1786	1548	1303	1148	1048
trail #2	4189	4268	2792	2094	1718	1545	1339	1111	1047
trail #3	4154	4628	2788	2098	1852	1559	1313	1144	1071
trail #4	4187	4288	2772	2202	1860	1528	1285	1155	1023
trail #5	4223	4399	2674	1991	1956	1522	1321	1185	1016
mean	4211.8	4362.4	2750.6	2111.4	1834.4	1540.4	1312.2	1148.6	1041

Note: sub-thread count does not include main thread. For example, a 8 sub-threaded trail actually uses 9 threads.

Here is the table as a graph for better comparison:



As expected, the multi-threaded is faster depending on how many threads are used, such as 3 threads being faster than 2. But the scale slowly diminishes over higher thread count due the multi-threaded solution doing more steps such as setting up and instantiating the threads and deep-copying to ensure data gets passed instead of reference.

Task 3

Code's explanation and justification

Code is structured to simulate a HPC compute cluster by simulating the Thread-per-Node Model. Due to the bottleneck of network communication between nodes, such as delays (200ms), the author has decided to change the Shared Memory Model (Delporte-Gallet et al., 2003) in to in favour with the Message-Passing Model with the Message Passing Interface (MPI) (Delporte-Gallet et al., 2003; Shafi et al., 2009).

To properly simulate the network (by the Message-Passing Model), the author needs to limit network restrictions of message delays, message size and no node-to-node pass by reference usage (Delporte-Gallet et al., 2003). The job code has been split into the kernel application job and the int[][] normalization job the reduce message size. Both jobs implement 'Task3Job' (as seen below) to allow both jobs to be stared and used in the thread pool under one object instance type. Both jobs takes serialized data and primitives only to prevent any references for passing between the threads (Opyrchal and Prakash, 1999).

```java
import java.io.Serializable;
//^ For 'Task3Job' to implement 'Serializable'

public interface Task3Job extends Serializable{
    //^ any class implementing this interface also implements 'Serializable' so the classes'
    //^ instances are able to get serialized and deserialized.
    //* Interface allows the worker thread to accept any kind of job (deep copy and kernel application).
    //* Implemented by the 'Task3JobKernelApplication', 'Task3JobNormalization', and
    //* 'Task3JobPoisonPill' classes.
    boolean isDone();
    //^ allows sub-thread to check if the job has finished yet (returns true if '.perform()' finishes)
    int[] perform();
    //^ Executes the main body (in the 'perform' public method) of the job in the sub-thread in
    //^ concurrency.
    int[][] deserializeIntArrArr(byte[] data);
    //^ Deserialises the 2d int array.
    //^ Due to the argument being serialized, reference calls are encapsulated inside of the
    //^ the method itself meaning that no "pass by reference" is done over the threads/nodes
    //^ in the computer network in the method's body.
    //^ Despite byte[] being an object, in a network simulation (Thread-per-Node model),
    //^ byte[] serves container for raw data for Messaging Passing Interface (MPI), rather
    //^ than a reference to the original data structure - thus the method call also does
    //^ not pass any direct reference calls between threads.
}
```

Code comments (in 'deserializeIntArrArr'), justifying serialization, are supported by ().

Another decision based on the Thread-per-Node/Message-Passing Model restrictions, is to use integer primitives instead of enumerations. This is because enumerations will make the serialization process, to prevent cross-node reference passes, more complex and thus takes more computational time to complete.

Because the Thread-per-Node Model is used, the main code and the thread pool must be in the same main thread to simulate the single main/master node.

The thread pool deals with assigning the jobs to the sub-threads and retrieving their stored results. Thread pool's fields, constructor and imports can be seen below:

```java
//: all relevant imports
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;

public class Task3ThreadPool {
    //* thread pool is used to a Thread-per-Node model (HPC cluster of computer nodes).

    private boolean sleep;
    //^ stops the thread pool from searching for threads to
    //^ message the jobs to when there are no jobs left.
    private Task3WorkerNode[] threads;
    //^ where the worker threads are referenced.
    //^ Each thread ID will be their index in the array
    private ArrayList<Task3Job> jobQueue;
    //^ FIFO stack structure (first in first out).
    //^ Contains the jobs as messages.
    private int[][] threadJobResults;
    //^ job results that are concurrently written to by the threads (atomic).
    //^ Each thread is to write to corresponding element, even if overwriting,
    //^ thus the index serves as the thread's ID (identification).
    private int maxJobs;
    //^ For code integrity purposes - if another developer changes initial
    //^ matrix size to a small size it will still work (STILL MUST BE ATLEAST 1x1).
    public Task3ThreadPool(int threadCount) {
        //: set-up the fields
        this.sleep = false;
        //^ activate the thread pool
        jobQueue = new ArrayList<Task3Job>();
        //^ for good practice and scalability - the data structure has dynamic size
        threadJobResults = new int[threadCount][];
        //^ is a jagged array, because jobs' returned int arrays' length are not predefined

        //: set-up and fill the thread pool
        this.threads = new Task3WorkerNode[threadCount];
        for (int index = 0; index < threadCount; index++) {
            this.threads[index] = new Task3WorkerNode();
            //^ worker thread instance
            this.threads[index].start();
            //^ start the thread's execution
        }
    }
```

Using an array list, rather than just an array, take more computational time (). This because array lists have more overhead such as using methods to access and write to elements rather than direct access by index, more overhead in memory, etc. However, it is still used because it more accurately reflects how the job queue works in a thread pool.

The thread pool gets controlled by the main ('Task3') class by the public methods as seen:

```java
    //: public methods that are called by main ('Task3') class
    public int threadCount() /*getter method*/ { return this.threads.length; }
    public void wakeUp() { this.start(); }
    //^ notifies the thread pool - simulates the ".notify()" method call
    public void enqueueJob(Task3Job job) /*add method (FIFO style)*/ { jobQueue.add(job); }
    //^ enqueues a job by appending it to the end of the stack
    public byte[] getResults() /*getter method*/ { return this.serializeIntArrArr(this.threadJobResults); }
    public void clear() /*reset method*/ {
        //* not exactly aligned with the Thread-per-Node Model but the program must not exceed the Java
        //* default maximum heap memory limit, which cannot be changed in-program.
        //* The lower it is, the better the program's scalability.
        //: these data are, at this point of time, redundant so they must be deleted
        this.threadJobResults = new int[this.threads.length][];
        for (int index = 0; index < this.threads.length; index++) { threads[index].clear(); }
        //^ clears each worker node's stored result
        try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
        //^ to impose a delay of 200ms to simulate the speeds of modern computers and cluster networks.
        //^ Tells all threads to delete their now redundant stored result.
    }
```

It also have the major methods that simulate the main functionality of a thread pool.

Firstly, is the "start" private methods that assigns the jobs to the threads as seen:

```java
    private void start() {
        //* called by ".wakeUp()" by other classes.
        //* Assign jobs and gather results.
        this.maxJobs = jobQueue.size();
        this.sleep = false;
        while (!this.sleep) {
            try { Thread.sleep(1); } catch (InterruptedException e) { e.printStackTrace(); }
            //^ to avoid busy-waiting
            for (int index = 0; index < this.maxJobs; index++) {
                //* assign a job per worker thread
                if (this.jobQueue.size() == 0) { break; }
                //^ cannot assign a non-existant job to a worker thread
                threads[index].setJob(this.serializeJob(this.dequeueJob()));
                //^ take from FIFO stack and into a serialized MPI message to the worker thread
            }
            try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
            //^ to impose a delay of 200ms for each computational job executed to simulate the
            //^ speeds of modern computers and cluster networks.
            //^ this node-to-node message is the serialized job from the main thread to the sub-thread.
            //^ Thread-pool can send multiple messages without waiting for delay of the sub-thread
            //^ receiving the message.
            if (this.jobQueue.size() == 0) { this.sleep = true; }
            //^ go to sleep when there are currently no more jobs
        }
        this.requestResults();
        //^ after assigning jobs, we gather all results when all threads finish their given jobs
    }
```

It calls "request" private method to then fetch the results as seen:

```java
    private void requestResults() {
        boolean allResults = false;
        //^ dictates if it thread pool should keep waiting for all results or not
        boolean[] alldone = new boolean[this.maxJobs];
        //^ only fetch results from the relevant threads to save processing time
        while (!allResults) {
            //* far faster than the alternative of constantly checking a sub-thread job until it
            //* is done one at a time.
            for (int index = 0; index < this.maxJobs; index++) {
                //* get all results to if each thread has finished the job or not
                alldone[index] = threads[index].isJobDone();
                //^ store the each job status
            }
            try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
            //^ to impose a delay of 200ms for each computational job executed to simulate the
            //^ speeds of modern computers and cluster networks.
            //^ These node-to-node messages are to check if the all sub-thread has finished
            //^ their given jobs.
            allResults = true;
            //^ Assume that all threads have finished
            for (int index = 0; index < this.maxJobs; index++) {
                //* only gathers threads that were given relevant jobs
                if(alldone[index] == false) {
                    //^ current thread hasn't finished?
                    allResults = false;
                    //^ if one thread isn't finished then all threads are not finished!
                    break;
                    //^ no point to continue so break off to save computational time
                }

            }
        }
        for (int index = 0; index < this.maxJobs; index++) {
            //* gather results of all relevant threads
            threadJobResults[index] = this.deserializeIntArr(this.threads[index].requestResult());
            //^ deserialize raw data MPI message into the fetched int[] result to be stored
        }
        try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
        //^ to impose a delay of 200ms for each computational job executed to simulate the
        //^ speeds of modern computers and cluster networks.
        //^ These node-to-node messages are to get the serialized results from the sub-threads.
    }
```

When passing jobs, it first serialises them as seen:

```java
    private byte[] serializeJob(Task3Job job){
        //* between thread pool and worker nodes/threads - to reduce reference calls as much as possible.
        //* To send serialized copies so referencing between nodes are limited to twice per communication/message.
        ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
        //^ holds and formats the byte data of the argument
        ObjectOutputStream outStream = null;
        //^ 'ObjectOutputStream' serialize and write objects.
        //^ initialized as null because of the try-statement but null should never used

        try {
            //^ try-statement to deal with possible IOException errors (mandated by compiler)
            outStream = new ObjectOutputStream(byteOutStream);
            //^ 'ObjectOutputStream' initialized for serialization and writing to 'byteOutStream'
            outStream.writeObject(job);
            //^ serialize the argument and writes it to 'byteOutStream'
            outStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException e) { e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance

        return byteOutStream.toByteArray();
        //^ Returns a byte array containing the serialized data
    }
```

It also deserialize the job's results as seen:

```java
    private int[] deserializeIntArr(byte[] data){
        ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
        //^ holds and formats the byte data of the argument.
        ObjectInputStream inStream;
        //^ 'ObjectOutputStream' serialize and write objects.
        int[] deserialized = null;
        //^ initialized as null because of the try-statement but null should never be returned

        try {
            //^ try-statement to deal with potential IOException or ClassNotFoundException errors (mandated by compiler)
            inStream = new ObjectInputStream(byteInStream);
            //^ 'ObjectInputStream' initialized for deserialization and writing to 'byteInStream'
            deserialized = (int[]) inStream.readObject();
            //^ deserialize the byte array argument and writes it to 'byteOutStream'.
            //^ the only code in this method that can potentially cause a 'ClassNotFoundException' error.
            inStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {  e.printStackTrace(); }
        //^ for possible errors relating to deserialization, int[] parsing, and the 'ObjectInputStream' instance

        return deserialized;
        //^ Returns a int array containing the deserialized data
    }
```

It also serialises the results of all relevant threads' jobs to send to the main code but, unlike the other serialization and deserialization methods used to prevent pass be reference, it does it only for deep copying as seen:

```java
    private byte[] serializeIntArrArr(int[][] array){
        //* between thread pool and main - same thread but still used for deep-copying
        ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
        //^ holds and formats the byte data of the argument
        ObjectOutputStream outStream = null;
        //^ 'ObjectOutputStream' serialize and write objects.
        //^ initialized as null because of the try-statement but null should never used

        try {
            //^ try-statement to deal with possible IOException errors (mandated by compiler)
            outStream = new ObjectOutputStream(byteOutStream);
            //^ 'ObjectOutputStream' initialized for serialization and writing to 'byteOutStream'
            outStream.writeObject(array);
            //^ serialize the 2d int argument and writes it to 'byteOutStream'
            outStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException e) { e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance

        return byteOutStream.toByteArray();
        //^ Returns a byte array containing the serialized data
    }
}
```

The jobs are handled by the worker threads where you can see the worker thread class's fields, constructors, and public methods (that are called by other classes):

```
//: import for serialization.
//: As compute nodes have copies of the built-in libraries, this is not shared
//: memory - sticking to the Thread-per-Node model (HPC cluster simulation).
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Task3WorkerNode extends Thread {
    //* simulates an individual compute node in a HPC cluster in accordance to
    //* the Thread-per-Node model.
    private Task3Job job;
    //^ stores the current job
    private boolean jobDone = true;
    //^ toggles when either a new job is given or the current job has finished
    private boolean running;
    //^ acts as a Poison Pill
    private int[] jobResult;
    //^ result of current job is caught and stored here

    public void setJob(byte[] job) {
        //^ called by main thread when a new job needs doing
        this.job = this.deserializeJob(job);
        this.jobDone = false;
    }
    public void done()/*setter method*/{ this.running = false;  }
    //^ when thread is no longer needed - gets called by main thread
    public boolean isJobDone()/*getter method*/{ return this.jobDone; }
    //^ when thread finished current job - gets called by main thread
    public byte[] requestResult() { return this.serializeIntArr(this.jobResult); }
    //^ in compute clusters, results are stored in the worker nodes before retrieval
    public void clear() { this.jobResult = null; }
    //^ to help prevent the Java program from exceeding the default maximum heap size limit
```

The worker thread constantly checks for tasks, do them and store the job's results in the "run". The method executes once but continuously until the thread is no longer needed and closes as seen:

```
    public void run() {
        this.running = true;
        //^ dictates if the thread should continue running or end
        while (this.running) {
            //* while-loop runs constantly till thread must end
            if (job != null && !this.jobDone) {
                //^ execute when there is an existing job that is not completed
                jobResult = this.job.perform();
                //^ where the job actually gets done
                if (jobResult.length == 0 ) { this.running = false; }
                //^ shuts down thread (after this loop) if job is a Poison Pill
                this.jobDone = true;
                //^ prevents node from repeating the same job twice (clears redundancy)
                this.job = null;
                //^ prevents node from running empty code before first job
            }
            try { Thread.sleep(1); } catch (InterruptedException e) { }
            //^ to avoid busy-waiting
        }
    }
```

Despite the methods not being called outside class, it remains public because so does the "Thead" class that the worker thread class inherits.

The worker thread class also have a serialization and deserialization private methods, for the input and resulting output respectively, to (again) prevent pass by reference between nodes as seen:

```
    private byte[] serializeIntArr(int[] array){
        //* to send the result as a serialized raw data message instead of pass by reference
        ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
        //^ holds and formats the byte data of the argument
        ObjectOutputStream outStream = null;
        //^ 'ObjectOutputStream' serialize and write objects.
        //^ initialized as null because of the try-statement but null should never used

        try {
            //^ try-statement to deal with possible IOException errors (mandated by compiler)
            outStream = new ObjectOutputStream(byteOutStream);
            //^ 'ObjectOutputStream' initialized for serialization and writing to 'byteOutStream'
            outStream.writeObject(array);
            //^ serialize the argument and writes it to 'byteOutStream'
            outStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException e) { e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
        return byteOutStream.toByteArray();
        //^ Returns a byte array containing the serialized data
    }
    private Task3Job deserializeJob(byte[] data){
        //* to deserialized the raw data message into a job
        ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
        //^ holds and formats the byte data of the argument
        ObjectInputStream inStream;
        //^ 'ObjectOutputStream' serialize and write objects.
        Task3Job deserialized = null;
        //^ initialized as null because of the try-statement but null should never be returned

        try {
            //^ try-statement to deal with potential IOException or ClassNotFoundException errors (mandated by compiler)
            inStream = new ObjectInputStream(byteInStream);
            //^ 'ObjectInputStream' initialized for deserialization and writing to 'byteInStream'
            deserialized = (Task3Job) inStream.readObject();
            //^ deserialize the byte array argument and writes it to 'byteOutStream'.
            //^ the only code in this method that can potentially cause a 'ClassNotFoundException' error.
            inStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {  e.printStackTrace(); }
        //^ for possible errors relating to deserialization, job parsing, and the 'ObjectInputStream' instance
        return deserialized;
        //^ Returns a job containing the deserialized data
    }
}
```

As mentioned, the kernel application and normalization (to deal with jagged integer arrays) are their own jobs as seen in the followings:

Kenel application:

```java
//: imports
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class Task3JobKernelApplication implements Task3Job {
    //: set-up.
    //: Kernels declared in this way to allow other developers to easily manipulate its elements' value
    //: and program will still work as intended.
    private int[][] kernelSharpen = {
        //* element's values can be changed if wished by a developer
        {0, -1, 0},
        {-1, 5, -1},
        {0, -1, 0}
    };
    private int[][] kernelEdgeDetection = {
        //* element's values can be changed if wished by a developer
        {-1, -1, -1},
        {-1, 8, -1},
        {-1, -1, -1}
    };

    private boolean done = false;
    //^ notify main thread when the job has finished
    private int kernelType;
    //^ determines which kernel filter (Sharpen or Edge-Detection) will be used
    private int[][] matrixInput;
    //^ the matrix to apply the specified kernel filter on
    private int start;
    //^ to calculate the position of the first matrix element to apply the operation on.
    private int end;
    //^ to calculate the position of the last matrix element to apply the operation on.
    private int[] result;
    //^ where the result of the job is stored to be later fetched by the worker node/thread

    public Task3JobKernelApplication(int kernelType, byte[] matrixInput, int start, int end) {
        this.kernelType = kernelType;
        this.matrixInput = this.deserializeIntArrArr(matrixInput);
        this.start = start;
        this.end = end;
    }
    @Override
    public boolean isDone()/*getter method*/{ return this.done; }
```

```
    public int[] perform() {
        //: set-up
        int matrixElement;
        //^ where the result of applying kernel, on a element, is temporarily stored
        int loops = this.end - this.start;
        //^ tells the nested loop how many matrix elements to apply the operation on (is calculated)
        this.result = new int[loops+1];
        //^ field assignment is based on local variable, hence it is here instead of in the constructor
        boolean first = true;
        //: set up the starting coords of the matrix
        int yAxis = start / this.matrixInput[0].length;
        //^ the index of the array of int sub-arrays - calculated by "start" and matrix dimensions
        int xAxis = start % this.matrixInput[0].length;
        //^ the index of the selected sub-array - calculated by "start" and dimensions

        //: apply kernel to each of the matrix elements
        for (int y = yAxis; y < this.matrixInput.length; y++) {
            if (loops < 0)/* thread finish its part of the matrix*/{ break; }
            if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis = 0;}
            first = !first;
            //^ no longer the first applied sub-array

            for (int x = xAxis; x < this.matrixInput[0].length; x++) {
                loops--;
                //: applies kernel of element depending on which kernel is selected.
                //: Didn't use enumerations (enum) because it will make the serialization process, to
                //: prevent cross-node reference passes, more complex and takes more
                //: computational time.
                switch(this.kernelType) {
                case 1: matrixElement = this.element(kernelSharpen, this.matrixInput, y, x); break;
                //^ '1' corresponds to applying the Sharpen kernel
                default: matrixElement = this.element(kernelEdgeDetection, this.matrixInput, y, x); break;
                //^ when kernelType == 2.
                //^ '2' corresponds to applying the Edge-Detection kernel.
                //^ using default satisfy compiler so I do not need to initialize 'matrixElement' with
                //^ a redundant value.
                }

                this.result[((result.length-1)-loops)-1] = matrixElement;
                //^ immediately write the new element value to the array of int arrays (concurrently)

                if (loops < 0){ break; }
                //^ if job has finished creating its part of the result
            }
        }
        this.done = true;
        //^ job finished
        return this.result;
        //^ store result for worker thread to fetch
    }
```

```java
    private int element(int[][] kernel, int[][] matrixInput, int yAxis, int xAxis) {
        int total = 0;
        //^ set-up
        //: apply kernel to the element with the element's neighbors
        for (int yAxisKernel = -1; yAxisKernel < kernel.length-1; yAxisKernel++) {
            for (int xAxisKernel = -1; xAxisKernel < kernel.length-1; xAxisKernel++) {
                try { total += kernel[yAxisKernel+1][xAxisKernel+1] * matrixInput[yAxis+yAxisKernel][xAxis+xAxisKernel]; }
                catch(ArrayIndexOutOfBoundsException error) { }
                //^ works as zero-padding by not adding operations that include out-of-bounds
                //^ indexes as that will be that same as adding zero.
            }
        }
        return total;
    }
    @Override
    public int[][] deserializeIntArrArr(byte[] data){
        //* to deserialize the raw data int[][] to operate on it
        ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
        //^ holds and formats the byte data of the argument
        ObjectInputStream inStream;
        //^ 'ObjectOutputStream' serialize and write objects.
        int[][] deserialized = null;
        //^ initialized as null because of the try-statement but null should never be returned

        try {
            //^ try-statement to deal with potential IOException or ClassNotFoundException errors (mandated by compiler)
            inStream = new ObjectInputStream(byteInStream);
            //^ 'ObjectInputStream' initialized for deserialization and writing to 'byteInStream'
            deserialized = (int[][]) inStream.readObject();
            //^ deserialize the byte array argument and writes it to 'byteOutStream'.
            //^ the only code in this method that can potentially cause a 'ClassNotFoundException' error.
            inStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {  e.printStackTrace(); }
        //^ for possible errors relating to deserialization, int[][] parsing, and the 'ObjectInputStream' instance

        return deserialized;
        //^ Returns a 2d int array containing the deserialized data
    }
}
```

Normalization:

```java
//: relevant imports
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class Task3JobNormalization implements Task3Job{
    private boolean done = false;
    //^ notify main thread when the job has finished
    private int[][] unformatted;
    //^ the matrix to apply the specified kernel filter on
    private int start;
    //^ to calculate the position of the first matrix element to apply the operation on.
    private int length;
    //^ tells the nested loop how many matrix elements to apply the operation on (is calculated)
    private int[] result;
    //^ where the result of the job is stored to be later fetched by the worker node/thread

    public Task3JobNormalization(byte[] unformatted, int start, int length) {
        this.unformatted = this.deserializeIntArrArr(unformatted);
        //^ originally serialized it to prevent passing of references between threads/nodes
        this.start = start;
        this.length = length;
        this.result = new int[length];
        //^ the result is based of how big of proportion of normalization we want this job
        //^ to do itself.
    }

    @Override
    public boolean isDone()/*getter method*/{ return this.done; }

```

```java
    @Override
    public int[] perform(){
        //* to prevent the reference passed instead of the values (deep-copy)
        //:set-up
        boolean first = true;
        //^ for the stating index of the 2nd subject array and beyond
        int remaining = this.start;
        //: set up the starting index of the input (not all sub-arrays are always equal lengths)
        int yAxis = 0;
        //^ the starting index of the array of int sub-arrays - calculated by "start" and dimensions.
        //^ Assigned as zero for sake of initialization but the zero will not be read until change.
        int xAxis;
        //^ the starting index of the selected int sub-array - calculated by "start" and dimensions
        //^ (2nd dimensional index).
        int loops = this.length;
        //^ dictates how many elements left the job must operate on for its part of the normalization

        //: calculate the index of the int array of int arrays (yAxis)
        for(int index = 0; index < this.unformatted.length; index++) {
            if (remaining <= this.unformatted[index].length-1) { yAxis = index; break; }
            remaining -= (this.unformatted[index].length);
        }

        xAxis = remaining;
        //^ 2nd dimension index is based on 1st dimension (yAxis) calculation

        //: normalize part of the kernel's result based within specified range
        for (int y = yAxis; y < this.unformatted.length; y++) {
            if (loops == 0)/* thread finish its part of the matrix*/{ break; }
            if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis = 0;}
            if (first) { first = false; };
            //^ after this iteration, the current iteration no longer be the first

            for (int x = xAxis; x < this.unformatted[y].length; x++) {
                //^ going via each element in current sub-array
                loops--;
                //^ one less element to process the current job's part of the normalization
                if (!( y == this.unformatted.length || x == this.unformatted[y].length )) {
                    //^ prevent out-of-bounds error when going though the jagged array (as sub-array
                    //^ lengths are inconsistent and unknown).
                    this.result[(this.result.length-1)-loops] = this.unformatted[y][x];
                    //^ deep-copy the element's value to the array of atomic int arrays (concurrently)
                }
                else {
                    this.result[(this.result.length-1)-loops] = 0;
                    //^ We don't want resulting int[][] to also be jagged so we add placeholders as '0's
                }
```

```
                if (loops == 0){ break; }
                //^ if job has finished creating its part of the result
            }
        }

        this.done = true;
        //^ to tell the corresponding worker thread that the job has finished when it checks it
        return this.result;
        //^ stores result for the corresponding worker thread to fetch
    }
    @Override
    public int[][] deserializeIntArrArr(byte[] data){
        //^ has to be public because it is part of the 'Task3Job' interface
        //* to deserialize the raw data int[][] to operate on it
        ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
        //^ holds and formats the byte data of the argument
        ObjectInputStream inStream;
        //^ 'ObjectOutputStream' serialize and write objects.
        int[][] deserialized = null;
        //^ initialized as null because of the try-statement but null should never be returned

        try {
            //^ try-statement to deal with potential IOException or ClassNotFoundException errors (mandated by compiler)
            inStream = new ObjectInputStream(byteInStream);
            //^ 'ObjectInputStream' initialized for deserialization and writing to 'byteInStream'
            deserialized = (int[][]) inStream.readObject();
            //^ deserialize the byte array argument and writes it to 'byteOutStream'.
            //^ the only code in this method that can potentially cause a 'ClassNotFoundException' error.
            inStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {  e.printStackTrace(); }
        //^ for possible errors relating to deserialization, int[][] parsing, and the 'ObjectInputStream' instance

        return deserialized;
        //^ Returns a 2d int array containing the deserialized data
    }
}
```

And the class, with the main code, can be seen in the followings:

```java
//: all relevant imports
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Arrays;

public class Task3 {
    private static Utilities utilities = new Utilities();
    //^ for printing matrixes and generating matrixes
    private static Task3ThreadPool threadPool;
    //^ stores the thread pool instance in the main class as they must be in the same thread to reflect
    //^ the master node in the Thread-per-Node Model,
    private static int[][] matrix;
    //^ stores randomly generated matrix then used for the result of the Thread-per-Node solution
    private static int[][] goldenStandardMatrix;
    //^ stores the matrix result of the golden solution
    private static int[][] unformatted;
    //^ stores the jagged array from each kernel application

    public static void main(String[] args) {
        threadPool = new Task3ThreadPool( utilities.getProcessingEleCount() );
        //^ instantiate the thread pool with an appropriate number of sub-threads determined by user
        GoldenStandard goldenStandard = new GoldenStandard();

        Task3.matrix = Task3.utilities.matrixGen();
        //^ generate the matrix of random numbers (0-9)
        Task3.utilities.printMatrix(matrix, "Randomly generated matrix:");
        //^ prints the generated matrix (in a user-friendly manner)

        long startTime;
        //^ to store start time to time execution speeds

        System.out.println("Starting golden standard solution:");
        startTime = System.nanoTime();
        //^ get current time
        Task3.goldenStandardMatrix = goldenStandard.apply(matrix);
        //^ golden solution calculates result and gets caught into 'goldenStandardMatrix' field
        System.out.println("Finished golden standard solution.");
        System.out.println("Total time golden standard solution:");
        Task3.utilities.executionTime(startTime);
        //^ calculates and displays execution duration time based on started time
        Task3.utilities.printMatrix(goldenStandardMatrix, "Resulting matrix from the golden standard solution:");

        System.out.println("Starting compute cluster solution:");
        startTime = System.nanoTime();
        //: record initial matrix dimensions for the normalization parts
        int martix1stDimentsion = matrix.length;
        int martix2ndDimentsion = matrix[0].length;

        //: apply sharpen kernel and normalize result
        Task3.unformatted = Task3.applyKernel(Task3.matrix, 1);
        Task3.matrix = Task3.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);

        //: apply edge-detection kernel and normalize result
        Task3.unformatted = Task3.applyKernel(Task3.matrix, 2);
        Task3.matrix = Task3.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);

        Task3.poisonPill();
        //^ close all sub-threads
        //: output result and duration time
        System.out.println("Finished compute cluster solution.");
        System.out.println("Total time of compute cluster solution:");
```

```java
        Task3.utilities.executionTime(startTime);
        Task3.utilities.printMatrix(Task3.matrix, "Resulting matrix from the compute cluster solution:");

        System.out.println("Comparing both solutions:");
        System.out.println(Task3.utilities.compareMatrixies(Task3.matrix, goldenStandardMatrix));
        //^ comparison by simple subtractions
        System.out.println("Comparison done, program finished.");
    }
    private static int[] sliceIndexByWorkload(int[][] matrix, int start, int end, boolean applyKernel) {
        //* Necessary to minimize workload, instead of giving out whole copies (to nodes), as much as possible
        //* to not exceed the default maximum memory heap leap limit - especially for scaling matrix or
        //* sub-thread count (program scalability).
        //* Can work for both normalized and jagged arrays - for code integrity.
        //: all set-ups initialized to satisfy compiler but initialization should not be part of the returned.
        int startSlice = 0;
        //^ starting sub-array's index
        int endSlice = 0;
        //^ ending sub-array's index
        int startSubArrayIndex = 0;
        //^ starting index inside the range sub-arrays
        int endSubArrayIndex = 0;
        //^ ending index inside the range sub-arrays

        //: determine which sub-arrays are needed and what range inside of them of them
        endSubArrayIndex = end - start;
        //^ assume the the startSubArrayIndex is 0
        for (int index = 0; index < matrix.length; index++) {
            //* calculates both sub-array start and beginning start range index
            if (start < matrix[index].length) {
                startSubArrayIndex = start;
                startSlice = index;
                break;
            }
            start -= matrix[index].length;
        }
        endSubArrayIndex += startSubArrayIndex;
        //^ do not assume the the startSubArrayIndex is 0
        for (int index = 0; index < matrix.length; index++) {
            //* calculates just the sub-array end
            if (end < matrix[index].length) {
                endSlice = index+1;
                //^ adds 1 because array slicing does not include the ending index in the slice
                break;
            }
            end -= matrix[index].length;
        }

        if (applyKernel) {
            //^ only if applying kernels, not normalization.
            //: Make sure matrix part is surrounded with more the matrix for the kernel applications
            //: to give accurate result.
            //: extra comparisons for code integrity - in case developer scales matrix size down.
            if (startSlice != 0) {
                startSlice -= 1;
                //^ adds the values of above matrix sub-array.
                //* Considers the shift in range indexes when pushing elements at the start of the workload.
                int indexShift = matrix[startSlice].length;
                startSubArrayIndex += indexShift;
                endSubArrayIndex += indexShift;
            }
            if (endSlice != matrix.length) {
                endSlice += 1;
                //^ adds the values of above matrix sub-array
            }
        }
```

```java

        return new int[] {startSlice, endSlice, startSubArrayIndex, endSubArrayIndex};
    }
    private static int[][] applyKernel(int[][] matrix, int kernelType){
        int threadCount = Task3.threadPool.threadCount();
        //^ ask thread pool how many sub-threads is it currently handling
        int totalSize = matrix.length * matrix[0].length;
        //^ get element count to calculate distributed workload size
        int sectorSize = totalSize / threadCount;
        //^ calculate distributed workload size
        int[] args;

        //: For code integrity purposes - if another developer changes initial matrix size to a very small size (STILL MUST BE ATLEAST 1x1).
        if (sectorSize == 0) {
            //^ happens when there are more sub-threads than initial matrix element count
            sectorSize = 1;
            //^ distribute smallest possible workload
            threadCount = totalSize;
            //^ to not bother giving redundant jobs to the extra sub-threads
        }

        //: determines parameter arguments for queued jobs - considers odd elements out from workload divisions and certain sub-thread counts
        for (int index = 0; index < threadCount-1; index++) {
            //* fills the rest of the thread pool and gives the data to the rest of the threads to do the jobs
            args = sliceIndexByWorkload(matrix, index*sectorSize, ((index+1)*sectorSize)-1, true);
            Task3.threadPool.enqueueJob( new Task3JobKernelApplication(kernelType,Task3.serializeIntArrArr(Arrays.copyOfRange(matrix,args[0],args[1])),args[2],args[3]) );
        }
        if (threadCount > 1) {
            //* to account for the remainder matrix elements after division by number of sub-threads.
            args = sliceIndexByWorkload(matrix, (threadCount-1)*sectorSize,totalSize-1, true);
            Task3.threadPool.enqueueJob( new Task3JobKernelApplication(kernelType,Task3.serializeIntArrArr(Arrays.copyOfRange(matrix,args[0],args[1])),args[2],args[3]) );
        }
        if (threadCount == 1) {
            //* if user/developer wants to use only one sub-thread, for testing or comparison reasons.
            Task3.threadPool.enqueueJob( new Task3JobKernelApplication(kernelType,Task3.serializeIntArrArr(matrix),0,totalSize-1) );
        }

        Task3.threadPool.wakeUp();
        //^ activate thread pool to let sub-threads fetch jobs

        int[][] result = Arrays.copyOfRange(Task3.deserializeIntArrArr(Task3.threadPool.getResults()),0,threadCount);
        //^ the array slicing is for the edge-detection kernel application as results from last thread pool wake-up will be brought along
        //^ in accordance to the 'code integrity purposes' mentioned in this method.

        //: for-loop is for the sharpen kernel application as results will include nulls in accordance to the 'code integrity purposes' mentioned in this method.
        for (int index = 0; index < result.length; index++) {
            //* For code integrity purposes (same as the previous integrity purposes)
            if (result[index]==null) { return Arrays.copyOfRange(result, 0, index); }
        }

        threadPool.clear();
        //^ remove all now-redundant stored data from thread pool to minimise program's heap memory usage
        return result;
        //^ returns the result as an jagged int array
    }
```

```java
    private static int[][] normalization(int[][] unformatted, int subArraysCount, int subArraysLength){
        int threadCount = Task3.threadPool.threadCount();
        //^ ask thread pool how many sub-threads is it currently handling
        int[][] matrix = new int[subArraysCount][subArraysLength];
        //^ creates new int[][] for the result with dimensions based on the original matrix
        int[][] preMatrix =  new int[threadCount][];
        //^ where the jagged int array
        int totalSize = matrix.length * matrix[0].length;
        //^ get element count to calculate distributed workload size
        int sectorSize = subArraysLength;
        //^ better renaming for method;s body for easier code development and maintenance
        int sectorCount = subArraysCount / threadCount;
        //^ calculate how many sub-arrays (of the jagged array) each thread can handle
        int sectorCountRemainder = subArraysCount % threadCount;
        //^ remainder from "sectorCount" calculation
        int distributedLoadLength;
        //^ stores the length (element count) distributed load
        int matrixIndex;
        //^ index of the to-be-returned result matrix
        int activeThreads = threadCount;
        //^ better renaming for method's body for easier code development and maintenance
        int[] args;

        distributedLoadLength = sectorCount * sectorSize;
        //^ Assume there is no remainder in the "sectorCount" calculation

        //: increase workload if the "distributedLoadLength" assumption was wrong
        if (sectorCountRemainder != 0) {
            sectorCount++;
            distributedLoadLength = sectorCount * sectorSize;
            activeThreads = (totalSize / distributedLoadLength)+1;
        }

        args = sliceIndexByWorkload(unformatted, 0, distributedLoadLength, false);
        if (threadCount == 1){ args[1] = 1; }
        //^ if only one sub-thread was used
        Task3.threadPool.enqueueJob( new Task3JobNormalization(Task3.serializeIntArrArr(Arrays.copyOfRange(unformatted,0,args[1])),args[2],distributedLoadLength) );
        //^ serialize queue the first job (the beginning)

        for (int index = 1; index < activeThreads; index++) {
            //* serialize queue the other jobs (to the end)
            args = sliceIndexByWorkload(unformatted, index*distributedLoadLength, (index+1)*distributedLoadLength, false);
            //Task4.threadPool.enqueueJob( new Task4JobNormalization(Task4.serializeIntArrArr(unformatted),(index*distributedLoadLength),distributedLoadLength) );
            if (args[1] == 0) { args[1] = unformatted.length; }
            //^ for the last job
            Task3.threadPool.enqueueJob( new Task3JobNormalization(Task3.serializeIntArrArr(Arrays.copyOfRange(unformatted,args[0],args[1])),args[2],distributedLoadLength) );
        }

        //: wake up thread pool and get the sub-threads collective results
        threadPool.wakeUp();
        preMatrix = Task3.deserializeIntArrArr(threadPool.getResults());

        //: translate the result to the matrix due to the place-holderelements
        matrixIndex = 0;
        for (int preMatrixIndex = 0; preMatrixIndex < preMatrix.length; preMatrixIndex++) {
            boolean doneAllElements = false;
            for (int jobResultIndex = 0; jobResultIndex < preMatrix[preMatrixIndex].length; jobResultIndex += sectorSize ) {
                matrix[matrixIndex] = Arrays.copyOfRange( preMatrix[preMatrixIndex], jobResultIndex, jobResultIndex+sectorSize );
                //^ get all elements for one row of the resulting matrix
                matrixIndex++;
                //^ next row
                if ((matrixIndex*subArraysLength) >= totalSize) { doneAllElements = true; break; }
                //^ ignore the place-holders
            }
            if (doneAllElements) { break; }
```

```java
        }

        threadPool.clear();
        //^ remove all now-redundant stored data from thread pool to minimise program's heap memory usage
        return matrix;
        //^ return as the kernel application result normalized
    }
    private static void poisonPill() {
        int threadCount = Task3.threadPool.threadCount();
        for (int index = 0; index < threadCount; index++) {
            //* fills the thread pool with Poison Pills to close all threads
            Task3.threadPool.enqueueJob( new Task3JobPoisonPill() );
        }
        Task3.threadPool.wakeUp();
        //^ make thread pool send poison pill (as jobs) to the sub-theads to execute
    }
    private static byte[] serializeIntArrArr(int[][] intArrArr){
        //* To send serialized copies so referencing between nodes are limited to twice per communication/message
        ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
        //^ holds and formats the byte data of the argument
        ObjectOutputStream outStream = null;
        //^ 'ObjectOutputStream' serialize and write objects.
        //^ Initialized as null because of the try-statement but null should never used

        try {
            //^ try-statement to deal with possible IOException errors (mandated by compiler)
            outStream = new ObjectOutputStream(byteOutStream);
            //^ 'ObjectOutputStream' initialized for serialization and writing to 'byteOutStream'
            outStream.writeObject(intArrArr);
            //^ serialize the argument and writes it to 'byteOutStream'
            outStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException e) { e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
        return byteOutStream.toByteArray();
        //^ Returns a byte array containing the serialized data
    }
    private static int[][] deserializeIntArrArr(byte[] data){
        //* no 200ms network delay in this method's calls because 'main' and
        //* the thread pool are both in the same main thread
        ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
        //^ holds and formats the byte data of the argument
        ObjectInputStream inStream;
        //^ 'ObjectOutputStream' serialize and write objects.
        int[][] deserialized = null;
        //^ initialized as null because of the try-statement but null should never be returned

        try {
            //^ try-statement to deal with potential IOException or ClassNotFoundException errors (mandated by compiler)
            inStream = new ObjectInputStream(byteInStream);
            // 'ObjectInputStream' initialized for deserialization and writing to 'byteInStream'
            deserialized = (int[][]) inStream.readObject();
            //^ deserialize the byte array argument and writes it to 'byteOutStream'.
            //^ The only code in this method that can potentially cause a 'ClassNotFoundException' error.
            inStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {  e.printStackTrace(); }
        //^ for possible errors relating to deserialization, int[][] parsing, and the 'ObjectInputStream' instance
        return deserialized;
        //^ Returns a 2d int array containing the deserialized data
    }
}
```

Thread safety

As this reflects a computer network, the Poison Pill has been made as an actual job (to prevent zombie threads after programs completion) as seen below:

```
public class Task3JobPoisonPill implements Task3Job{
    //^ implements 'Task3Job' so it can be stored in the thread pool's
    //^ FIFO job queue.
    private boolean done = false;
    //^ simple set-up

    @Override
    public boolean isDone() { return done; }
    //^ returns true when '.perform()' gets executed, otherwise return false

    @Override
    public int[] perform() { this.done = true; return new int[] { }; }
    //^ when sub-thread catches result of job, it breaks while-loop and
    //^ closes thread if result is an empty int array.
    //^ It returns int[] as the other jobs returns int[] as well, hence
    //^ why so does the method in the 'Task3Job' interface.

    @Override
    public int[][] deserializeIntArrArr(byte[] data) { return null; }
    //^ this method will never be called but is there because deserialization
    //^ is one of the main parts of a job (hence is in the job interface),
    //^ deserialized data to produce results, the Poison Pill enters sub-thread
    //^ but unlike jobs to process as a job to stop the sub-thread's while-loop
    //^ (inside 'start' method) to close the sub-threads otherwise program will
    //^ have zombie threads.
}
```

As the Shared Memory Model is no longer used (and no 'pass by reference' between threads/nodes), there is no shared memory/fields between threads hence removing the risk of concurrent data/fields/variables problems such as race conditions (Delporte-Gallet et al., 2003).

The "Thread.sleep(1)" deals with preventing busy-waiting where busy-waiting is the continuous polling what can negatively impact other nodes/threads running in the simulated network.

Code's output

Like task 2, code's output is done for 10x10 array for the same reasons in the following:

```
Java runtime have 11 available effective processing elements (minus the main thread)
How many processing elements do you want for this operation?
3
Human counterpart compliant
Randomly generated matrix:
[5, 8, 0, 5, 9, 3, 1, 9, 6, 3]
[6, 6, 9, 5, 8, 2, 0, 6, 0, 6]
[1, 7, 5, 8, 5, 1, 5, 2, 5, 7]
[5, 7, 6, 4, 6, 8, 4, 7, 9, 1]
[6, 5, 0, 1, 4, 0, 3, 6, 3, 4]
[9, 2, 8, 0, 1, 9, 8, 9, 1, 6]
[0, 2, 7, 4, 5, 2, 8, 4, 1, 5]
[4, 5, 1, 6, 8, 8, 9, 7, 4, 4]
[7, 1, 3, 5, 3, 8, 0, 8, 7, 9]
[7, 7, 5, 5, 0, 6, 0, 5, 4, 3]
# END MATRIX PRINT #
Starting golden standard solution:
Finished golden standard solution.
Total time golden standard solution:
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
1178800, 1, 0
Resulting matrix from the golden standard solution:
[41, 196, -240, 38, 206, -1, -94, 263, 93, 9]
[101, -63, 187, -124, 108, -51, -147, 134, -293, 132]
[-161, 62, -126, 106, -39, -178, 129, -151, 2, 166]
[52, 72, 82, -38, 51, 236, -38, 74, 203, -184]
[32, -2, -195, -42, 79, -268, -119, 9, -118, 65]
[308, -182, 271, -151, -101, 282, 55, 208, -171, 164]
[-164, -100, 146, -32, 48, -272, 45, -104, -126, 112]
[60, 134, -169, 57, 102, 65, 142, 48, -66, -31]
[133, -224, -27, 42, -145, 165, -313, 143, -5, 228]
[140, 136, 49, 119, -174, 209, -142, 104, -37, -30]
# END MATRIX PRINT #
Starting compute cluster solution:
Finished compute cluster solution.
Total time of compute cluster solution:
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
3880437400, 3880, 3
Resulting matrix from the compute cluster solution:
[41, 196, -240, 38, 206, -1, -94, 263, 93, 9]
[101, -63, 187, -124, 108, -51, -147, 134, -293, 132]
[-161, 62, -126, 106, -39, -178, 129, -151, 2, 166]
[52, 72, 82, -38, 51, 236, -38, 74, 203, -184]
[32, -2, -195, -42, 79, -268, -119, 9, -118, 65]
[308, -182, 271, -151, -101, 282, 55, 208, -171, 164]
[-164, -100, 146, -32, 48, -272, 45, -104, -126, 112]
[60, 134, -169, 57, 102, 65, 142, 48, -66, -31]
[133, -224, -27, 42, -145, 165, -313, 143, -5, 228]
[140, 136, 49, 119, -174, 209, -142, 104, -37, -30]
# END MATRIX PRINT #
Comparing both solutions:
subtraction proof (a simple subtraction between both matrixes):
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
# END MATRIX PRINT #
both matrixes are identical in both: dimensions and elements' value
Comparison done, program finished.
```

Speed

The following table shows the speeds:

(seconds to the lower whole integer)	golden standard	1 sub-thread	2 sub-threads	3 sub-threads	4 sub-threads	5 sub-threads	6 sub-threads	7 sub-threads	8 sub-threads
trail #1	4	15	13	14	17	19	21	25	27
trail #2	4	13	13	14	17	20	22	25	28
trail #3	4	13	13	15	17	20	21	26	28
trail #4	4	13	13	16	17	19	22	26	28
trail #5	4	13	13	15	17	19	21	24	28
mean	4	13.4	13	14.8	17	19.4	21.4	25.2	27.8

Note: Due to magnitude of execution duration, time is measured in seconds instead of milliseconds. Seconds are rounded down because that is how Java converts nano seconds to seconds.

As observed, they are slower than task 1 and 2 due to the limitations of the Thread-per-Node/ Model restrictions such as the frequency of serializing and reserializing huge data threads (Opyrchal and Prakash, 1999) and 200 millisecond delay per MPI message sent between nodes. This only increases with the sub-thread count, but the time saving with multi-threading still applies because the increase in execution time to cub-thread/node count is not linear.

Task 4

Defining node failure

The author is instructed for sub-threads to have a passed probability of death before returning the results. Due using the Task 3 (and Task 4) thread pool model, the live-or-die calculation is done after every job performed but before the sub-thread fetches the result. Apart from name changes (“Task3” to “Task4”) and code for failures, the code will be similar to that of Task 3 so only relevant code changes will be shown. This means stuff, such as thread safety, is based on what is discussed in Task 3.

Failure rate is determined by user as seen:

```
    private static int failureRate;
    //^ stores failure rate in the main "Task4" class to be passed to the worker node constructor
    //^ as a parameter/argument.

    public static void main(String[] args) {
        failureRate = Task4.inputFailureRate(utilities.scanner);
        //^ get failure rate from user.
        //^ Cannot open 2 scanners (even with "scanner.close()") so had to use the utilities scanner.
        threadPool = new Task4ThreadPool( utilities.getProcessingEleCount(), failureRate );
        //^ instantiate the thread pool with an appropriate number of sub-threads determined by user
```

Method of getting failure rate is as seen:

```
    private static int inputFailureRate() {
        //* private method of the main Task4 class.
        //* Code taken and modified from "Utilities.getProcessingEleCount()".
        //: set-up
        Scanner scanner = new Scanner(System.in);
        boolean validCount = false;
        //^ stores the validity of users input
        int userInput = 0;
        //^ '0' is a placeholder, but assume user first meant 0% failure rate

        while(!validCount) {
            //* gets a valid input (0-100 as percentage) from user
            System.out.println("Enter failure rate for sub-threads before they return results (in percentage between 0-100)");
            if (!scanner.hasNextInt()) {
                System.out.println("Error encounted on human counterpart: invalid input - input must be an integer");
                continue;
            }
            userInput = scanner.nextInt();
            if( userInput < 1) {
                //* below percentage range
                System.out.println("Error encounted on human counterpart: invalid input - (integer) input must be positive");
                continue;
            }
            if( userInput > 100) {
                //* above percentage range
                System.out.println("Error encounted on human counterpart: invalid input - connot be above 100%");
                continue;
            }
            //: success
            System.out.println("Human counterpart compliant");
            System.out.println( String.format("Failure rate is %d%", userInput) );
            validCount = true;
        }

        //: success
        scanner.close();
        return userInput;
    }
}
```

Thread pool's passing of the failure rate as seen:

```
    public Task4ThreadPool(int threadCount, int failureRate) {
        //: set-up the fields
        this.sleep = false;
        //^ activate the thread pool
        jobQueue = new ArrayList<>();
        //^ for good practice and scalability - the data structure has dynamic size
        threadJobResults = new int[threadCount][];
        //^ is a jagged array, because jobs' returned int arrays' length are not predefined

        //: set-up and fill the thread pool
        this.threads = new Task4WorkerNode[threadCount];
        for (int index = 0; index < threadCount; index++) {
            this.threads[index] = new Task4WorkerNode(failureRate, index);
            //^ worker thread instance, parameters are both primitives so no references
            //^ are passed cross-node/thread.
            this.threads[index].start();
            //^ start the thread's execution
        }
    }
```

Thread's live-or-die set-up and calculation as seen:

```java
    private int failureRate;
    //^ the probability of thread dying before fetching job's result
    private int threadPoolIndex;
    //^ let the user know which node/thread died when it does.

    public Task4WorkerNode(int failureRate, int threadPoolIndex) {
        //* constructor for the failure percentage change.
        //* Parameters are primitives so no references are passed between nodes.
        this.failureRate = failureRate;
        this.threadPoolIndex = threadPoolIndex;
    }

    private boolean nodeDeath() {
        //* calculation to decide if the node/thread was to live or die
        Random rand = new Random();
        boolean death = rand.nextInt(100) < this.failureRate;
        //^ based on percentage chance, does node/thread dies?
        if (death) {
            //* node/thread dies
            long threadId = Thread.currentThread().getId();
            //^ does not interfere with node's job to actually simulate the node's death - not realising that it is dead.
            System.out.println(String.format("A node/thread has died (thread pool index: %d, thread ID: %d) - unknown to the thread pool", this.threadPoolIndex, threadId));
            //^ thread pool index is more relevant than thread id because index correlate to what workload the specific sub-thread is handling
            //^ but still including both in case if user wants both numbers.
            //^ This tells user that sub-thread died but thread pool does not know that yet.
        }
        return death;
        //^ will die?
    }

    public void run() {
        this.running = true;
        //^ dictates if the thread should continue running or end
        while (this.running) {
            //* while-loop runs constantly till thread must end

            if (Thread.interrupted()) { break; }
            //^ ending ".run()", when interrupted, to end thread by death.
            //^ Does not toggle "this.running" because node/thread cannot do
            //^ anything when it dies.

            if (job != null && !this.jobDone) {
                //^ execute when there is an existing job that is not completed

                int[] jobResultCache = this.job.perform();
                //^ where the job actually gets done but caught result is stored
                //^ in-scope until the thread survives the death probability.
                if (!this.nodeDeath()) { jobResult = jobResultCache; }
                //^ if thread does not die (by % probability), then store result
                //^ where it can be fetched by the thread pool and continue as normal.
                else { this.interrupt(); continue; }
                //^ if thread dies then end ".run()" execution by interruption when it gets checked

```

Findings and fixes

Extra print statements are added to help locate abnormalities such as seen:

```java
        //: apply sharpen kernel and normalize result
        Task4.unformatted = Task4.applyKernel(Task4.matrix, 1);
        System.out.println("compute cluster solution - Sharpen kernel application applied");
        Task4.matrix = Task4.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);
        System.out.println("compute cluster solution - applied matrix normalized (1 of 2)");

        //: apply edge-detection kernel and normalize result
        Task4.unformatted = Task4.applyKernel(Task4.matrix, 2);
        System.out.println("compute cluster solution - Edge-detection kernel application applied");
        Task4.matrix = Task4.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);
        System.out.println("compute cluster solution - applied matrix normalized (2 of 2)");

        Task4.poisonPill();
        //^ close all sub-threads
        System.out.println("compute cluster solution - Poison Pills administered");
        //: output result and duration time
        System.out.println("Total time of compute cluster solution:");
        Task4.utilities.executionTime(startTime);
```

As well as disabling the (10000x10000) matrix prints to fit the test outputs in the document.

Without error handling

```
Enter failure rate for sub-threads before they return results (in percentage between 0-100)
10
Human counterpart compliant
Failure rate is 10 percent
Java runtime have 11 available effective processing elements (minus the main thread)
How many processing elements do you want for this operation?
3
Human counterpart compliant
Starting golden standard solution:
Finished golden standard solution.
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
4285570300, 4285, 4
Starting compute cluster solution:
compute cluster solution - Sharpen kernel application applied
compute cluster solution - applied matrix normalized (1 of 2)
compute cluster solution - Edge-detection kernel application applied
A node/thread has died (thread pool index: 2, thread ID: 23) - unknown to the thread pool
```

When a thread dies the program continues to run indefinitely without result. This is because the thread pool is continuously trying to fetch a result from the dead thread, but it cannot.

Error handling fix

Node failure

A fix could be checking if threads is alive after one returns a result, but the worst must be assumed – all sub-threads failing. A call to all threads, after a set time, can work but the time taken to solve the workload is not fully predictable (especially if to scale the program). Nodes do not know when they died, so a periodic time call may be the best (where a sub-thread is pronounced thread when failing the periodic call many times) as it may slow down the execution speed a bit, but it takes the worst-case scenarios into consideration (Lausdahl et al., 2010).

To take scalability into account we need a maths function for thread count much higher than what is currently used. To increase the accuracy of the equation, the values for 9,10 and 11 sub-threads were accounted for, as seen, as the testing workstation can support up to 12 total threads:

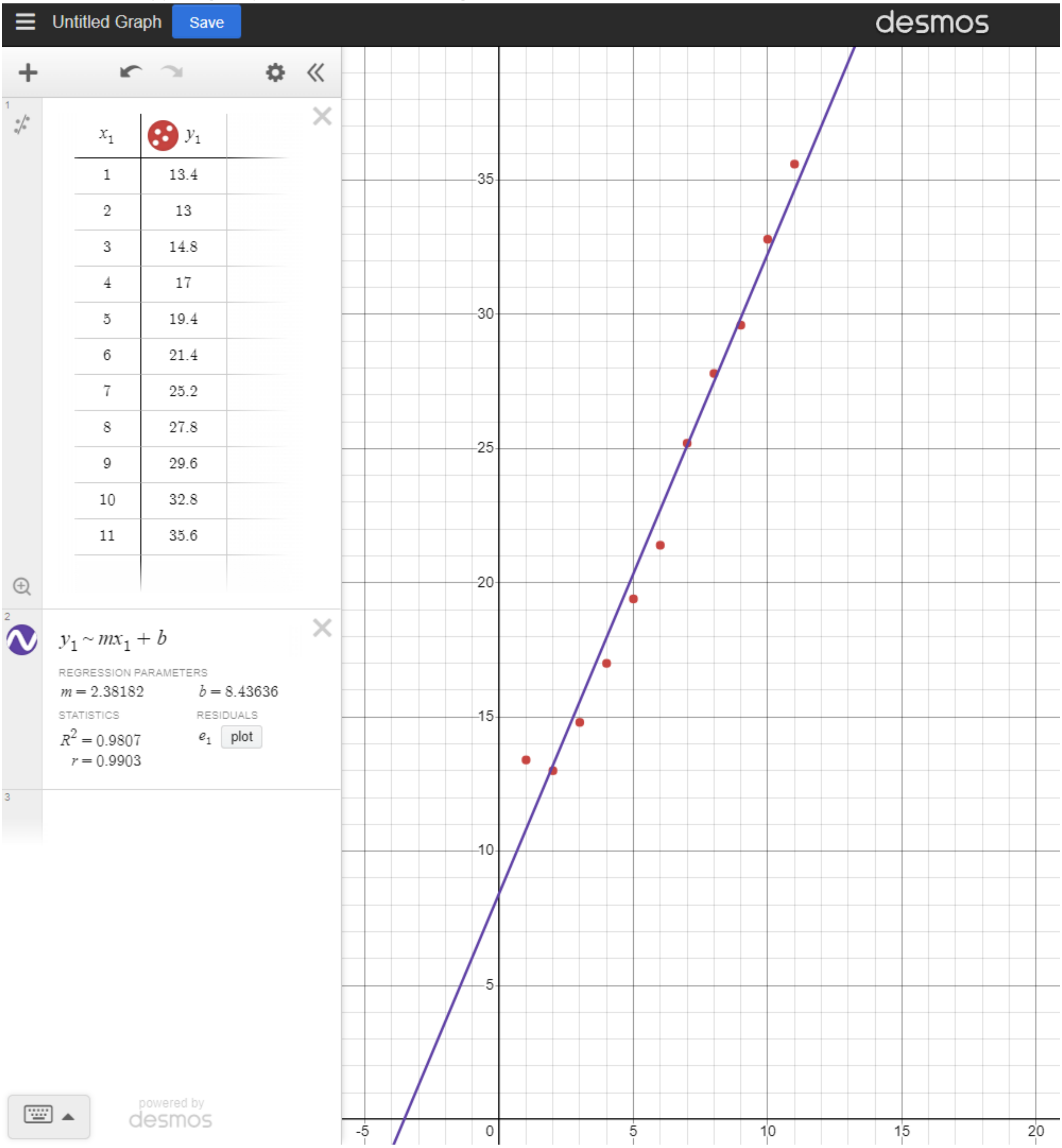
(seconds to the lower whole integer)	9 sub-threads	10 sub-threads	11 sub-threads
trail #1	30	33	34
trail #2	29	32	36
trail #3	30	33	36
trail #4	29	33	36
trail #5	30	33	36
mean	29.6	32.8	35.6

Note: Tested CPU can only support up to 12 total threads effectively – so only up to 11 sub-threads (with the main thread) could be used.

According to Task 3 speeds and the table above, the predicted mathematical equation of correlation between execution time and thread count is (to 2 d.p.):

$$y = 2.38x + 8.44$$

This was calculated by plotting the speed tables' values and finding the "best line of fit" as seen:



Screenshot and calculation from https://www.desmos.com/calculator

As computers process integers much faster than floats, the numbers in the equation have been rounded up:

$$y = 3x + 9$$

Where: y = predicted processing time, x = sub-thread count.

The round up takes other programs' interruptions into account.

After much testing, each part of the solution takes around a quarter of the time (both kernel applications and both normalizations), meanwhile Poison Pills only takes around 600ms (regardless of thread count). These findings will be accounted for in the periodic calling.

Example of measuring each part with 11 sub-threads:

```java
        //: apply sharpen kernel and normalize result
        startTime = System.nanoTime();
        Task3.unformatted = Task3.applyKernel(Task3.matrix, 1);
        Task3.utilities.executionTime(startTime);
        startTime = System.nanoTime();
        Task3.matrix = Task3.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);
        Task3.utilities.executionTime(startTime);

        //: apply edge-detection kernel and normalize result
        startTime = System.nanoTime();
        Task3.unformatted = Task3.applyKernel(Task3.matrix, 2);
        Task3.utilities.executionTime(startTime);
        startTime = System.nanoTime();
        Task3.matrix = Task3.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);
        Task3.utilities.executionTime(startTime);
        startTime = System.nanoTime();

        Task3.poisonPill();
        Task3.utilities.executionTime(startTime);
        //^ close all sub-threads
        //: output result and duration time
        System.out.println("Total time of compute cluster solution:");
        Task3.utilities.executionTime(startTime2);
        //Task3.utilities.printMatrix(Task3.matrix, "Resulting matrix from the compute cluster solution:");

        System.out.println("Comparing both solutions:");
        System.out.println(Task3.utilities.compareMatrixies(Task3.matrix, goldenStandardMatrix));
        //^ comparison by simple subtractions
        System.out.println("Comparison done, program finished.");
```

```
Console
<terminated> Task3 [Java Application] C:\Program Files\Java\jdk-22\bin\javaw.exe (Jan 2, 2025, 12:45:37 PM – 12:46:18 PM) [pid: 6832]
Java runtime have 11 available effective processing elements (minus the main thread)
How many processing elements do you want for this operation?
11
Human counterpart compliant
Starting golden standard solution:
Finished golden standard solution.
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
4226079300, 4226, 4
Starting compute cluster solution:
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
9137876900, 9137, 9
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
8647034000, 8647, 8
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
8256288000, 8256, 8
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
8242678600, 8242, 8
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
624861000, 624, 0
Total time of compute cluster solution:
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
34913133000, 34913, 34
Comparing both solutions:
both matrixes are identical in both: dimensions and elements' value
Comparison done, program finished.
```

The periodic call and check can be seen in the thread pool as seen:

```java
    private void requestResults() {
        this.nodeInactivityCount = new int[this.threads.length];
        boolean allResults = false;
        //^ dictates if it thread pool should keep waiting for all results or not
        boolean[] alldone = new boolean[this.maxJobs];
        //^ only fetch results from the relevant threads to save processing time
        while (!allResults) {
            //* far faster than the alternative of constantly checking a sub-thread job until it
            //* is done one at a time.
            for (int index = 0; index < this.maxJobs; index++) {
                //* get all results to if each thread has finished the job or not
                alldone[index] = threads[index].isJobDone();
                //^ store the each job status
            }
            try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
            //^ to impose a delay of 200ms for each computational job executed to simulate the
            //^ speeds of modern computers and cluster networks.
            //^ These node-to-node messages are to check if the all sub-thread has finished
            //^ their given jobs.
            allResults = true;
            //^ Assume that all threads have finished
            for (int index = 0; index < this.maxJobs; index++) {
                //* only gathers threads that were given relevant jobs
                if(alldone[index] == false) {
                    //^ current thread hasn't finished?
                    allResults = false;
                    //^ if one thread isn't finished then all threads are not finished!

                    //: check for dead threads by multiple periodic calls.
                    //: Assume that the main thread never dies.
                    nodeInactivityCount[index]++;
                    //^ sub-thread has not given result yet
                    if (nodeInactivityCount[index]*200 > this.predictedCompletion) {
                        //^ comparison is milliseconds.
                        //^ Allow sub-thread to take up to double predicted duration to takes
                        //^ int division and older/slower computers that run this program into account.
                        //* If sub-thread takes too long, assume it died, then recreate it with the missing work.
                        this.threads[index] = null;
                        this.threads[index] = new Task4WorkerNode(failureRate, index);
                        //^ make new thread in the dead thread's place in the thread pool
                        this.threads[index].setJob(this.serializeJob(this.jobHistory[index]));
                        //^ assign missing work based on job assignment history
                        try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
                        //^ to impose a delay of 200ms for another computational job executed to simulate the
                        //^ speeds of modern computers and cluster networks.
                        //^ Sent to the node in place of the one that was detected as dead.
                        System.out.println(String.format("Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: %d)", index));
                        this.threads[index].start();
                        //^ Tell the new sub-thread to start executing jobs.
                        try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
                        //^ to impose a delay of 200ms when telling the node to execute the job.
                        //^ To simulate the speeds of modern computers and cluster networks.
                        nodeInactivityCount[index] = 0;
                        //^ reset inactivity counter as thread is brand new
                    }

                }
            }
        }
        for (int index = 0; index < this.maxJobs; index++) {
            //* gather results of all relevant threads
            threadJobResults[index] = this.deserializeIntArr(this.threads[index].requestResult());
            //^ deserialize raw data MPI message into the fetched int[] result to be stored
        }
        try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
```

When testing, sometimes the program will sometimes run into a “OutOfMemoryError” after recreating too many nodes/threads as seen:

```
Enter failure rate for sub-threads before they return results (in percentage between 0-100)
50
Human counterpart compliant
Failure rate is 50 percent
Java runtime have 11 available effective processing elements (minus the main thread)
How many processing elements do you want for this operation?
8
Human counterpart compliant
Starting golden standard solution:
Finished golden standard solution.
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
4463087000, 4463, 4
Starting compute cluster solution:
A node/thread has died (thread pool index: 5, thread ID: 26) - unknown to the thread pool
A node/thread has died (thread pool index: 6, thread ID: 27) - unknown to the thread pool
A node/thread has died (thread pool index: 7, thread ID: 28) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 5)
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 6)
A node/thread has died (thread pool index: 6, thread ID: 41) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 7)
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 6)
A node/thread has died (thread pool index: 6, thread ID: 43) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 6)
=== compute cluster solution - Sharpen kernel application applied ===
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.base/java.util.Arrays.copyOf(Arrays.java:3540)
        at java.base/java.io.ByteArrayOutputStream.toByteArray(ByteArrayOutputStream.java:187)
        at Task4.serializeIntArrArr(Task4.java:221)
        at Task4.normalization(Task4.java:167)
        at Task4.main(Task4.java:61)
```

In this test, 8 sub-threads were used with a 50% fail rate. 8 sub-threads worked fine in the Task 3 version – meaning either there must be a memory leak or a process exceeded the maximum heap space given by Java.

The error originated from “byteOutStream.toByteArray();” in the “serializeIntArrArr(int[][] intArrArr)” during the first matrix normalization part, but not on the first “serializeIntArrArr(int[][] intArrArr)” call where previous calls’ int[][] are similar sizes (either 8 int[] of 12500000 or 10000 int[] of 10000 elements which are the same numbers of elements). This means that so many 100 million element 2-dimensional arrays deep-copies are stored that it exceeds the Java’s default heap memory limit.

This has fixed by modifying the thread pool “clear()” method (called by Task4 main code per part) to clear any big data (as soon as made redundant), including the Task 4 additions such as “jobHistory” which held a copy of each job (with workload) to keep the current heap memory usage as low as possible as seen:

```java
    public void clear() /*reset method*/ {
        //* not exactly aligned with the Thread-per-Node Model but the program must not exceed the Java
        //* default maximum heap memory limit, which cannot be changed in-program.
        //* The lower it is, the better the program's scalability.
        //: these data are, at this point of time, redundant so they must be deleted
        this.threadJobResults = new int[this.threads.length][];
        this.jobHistory = null;
        for (int index = 0; index < this.threads.length; index++) { threads[index].clear(); }
        //^ clears each worker node's stored result
        try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
        //^ to impose a delay of 200ms to simulate the speeds of modern computers and cluster networks.
        //^ Tells all threads to delete their now redundant stored result.
    }
```

Because this helps with Tasks 3’s scalability even further, its code and speed results were changed, equations recalculated, etc.

This helps as seen with testing 11 sub-threads with 20% failure rate:

```
Enter failure rate for sub-threads before they return results (in percentage between 0-100)
20
Human counterpart compliant
Failure rate is 20 percent
Java runtime have 11 available effective processing elements (minus the main thread)
How many processing elements do you want for this operation?
11
Human counterpart compliant
Starting golden standard solution:
Finished golden standard solution.
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
4334134600, 4334, 4
Starting compute cluster solution:
A node/thread has died (thread pool index: 8, thread ID: 29) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 8)
=== compute cluster solution - Sharpen kernel application applied ===
A node/thread has died (thread pool index: 1, thread ID: 22) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 1)
=== compute cluster solution - applied matrix normalized (1 of 2) ===
A node/thread has died (thread pool index: 2, thread ID: 23) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 2)
A node/thread has died (thread pool index: 2, thread ID: 45) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 2)
A node/thread has died (thread pool index: 2, thread ID: 46) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 2)
=== compute cluster solution - Edge-detection kernel application applied ===
A node/thread has died (thread pool index: 1, thread ID: 44) - unknown to the thread pool
A node/thread has died (thread pool index: 2, thread ID: 47) - unknown to the thread pool
A node/thread has died (thread pool index: 3, thread ID: 24) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 1)
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 2)
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 3)
=== compute cluster solution - applied matrix normalized (2 of 2) ===
A node/thread has died (thread pool index: 2, thread ID: 49) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 2)
compute cluster solution - Poison Pills administered
Total time of compute cluster solution:
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
159924871000, 159924, 159
Comparing both solutions:
both matrixes are identical in both: dimensions and elements' value
Comparison done, program finished.
```

Program is fully capable of doing more sub-threads, but the CPU used only supports up to 12 threads (including main thread).

Here is a program run with 11 sub-threads and 95% failure rate, but node death messages are omitted to fit the output in this report as seen:

```
Enter failure rate for sub-threads before they return results (in percentage between 0-100)
95
Human counterpart compliant
Failure rate is 95 percent
Java runtime have 11 available effective processing elements (minus the main thread)
How many processing elements do you want for this operation?
11
Human counterpart compliant
Starting golden standard solution:
Finished golden standard solution.
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
4141353300, 4141, 4
Starting compute cluster solution:
=== compute cluster solution - Sharpen kernel application applied ===
=== compute cluster solution - applied matrix normalized (1 of 2) ===
=== compute cluster solution - Edge-detection kernel application applied ===
=== compute cluster solution - applied matrix normalized (2 of 2) ===
compute cluster solution - Poison Pills administered
Total time of compute cluster solution:
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
6836648018100, 6836648, 6836
Comparing both solutions:
both matrixes are identical in both: dimensions and elements' value
Comparison done, program finished.
```

Corrupted data

In a node cluster, node-to-node messages can get fully corrupted (Shafi et al., 2009) by accidental or malicious interference – unable to send the job. However, if only slightly corrupted, the job can still be done but with the wrong result – making the final main node result incorrect. To combat this, the result can be compared with the golden standard and if they are not identical then the whole process happens again as seen:

```java
        System.out.println("Starting compute cluster solution:");
        startTime = System.nanoTime();
        while (true) {
            //* To simulate possible MPI message corruptions disrupting the final error.
            //: Record initial matrix dimensions for the normalization parts
            int martix1stDimentsion = matrix.length;
            int martix2ndDimentsion = matrix[0].length;

            //: apply sharpen kernel and normalize result
            Task4.unformatted = Task4.applyKernel(Task4.matrix, 1);
            System.out.println("=== compute cluster solution - Sharpen kernel application applied ===");
            Task4.matrix = Task4.normalization(Task4.unformatted, martix1stDimentsion, martix2ndDimentsion);
            System.out.println("=== compute cluster solution - applied matrix normalized (1 of 2) ===");

            //: apply edge-detection kernel and normalize result
            Task4.unformatted = Task4.applyKernel(Task4.matrix, 2);
            System.out.println("=== compute cluster solution - Edge-detection kernel application applied ===");
            Task4.matrix = Task4.normalization(Task4.unformatted, martix1stDimentsion, martix2ndDimentsion);
            System.out.println("=== compute cluster solution - applied matrix normalized (2 of 2) ===");

            Task4.poisonPill();
            //^ close all sub-threads
            System.out.println("compute cluster solution - Poison Pills administered");
            //: output result and duration time
            System.out.println("Total time of compute cluster solution:");
            Task4.utilities.executionTime(startTime);
            Task4.utilities.printMatrix(Task4.matrix, "Resulting matrix from the compute cluster solution:");

            System.out.println("Comparing both solutions:");
            String comparison = Task4.utilities.compareMatrixies(Task4.matrix, goldenStandardMatrix);
            //^ comparison by simple subtractions
            System.out.println(comparison);
            if (comparison == "both matrixes are identical in both: dimensions and elements' value") { break; }

            System.out.println("Redoing solution!");
        }
        System.out.println("Comparison successful, program finished.");
    }
```

The author also noted the jobs finishing much earlier than double predicted time so now job must complete before completed time to not be considered dead instead of double predicted time but still have a time gap for program interruptions, older hardware, and scalability as seen:

```
nodeInactivityCount[index]++;
//^ sub-thread has not given result yet
if (nodeInactivityCount[index]*200 > this.predictedCompletion) {
    //^ comparison is milliseconds.
    //^ Allow sub-thread to take up to double predicted duration to takes
    //^ int division and older/slower computers that run this program into account.
    //* If sub-thread takes too long, assume it died, then recreate it with the missing work.
```

This finding must be because the double prediction time took MPI message delay into account when in fact it did not need to.

With error handling

Speeds with 60% failure rate (except the golden standard)

(seconds to the lower whole integer)	golden standard	1 sub-thread	2 sub-threads	3 sub-threads	4 sub-threads	5 sub-threads	6 sub-threads	7 sub-threads	8 sub-threads
trail #1	4	14	96	75	188	275	216	246	308
trail #2	4	58	75	130	92	188	190	298	342
trail #3	4	66	105	166	124	213	288	193	265
trail #4	4	22	79	138	106	211	256	225	311
trail #5	4	35	63	63	167	170	117	180	245
mean	4	39	83.6	114.4	135.4	211.4	213.4	228.4	294.2

75% was used for balance between node cluster simulation and not waiting too long.

Code output

The worst case, 11 sub-threads with 95% failure rate, can be seen:

```
Enter failure rate for sub-threads before they return results (in percentage between 0-100)
95
Human counterpart compliant
Failure rate is 95 percent
Java runtime have 11 available effective processing elements (minus the main thread)
How many processing elements do you want for this operation?
11
Human counterpart compliant
Starting golden standard solution:
Finished golden standard solution.
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
6872299800, 6872, 6
Starting compute cluster solution:
=== compute cluster solution - Sharpen kernel application applied ===
=== compute cluster solution - applied matrix normalized (1 of 2) ===
=== compute cluster solution - Edge-detection kernel application applied ===
=== compute cluster solution - applied matrix normalized (2 of 2) ===
compute cluster solution - Poison Pills administered
Total time of compute cluster solution:
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
3399055178200, 3399055, 3399
Comparing both solutions:
both matrixes are identical in both: dimensions and elements' value
Comparison successful, program finished.
```

With the periodic call change, it is now much faster.

Besides the worst-case scenario, here is a more realistic 20% failure rate 7 sub-thread program call (Assume network is old and unstable):

```
Enter failure rate for sub-threads before they return results (in percentage between 0-100)
20
Human counterpart compliant
Failure rate is 20 percent
Java runtime have 11 available effective processing elements (minus the main thread)
How many processing elements do you want for this operation?
7
Human counterpart compliant
Starting golden standard solution:
Finished golden standard solution.
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
4285438300, 4285, 4
Starting compute cluster solution:
A node/thread has died (thread pool index: 3, thread ID: 24) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 3)
=== compute cluster solution - Sharpen kernel application applied ===
=== compute cluster solution - applied matrix normalized (1 of 2) ===
A node/thread has died (thread pool index: 0, thread ID: 21) - unknown to the thread pool
A node/thread has died (thread pool index: 1, thread ID: 22) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 0)
A node/thread has died (thread pool index: 0, thread ID: 40) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 1)
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 0)
=== compute cluster solution - Edge-detection kernel application applied ===
A node/thread has died (thread pool index: 4, thread ID: 25) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 4)
=== compute cluster solution - applied matrix normalized (2 of 2) ===
A node/thread has died (thread pool index: 1, thread ID: 41) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 1)
compute cluster solution - Poison Pills administered
Total time of compute cluster solution:
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
50861369600, 50861, 50
Comparing both solutions:
both matrixes are identical in both: dimensions and elements' value
Comparison successful, program finished.
```

Scalability

The program was made to simulate the Thread-per-Node Model using the Message-Passing Model, therefore the program was made with scalability in mind (Delporte-Gallet et al., 2003) – both ways.

A developer can easily change the size/dimensions of the randomly generated matrix by changing the integers in the 2-dimensional integer array declaration argument.

```
    public int[][] matrixGen() { return fill(new int[10000][10000]); }
    //^ setting up matrix's dimensions (10,000x10,000) and generating it.
    //^ Other developers can easily manipulate matrix's size in the "new int[][]".
```

This can be seen with applying 11 sub-threads on a 9x9 matrix with 10% fail rate as seen:

```
Enter failure rate for sub-threads before they return results (in percentage between 0-100)
10
Human counterpart compliant
Failure rate is 10 percent
Java runtime have 11 available effective processing elements (minus the main thread)
How many processing elements do you want for this operation?
11
Human counterpart compliant
Randomly generated matrix:
[5, 8, 0, 4, 3, 8, 2, 2, 0]
[8, 9, 8, 0, 7, 8, 8, 6, 9]
[9, 3, 2, 0, 6, 5, 5, 5, 1]
[6, 8, 4, 5, 2, 1, 2, 4, 1]
[2, 3, 0, 4, 0, 6, 9, 4, 4]
[8, 2, 2, 7, 2, 9, 5, 1, 8]
[5, 8, 6, 2, 0, 9, 5, 9, 7]
[1, 4, 4, 8, 4, 7, 9, 5, 6]
[8, 9, 9, 2, 6, 9, 6, 2, 7]
# END MATRIX PRINT #
Starting golden standard solution:
Finished golden standard solution.
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
835200, 0, 0
Resulting matrix from the golden standard solution:
[11, 155, -231, 132, -87, 179, -130, -28, -134]
[68, 73, 241, -190, 103, 18, 94, 4, 308]
[167, -231, -86, -161, 122, -9, 0, 19, -137]
[53, 183, 34, 138, -34, -120, -130, 36, -48]
[-114, -11, -168, 59, -176, 77, 219, -22, 43]
[232, -142, -78, 244, -97, 164, -79, -266, 197]
[35, 168, 72, -119, -222, 173, -135, 171, 52]
[-179, -126, -149, 214, -64, -40, 101, -102, 31]
[230, 153, 210, -169, 76, 158, 37, -133, 214]
# END MATRIX PRINT #
Starting compute cluster solution:
=== compute cluster solution - Sharpen kernel application applied ===
A node/thread has died (thread pool index: 8, thread ID: 29) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 8)
=== compute cluster solution - applied matrix normalized (1 of 2) ===
A node/thread has died (thread pool index: 3, thread ID: 24) - unknown to the thread pool
A node/thread has died (thread pool index: 8, thread ID: 32) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 3)
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 8)
=== compute cluster solution - Edge-detection kernel application applied ===
A node/thread has died (thread pool index: 5, thread ID: 26) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 5)
=== compute cluster solution - applied matrix normalized (2 of 2) ===
A node/thread has died (thread pool index: 1, thread ID: 22) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 1)
compute cluster solution - Poison Pills administered
Total time of compute cluster solution:
Execution time in nanoseconds, milliseconds, and seconds respectfully: VVV
87681200800, 87681, 87
Resulting matrix from the compute cluster solution:
[11, 155, -231, 132, -87, 179, -130, -28, -134]
[68, 73, 241, -190, 103, 18, 94, 4, 308]
[167, -231, -86, -161, 122, -9, 0, 19, -137]
[53, 183, 34, 138, -34, -120, -130, 36, -48]
[-114, -11, -168, 59, -176, 77, 219, -22, 43]
[232, -142, -78, 244, -97, 164, -79, -266, 197]
[35, 168, 72, -119, -222, 173, -135, 171, 52]
[-179, -126, -149, 214, -64, -40, 101, -102, 31]
[230, 153, 210, -169, 76, 158, 37, -133, 214]
# END MATRIX PRINT #
Comparing both solutions:
subtraction proof (a simple subtraction between both matrixes):
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
# END MATRIX PRINT #
both matrixes are identical in both: dimensions and elements' value
Comparison successful, program finished.
```

Or a 25000x25000 matrix as seen:

```
    public int[][] matrixGen() { return fill(new int[12500][12500]); }
    //^ setting up matrix's dimensions (10,000x10,000) and generating it.
    //^ Other developers can easily manipulate matrix's size in the "new int[][]".
```

```
Enter failure rate for sub-threads before they return results (in percentage between 0-100)
10
Human counterpart compliant
Failure rate is 10 percent
Java runtime have 11 available effective processing elements (minus the main thread)
How many processing elements do you want for this operation?
11
Human counterpart compliant
Starting golden standard solution:
Finished golden standard solution.
Execution time in nanoseconds, milliseconds, and seconds respectfully: vvv
6817889000, 6817, 6
Starting compute cluster solution:
A node/thread has died (thread pool index: 6, thread ID: 27) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 6)
=== compute cluster solution - Sharpen kernel application applied ===
=== compute cluster solution - applied matrix normalized (1 of 2) ===
A node/thread has died (thread pool index: 9, thread ID: 30) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 9)
=== compute cluster solution - Edge-detection kernel application applied ===
A node/thread has died (thread pool index: 9, thread ID: 44) - unknown to the thread pool
Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: 9)
=== compute cluster solution - applied matrix normalized (2 of 2) ===
compute cluster solution - Poison Pills administered
Total time of compute cluster solution:
Execution time in nanoseconds, milliseconds, and seconds respectfully: vvv
49057895100, 49057, 49
Comparing both solutions:
both matrixes are identical in both: dimensions and elements' value
Comparison successful, program finished.
```

Initial matrix can still be far much larger (than shown) if it can fit inside the JVM heap memory, otherwise you may need to manually increase it such as with the "-XmxNg" argument where "N" is the number of gigabytes for the new heap memory limit.

Also, if a developer wants to program either (or both) kernels' values, they can by editing the 2-dimensional integer arrays in the "Task4JobKernelApplication" java file fields as seen:

```
public class Task4JobKernelApplication implements Task4Job {
    //: set-up.
    //: Kernels declared in this way to allow other developers to easily manipulate its elements' value
    //: and program will still work as intended.
    private int[][] kernelSharpen = {
        //* element's values can be changed if wished by a developer
        {0, -1, 0},
        {-1, 5, -1},
        {0, -1, 0}
    };
    private int[][] kernelEdgeDetection = {
        //* element's values can be changed if wished by a developer
        {-1, -1, -1},
        {-1, 8, -1},
        {-1, -1, -1}
    };
```

Both kernels' current values taken from Powell (2024).

References

- Powell, V. (2024). Image Kernels explained visually. [online] SETOSA. Available at: https://setosa.io/ev/image-kernels/.
- Delporte-Gallet, C., Fauconnier, H. and Guerraoui, R. (2003). Shared memory vs message passing. [online] Infoscience. Available at: https://infoscience.epfl.ch/server/api/core/bitstreams/eae9da30-a424-4bbe-a6d1-17f8fa12201c/content.
- Opyrchal, L. and Prakash, A. (1999). Efficient object serialization in Java. [online] IEEE. DOI: https://www.doi.org/10.1109/ECMDD.1999.776421.

- Shafi, A., Carpenter, B. and Baker, M. (2009). Nested parallelism for multi-core HPC systems using Java. [online] ScienceDirect. DOI: https://doi.org/10.1016/j.jpdc.2009.02.006.
- Lausdahl, K., Verhoef, M., Larsen, P.G. and Wolff, S. (2010). Overview of VDM-RT constructs and semantic issues. [online] Newcastle University. Available at: https://blog.lausdahl.com/wp-content/uploads/publications/Lausdahl&10.pdf.

Appendixes

Note: each file name is the name of the single class inside of each one.

Code files used by multiple tasks

Golden Standard class

```
public class GoldenStandard {
	//: set-up.
	//: kernels declared in this way to allow other developers to easily manipulate its
elements' value
	private int[][] kernelSharpen = {
		{0, -1, 0},
		{-1, 5, -1},
		{0, -1, 0}
	};
	private int[][] kernelEdgeDetection = {
		//^ also called as "outline" kernel
		{-1, -1, -1},
		{-1, 8, -1},
		{-1, -1, -1}
	};

	public int[][] apply(int[][] matrixInput){
		matrixInput = applyKernel(1 , matrixInput);
		//^ Apply the "Sharpen" kernel to the generated matrix
		matrixInput = applyKernel(2 , matrixInput);
		//^ Apply the "Edge-Detection" kernel to the resulting
		//^ matrix of the "Sharpen" kernel application.
		return matrixInput;
	}

	public int[][] applyKernel(int kernelType, int[][] matrixInput) {
		int[][] matrixResult = new int[matrixInput.length][matrixInput[0].length];
		//^ set-up
		//: apply kernel to each of the matrix elements
		for (int yAxis = 0; yAxis < matrixInput.length; yAxis++) {
			for (int xAxis = 0; xAxis < matrixInput[0].length; xAxis++) {
				switch(kernelType) {
				case 1: matrixResult[yAxis][xAxis] = this.element(kernelSharpen,
matrixInput, yAxis, xAxis); break;
				default: matrixResult[yAxis][xAxis] =
this.element(kernelEdgeDetection, matrixInput, yAxis, xAxis); break;
				//^ when kernelType == 2
				}
			}
		}
		return matrixResult;
	}
	private int element(int[][] kernel, int[][] matrixInput, int yAxis, int xAxis) {
		int total = 0;
		//^ set-up
		//: apply kernel to the element with the element's neighbors
		for (int yAxisKernel = -1; yAxisKernel < kernel.length-1; yAxisKernel++) {
			for (int xAxisKernel = -1; xAxisKernel < kernel.length-1; xAxisKernel++) {
```

```
                        try { total += kernel[yAxisKernel+1][xAxisKernel+1] *
matrixInput[yAxis+yAxisKernel][xAxis+xAxisKernel]; }
                        catch(ArrayIndexOutOfBoundsException error) { }
                        //^ works as zero-padding by not adding operations that include out-
of-bounds
                        //^ indexes as that will be that same as adding zero.
                  }
            }
            return total;
      }
}
```

Utilities class

```
//: imports
import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;
import java.util.concurrent.atomic.AtomicIntegerArray;

public class Utilities {
      public Scanner scanner = new Scanner(System.in);

      private int[][] fill(int[][] matrix) {
            Random rand = new Random();
            //^ random integer generator
            //* fill matrix with random integers (0-9)
            for (int yAxis = 0; yAxis < matrix.length; yAxis++) {
                  for (int xAxis = 0; xAxis < matrix[0].length; xAxis++) {
                        matrix[yAxis][xAxis] = rand.nextInt(10);
                        //^ assign random integer to subjected element
                  }
            }
            return matrix;
      }
      private int[] sameMatrixes(int[][] matrix1, int[][] matrix2) {
            int [][] matrixResult = new int[matrix1.length][matrix1[0].length];
            //^ set-up for the subtraction proof
            //* {-1,0} means same, otherwise matrixes are not the same.
            //* {-2,0} means different dimensions.
            //: subtraction proof
            if ( (matrix1.length != matrix2.length) || (matrix1[0].length !=
matrix2[0].length) ) { return new int[] {-2,0}; }
            //^ check dimensions
            for (int yAxis = 0; yAxis < matrix1.length; yAxis++) {
                  for (int xAxis = 0; xAxis < matrix1.length; xAxis++) {
                        matrixResult[yAxis][xAxis] = matrix1[yAxis][xAxis] -
matrix2[yAxis][xAxis];
                        //^ subtract each individual element value from the corresponding
one from the other matrix at a time
                  }
            }
            printMatrix(matrixResult, "subtraction proof (a simple subtraction between both
matrixes):");
            //^ print the subtraction proof

            //: check the subtraction proof
            for (int yAxis = 0; yAxis < matrix1.length; yAxis++) {
                  for (int xAxis = 0; xAxis < matrix1.length; xAxis++) {
                        if (matrixResult[yAxis][xAxis] != 0) { return new int[]
{yAxis,xAxis}; }
                        //^ return location of first non-identical element value
```

```
                    }
                }
                return new int[] {-1,0};
                //^ executes if matrixes are identical
        }

        public String compareMatrixies(int[][] matrix1, int[][] matrix2) {
                int[] result = sameMatrixes( matrix1, matrix2);
                //^ set-up
                if (result[0] == -2) {return "non-identical matrixes - matrix dimensions are 
different";}
                else if (result[0] == -1) {return "both matrixes are identical in both: dimensions 
and elements' value";}
                else {return String.format("non-identical matrixes - atleast one element's value 
is unequal. First unequal at [%d][%d]",result[0],result[1]);}
        }
        public int[][] matrixGen() { return fill(new int[10000][10000]); }
        //^ setting up matrix's dimensions (10,000x10,000) and generating it.
        //^ Other developers can easily manipulate matrix's size in the "new int[][]".
        public void printMatrix(int[][] matrix, String name) {
                //* printing matrices with descriptions
                System.out.println(name);
                //^ print name/description of matrix
                for (int yAxis = 0; yAxis < matrix.length; yAxis++) {
                        //* print the actual matrix
                        System.out.println( Arrays.toString(matrix[yAxis]) );
                }
                System.out.println("# END MATRIX PRINT #");
                //^ Notify that current print has ended.
                //^ Easily differentiate from other print jobs
        }
        public int getProcessingEleCount() {
                //: set-up
                int availableCount = Runtime.getRuntime().availableProcessors()-1;
                //^ informative to user (non-important).
                //^ 1 less to take the main thread into account
                boolean validCount = false;
                int userCount = 0;
                //^ '0' is a placeholder, method will always return an integer above 0

                while(!validCount) {
                        //* gets a valid input (integer above 0) from user
                        System.out.println( String.format("Java runtime have %d available effective 
processing elements (minus the main thread)", availableCount) );
                        System.out.println("How many processing elements do you want for this 
operation?");
                        if (!this.scanner.hasNextInt()) {
                                System.out.println("Error encounted on human counterpart: invalid 
input - input must be an integer");
                                this.scanner.next();
                                continue;
                        }
                        userCount = this.scanner.nextInt();
                        if( userCount < 1) {
                                //* too few
                                System.out.println("Error encounted on human counterpart: invalid 
input - (integer) input must be positive");
                                this.scanner.next();
                                continue;
                        }
                        if( userCount > availableCount) {
                                System.out.println("Error encounted on human counterpart: invalid 
input - processor count higher than effective maximum");
```

```
                        this.scanner.next();
                        continue;
                    }
                    //: success
                    System.out.println("Human counterpart compliant");
                    validCount = true;
            }

            //: success
            //this.scanner.close();
            return userCount;
    }
    public int[][] convertAtomicArrToInt2D(AtomicIntegerArray[] atomicArray) {
            int[][] matrix = new int[atomicArray.length][atomicArray[0].length()];
            //^ set-up
            for (int yAxis = 0; yAxis < atomicArray.length; yAxis++) {
                    //^ though the 'AtomicIntegerArray' array
                    for (int xAxis = 0; xAxis < atomicArray[0].length(); xAxis++) {
                            //^ though the 'AtomicIntegerArray' array
                            matrix[yAxis][xAxis] = atomicArray[yAxis].get(xAxis);
                    }
            }
            return matrix;
    }
    public void executionTime(long startTime) {
            long endTime = System.nanoTime();
            //^ end execution time
            long executionTime = endTime - startTime;
            //^ calculate execution time
            //: prints the execution time
            System.out.println("Execution time in nanoseconds, milliseconds, and seconds
respectfully: VVV");
            System.out.println(executionTime+", "+(executionTime / 1000000)+",
"+(executionTime / 1000000000));
    }
}
```

Code files exclusively used for task 1

Main code class

```
public class Task1 {
    public static void main(String[] args) {
            //: Setting up by instantiating the classes
            GoldenStandard goldenStandard = new GoldenStandard();
            //^ to compare with tasks 2, 3, and 4
            Utilities utilities = new Utilities();
            //^ for printing a matrix and a generating matrix

            int[][] matrix = utilities.matrixGen();
            //^ generate the 10,000 by 10,000 random matrix
            utilities.printMatrix(matrix, "Randomly generated matrix:");
            //^ prints the initial matrix

            System.out.println("Program is now doing the golden standard solution...");
            int[][] matrixResult = goldenStandard.apply(matrix);
            //^ applying one kernel filter one after the other (sharpen then edge-detection)
            utilities.printMatrix(matrixResult, "Applying Sharpen kernel then Edge-Detection
kernel:");
            //^ prints the resulting matrix
            System.out.println("...The golden standard solution has finished");
    }
```

```
}
```

Code files exclusively used for task 2

Main code class

```
public class Task2 {
	public static int[][] matrixResult2Sharpen;
	//^ the result of deep-copying "matrixResult2"'s values
	//^ before applying the edge-detection kernel filter.
	//^ The threads can read or write to this concurrently.
	public static boolean toFormat = false;
	//^ tells the threads if you apply kernel filter or
	//^ to deep-copy.
	public static int[][] matrixResult2;
	//^ for the threads to write to, for the matrix result
	//^ of both kernels in the multi-threaded solution, concurrently.
	private static Task2WorkerThread[] threads;
	//^ set-up the thread pool to store the threads
	private static Utilities utilities = new Utilities();
	//^ for printing matrixes and generating matrixes

	public static void main(String[] args) {
		int[][] matrix = Task2.utilities.matrixGen();
		//^ generate the 10,000 by 10,000 random matrix
		utilities.printMatrix(matrix, "Randomly generated matrix:");
		//^ prints the initial matrix

		//: set up the atomic two-dimentional int array fields
		Task2.matrixResult2 = new int[matrix.length][matrix[0].length];
		Task2.matrixResult2Sharpen = new int[matrix.length][matrix[0].length];

		int[][] matrixResult = Task2.singleThreadedSolution(matrix, Task2.utilities);
		//^ doing the golden standard (with execution duration) for comparison with the
multi-threaded solution

		//: threaded (task 2 solution)
		System.out.println("Program is now doing the threaded solution...");
		int threadCount = Task2.utilities.getProcessingEleCount();
		//^ get the processor count form user (must be in range of 0 to maximum available
processors)
		long startTime = System.nanoTime();
		//^ start execution time
		Task2.multiThreadedSolution(matrix, threadCount);
		//^ applying one kernel filter after the other (sharpen then edge-detection) but
with parallel threads
		Task2.utilities.executionTime(startTime);
		//^ prints execution time of applying the kernel filters
		utilities.printMatrix(Task2.matrixResult2, "multi threaded solution result:");
		//^ prints the resulting matrix
		System.out.println("...The threaded solution has finished");

		System.out.println(Task2.utilities.compareMatrixies(matrixResult,
Task2.matrixResult2));
	}
	private static void multiThreadedSolution(int[][] matrix, int threadCount){
		//: set-up
		int totalSize = matrix.length * matrix[0].length;
		int sectorSize = totalSize / threadCount;
		//^ calculate element count for each job distribution
		Task2.threads = new Task2WorkerThread[threadCount];
		//^ set-up the thread pool
```

```
            for (int index = 0; index < Task2.threads.length; index++) {
                  Task2.threads[index] = new Task2WorkerThread();
                  Task2.threads[index].start();
            }

            multiThreadedJob(1, matrix, threadCount, sectorSize, totalSize);
            //^ apply the sharpen kernel

            //: deep-copy the result to pass the values, instead of reference, to the next job
            Task2.toFormat = true;
            multiThreadedJob(0, new int [][]{{}}, threadCount, sectorSize, totalSize);
            //^ values of first 2 arguments do not matter as they will not be used
            Task2.toFormat = false;

            multiThreadedJob(2, Task2.matrixResult2Sharpen, threadCount, sectorSize,
totalSize);
            //^ apply the edge-detection kernel

            for (int index = 0; index < Task2.threads.length; index++)
{ Task2.threads[index].done(); }
            //^ acts like a Poison Pill
            for (int index = 0; index < Task2.threads.length; index++) {
                  try { Task2.threads[index].join(); } catch (InterruptedException e)
{ e.printStackTrace(); }
            }
      }
      private static void multiThreadedJob(int kernelType, int[][] matrix, int threadCount,
int sectorSize, int totalSize){

            Task2.threads[threadCount-1].setJob( new Task2Job(kernelType,matrix,(threadCount-
1)*sectorSize,totalSize-1) );
            //^ fill the last thread's job of the thread pool first.
            //^ Last thread runs (with the job) with the sector size and leftover (which is
equal or more
            //^ than the rest), hence it runs before the others.
            //^ It occupies the last index because its sector size and leftover are at the end
of the matrix instead of
            //^ the beginning.

            for (int index = 0; index < threadCount-1; index++) {
                  //* fills the rest of the thread pool and gives the data to the rest of the
threads to do the jobs
                  Task2.threads[index].setJob( new
Task2Job(kernelType,matrix,index*sectorSize,((index+1)*sectorSize)-1) );
            }

            for (int index = 0; index < Task2.threads.length; index++) {
                  //* wait for all threads' jobs to finish
                  while (!Task2.threads[index].isJobDone()) {
                        try { Thread.sleep(1); } catch (InterruptedException e) { }
                        //^ to avoid busy-waiting
                  }
            }
      }
      private static int[][] singleThreadedSolution(int[][] matrix, Utilities
utilities)/*golden standard*/{
            GoldenStandard goldenStandard = new GoldenStandard();
            //^ instantiating the class that does the golden standard solution

            System.out.println("Program is now doing the golden standard solution...");
            long startTime = System.nanoTime();
            //^ start execution time
```

```
            int[][] matrixResult = goldenStandard.apply(matrix);
            //^ applying one kernel filter after the other (sharpen then edge-detection)
            utilities.executionTime(startTime);
            //^ prints execution time of applying the kernel filters
            utilities.printMatrix(matrixResult, "golden standard solution result:");
            //^ prints the resulting matrix
            System.out.println("...The golden standard solution has finished");

            return matrixResult;
      }
}
```

Worker thread class

```
public class Task2WorkerThread extends Thread{
      private Task2Job job;
      private boolean jobDone;
      private boolean running;

      public void setJob(Task2Job job) {
            //^ called by main thread when a new job needs doing
            this.job = job;
            this.jobDone = false;
      }
      public void done() { this.running = false;    }
      //^ when thread is no longer needed - gets called by main thread
      public boolean isJobDone() { return this.jobDone; }
      //^ when thread finished current job - gets called by main thread
      public void run() {
            this.running = true;
            while (this.running) {
                  if (job != null && !job.isDone()) {
                        //^ execute when there is an existing job that is not completed
                        this.job.perform();
                        //^ where the job actually gets done
                        this.jobDone = true;
                        this.job = null;
                  }
                  try { Thread.sleep(1); } catch (InterruptedException e) { }
                  //^ to avoid busy-waiting
            }
      }
}
```

Job class

```
public class Task2Job{
      //: set-up.
      //: kernels declared in this way to allow other developers to easily manipulate its
elements' value
      private int[][] kernelSharpen = {
            {0, -1, 0},
            {-1, 5, -1},
            {0, -1, 0}
      };
      private int[][] kernelEdgeDetection = {
            {-1, -1, -1},
            {-1, 8, -1},
            {-1, -1, -1}
      };

      private boolean done = false;
      //^ notify main thread when the job has finished
```

```
	private int kernelType;
	//^ determines which kernel filter (Sharpen or Edge-Detection) will be used
	private int[][] matrixInput;
	//^ the matrix to apply the specified kernel filter on
	private int start;
	//^ to calculate the position of the first matrix element to apply the operation on.
	private int end;
	//^ to calculate the position of the last matrix element to apply the operation on.
	private int loops;
	//^ tells the nested loop how many matrix elements to apply the operation on (is
calculated)
	private int xAxis;
	//^ the index of the array of int sub-arrays - calculated by "start" and matrix
dimensions
	private int yAxis;
	//^ the index of the selected sub-array - calculated by "start" and dimensions

	public Task2Job(int kernelType, int[][] matrixInput, int start, int end) {
		this.kernelType = kernelType;
		this.matrixInput = matrixInput;
		this.start = start;
		this.end = end;
		this.loops = this.end - this.start;
		this.yAxis = start / Task2.matrixResult2.length;
		this.xAxis = start % Task2.matrixResult2.length;
	}
	public boolean isDone() { return this.done; }
	public void perform() {
		if(Task2.toFormat) { this.deepCopyMatrix(Task2.matrixResult2,
Task2.matrixResult2Sharpen); return; }
		//: set-up
		int matrixElement;
		int loops = this.loops;
		boolean first = true;
		//: set up the starting coords of the matrix
		int yAxis = this.yAxis;
		int xAxis = this.xAxis;

		//: apply kernel to each of the matrix elements
		for (int y = yAxis; y < this.matrixInput.length; y++) {
			if (loops < 0)/* thread finish its part of the matrix*/{ break; }
			if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis =
0;}
			first = !first;

			for (int x = xAxis; x < this.matrixInput[0].length; x++) {
				loops--;
				switch(this.kernelType) {
				case 1: matrixElement = this.element(kernelSharpen,
this.matrixInput, y, x); break;
				default: matrixElement = this.element(kernelEdgeDetection,
this.matrixInput, y, x); break;
				//^ when kernelType == 2

				}

				Task2.matrixResult2[y][x] = matrixElement;
				//^ immediately write the new element value to the array of int
arrays (concurrently)

				if (loops < 0){ break; }
				//^ if job has finished creating its part of the result
			}
```

```
            }

            this.done = true;
            //^ job finished
        }
        private int element(int[][] kernel, int[][] matrixInput, int yAxis, int xAxis) {
            int total = 0;
            //^ set-up
            //: apply kernel to the element with the element's neighbors
            for (int yAxisKernel = -1; yAxisKernel < kernel.length-1; yAxisKernel++) {
                for (int xAxisKernel = -1; xAxisKernel < kernel.length-1; xAxisKernel++) {
                    try { total += kernel[yAxisKernel+1][xAxisKernel+1] *
matrixInput[yAxis+yAxisKernel][xAxis+xAxisKernel]; }
                    catch(ArrayIndexOutOfBoundsException error) { }
                    //^ works as zero-padding by not adding operations that include out-
of-bounds
                    //^ indexes as that will be that same as adding zero.
                }
            }
            return total;
        }
        private void deepCopyMatrix(int[][] matrixCopy, int[][] matrixPaste){
            //* to prevent the reference passed instead of the values (deep-copy)
            //:set-up
            int loops = this.loops;
            boolean first = true;
            //: set up the starting coords of the matrix
            int yAxis = this.yAxis;
            int xAxis = this.xAxis;

            //: deep-copy each of the matrix elements within specified range
            for (int y = yAxis; y < matrixCopy.length; y++) {
                if (loops < 0)/* thread finish its part of the matrix*/{ break; }
                if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis =
0;}
                first = !first;

                for (int x = xAxis; x < matrixCopy[0].length; x++) {
                    loops--;
                    matrixPaste[y][x] = matrixCopy[y][x];
                    //^ DEEP-copy the element's value to the array of atomic int arrays
(concurrently)

                    if (loops < 0){ break; }
                    //^ if job has finished creating its part of the result
                }
            }
            this.done = true;
        }
}
```

Code files exclusively used for task 3

Main code class

```
//: all relevant imports
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Arrays;
```

```
public class Task3 {
	private static Utilities utilities = new Utilities();
	//^ for printing matrixes and generating matrixes
	private static Task3ThreadPool threadPool;
	//^ stores the thread pool instance in the main class as they must be in the same thread
to reflect
	//^ the master node in the Thread-per-Node Model,
	private static int[][] matrix;
	//^ stores randomly generated matrix then used for the result of the Thread-per-Node
solution
	private static int[][] goldenStandardMatrix;
	//^ stores the matrix result of the golden solution
	private static int[][] unformatted;
	//^ stores the jagged array from each kernel application

	public static void main(String[] args) {
		threadPool = new Task3ThreadPool( utilities.getProcessingEleCount() );
		//^ instantiate the thread pool with an appropriate number of sub-threads
determined by user
		GoldenStandard goldenStandard = new GoldenStandard();

		Task3.matrix = Task3.utilities.matrixGen();
		//^ generate the matrix of random numbers (0-9)
		Task3.utilities.printMatrix(matrix, "Randomly generated matrix:");
		//^ prints the generated matrix (in a user-friendly manner)

		long startTime;
		//^ to store start time to time execution speeds

		System.out.println("Starting golden standard solution:");
		startTime = System.nanoTime();
		//^ get current time
		Task3.goldenStandardMatrix = goldenStandard.apply(matrix);
		//^ golden solution calculates result and gets caught into 'goldenStandardMatrix'
field
		System.out.println("Finished golden standard solution.");
		System.out.println("Total time golden standard solution:");
		Task3.utilities.executionTime(startTime);
		//^ calculates and displays execution duration time based on started time
		Task3.utilities.printMatrix(goldenStandardMatrix, "Resulting matrix from the
golden standard solution:");

		System.out.println("Starting compute cluster solution:");
		startTime = System.nanoTime();
		//: record initial matrix dimensions for the normalization parts
		int martix1stDimentsion = matrix.length;
		int martix2ndDimentsion = matrix[0].length;

		//: apply sharpen kernel and normalize result
		Task3.unformatted = Task3.applyKernel(Task3.matrix, 1);
		Task3.matrix = Task3.normalization(unformatted, martix1stDimentsion,
martix2ndDimentsion);

		//: apply edge-detection kernel and normalize result
		Task3.unformatted = Task3.applyKernel(Task3.matrix, 2);
		Task3.matrix = Task3.normalization(unformatted, martix1stDimentsion,
martix2ndDimentsion);

		Task3.poisonPill();
		//^ close all sub-threads
		//: output result and duration time
		System.out.println("Finished compute cluster solution.");
```

```
            System.out.println("Total time of compute cluster solution:");
            Task3.utilities.executionTime(startTime);
            Task3.utilities.printMatrix(Task3.matrix, "Resulting matrix from the compute
cluster solution:");

            System.out.println("Comparing both solutions:");
            System.out.println(Task3.utilities.compareMatrixies(Task3.matrix,
goldenStandardMatrix));
            //^ comparison by simple subtractions
            System.out.println("Comparison done, program finished.");
      }
      private static int[] sliceIndexByWorkload(int[][] matrix, int start, int end, boolean
applyKernel) {
            //* Necessary to minimize workload, instead of giving out whole copies (to nodes),
as much as possible
            //* to not exceed the default maximum memory heap leap limit - especially for
scaling matrix or
            //* sub-thread count (program scalability).
            //* Can work for both normalized and jagged arrays - for code integrity.
            //: all set-ups initialized to satisfy compiler but initialization should not be
part of the returned.
            int startSlice = 0;
            //^ starting sub-array's index
            int endSlice = 0;
            //^ ending sub-array's index
            int startSubArrayIndex = 0;
            //^ starting index inside the range sub-arrays
            int endSubArrayIndex = 0;
            //^ ending index inside the range sub-arrays

            //: determine which sub-arrays are needed and what range inside of them of them
            endSubArrayIndex = end - start;
            //^ assume the the startSubArrayIndex is 0
            for (int index = 0; index < matrix.length; index++) {
                  //* calculates both sub-array start and beginning start range index
                  if (start < matrix[index].length) {
                        startSubArrayIndex = start;
                        startSlice = index;
                        break;
                  }
                  start -= matrix[index].length;
            }
            endSubArrayIndex += startSubArrayIndex;
            //^ do not assume the the startSubArrayIndex is 0
            for (int index = 0; index < matrix.length; index++) {
                  //* calculates just the sub-array end
                  if (end < matrix[index].length) {
                        endSlice = index+1;
                        //^ adds 1 because array slicing does not include the ending index
in the slice
                        break;
                  }
                  end -= matrix[index].length;
            }

            if (applyKernel) {
                  //^ only if applying kernels, not normalization.
                  //: Make sure matrix part is surrounded with more the matrix for the kernel
applications
                  //: to give accurate result.
                  //: extra comparisons for code integrity - in case developer scales matrix
size down.
                  if (startSlice != 0) {
```

```
                    startSlice -= 1;
                    //^ adds the values of above matrix sub-array.
                    //* Considers the shift in range indexes when pushing elements at
the start of the workload.
                    int indexShift = matrix[startSlice].length;
                    startSubArrayIndex += indexShift;
                    endSubArrayIndex += indexShift;
                }
                if (endSlice != matrix.length) {
                    endSlice += 1;
                    //^ adds the values of above matrix sub-array
                }
            }

            return new int[] {startSlice, endSlice, startSubArrayIndex, endSubArrayIndex};
        }
        private static int[][] applyKernel(int[][] matrix, int kernelType){
            int threadCount = Task3.threadPool.threadCount();
            //^ ask thread pool how many sub-threads is it currently handling
            int totalSize = matrix.length * matrix[0].length;
            //^ get element count to calculate distributed workload size
            int sectorSize = totalSize / threadCount;
            //^ calculate distributed workload size
            int[] args;

            //: For code integrity purposes - if another developer changes initial matrix size
to a very small size (STILL MUST BE ATLEAST 1x1).
            if (sectorSize == 0) {
                //^ happens when there are more sub-threads than initial matrix element
count
                sectorSize = 1;
                //^ distribute smallest possible workload
                threadCount = totalSize;
                //^ to not bother giving redundant jobs to the extra sub-threads
            }

            //: determines parameter arguments for queued jobs - considers odd elements out
from workload divisions and certain sub-thread counts
            for (int index = 0; index < threadCount-1; index++) {
                //* fills the rest of the thread pool and gives the data to the rest of the
threads to do the jobs
                args = sliceIndexByWorkload(matrix, index*sectorSize,
((index+1)*sectorSize)-1, true);
                Task3.threadPool.enqueueJob( new
Task3JobKernelApplication(kernelType,Task3.serializeIntArrArr(Arrays.copyOfRange(matrix,args[0]
,args[1])),args[2],args[3]) );
            }
            if (threadCount > 1) {
                //* to account for the remainder matrix elements after division by number
of sub-threads.
                args = sliceIndexByWorkload(matrix, (threadCount-1)*sectorSize,totalSize-1,
true);
                Task3.threadPool.enqueueJob( new
Task3JobKernelApplication(kernelType,Task3.serializeIntArrArr(Arrays.copyOfRange(matrix,args[0]
,args[1])),args[2],args[3]) );
            }
            if (threadCount == 1) {
                //* if user/developer wants to use only one sub-thread, for testing or
comparison reasons.
                Task3.threadPool.enqueueJob( new
Task3JobKernelApplication(kernelType,Task3.serializeIntArrArr(matrix),0,totalSize-1) );
            }
```

```
            Task3.threadPool.wakeUp();
            //^ activate thread pool to let sub-threads fetch jobs

            int[][] result =
Arrays.copyOfRange(Task3.deserializeIntArrArr(Task3.threadPool.getResults()),0,threadCount);
            //^ the array slicing is for the edge-detection kernel application as results from
last thread pool wake-up will be brought along
            //^ in accordance to the 'code integrity purposes' mentioned in this method.

            //: for-loop is for the sharpen kernel application as results will include nulls
in accordance to the 'code integrity purposes' mentioned in this method.
            for (int index = 0; index < result.length; index++) {
                  //* For code integrity purposes (same as the previous integrity purposes)
                  if (result[index]==null) { return Arrays.copyOfRange(result, 0, index); }
            }

            threadPool.clear();
            //^ remove all now-redundant stored data from thread pool to minimise program's
heap memory usage
            return result;
            //^ returns the result as an jagged int array
      }
      private static int[][] normalization(int[][] unformatted, int subArraysCount, int
subArraysLength){
            int threadCount = Task3.threadPool.threadCount();
            //^ ask thread pool how many sub-threads is it currently handling
            int[][] matrix = new int[subArraysCount][subArraysLength];
            //^ creates new int[][] for the result with dimensions based on the original
matrix
            int[][] preMatrix =  new int[threadCount][];
            //^ where the jagged int array
            int totalSize = matrix.length * matrix[0].length;
            //^ get element count to calculate distributed workload size
            int sectorSize = subArraysLength;
            //^ better renaming for method;s body for easier code development and maintenance
            int sectorCount = subArraysCount / threadCount;
            //^ calculate how many sub-arrays (of the jagged array) each thread can handle
            int sectorCountRemainder = subArraysCount % threadCount;
            //^ remainder from "sectorCount" calculation
            int distributedLoadLength;
            //^ stores the length (element count) distributed load
            int matrixIndex;
            //^ index of the to-be-returned result matrix
            int activeThreads = threadCount;
            //^ better renaming for method's body for easier code development and maintenance
            int[] args;

            distributedLoadLength = sectorCount * sectorSize;
            //^ Assume there is no remainder in the "sectorCount" calculation

            //: increase workload if the "distributedLoadLength" assumption was wrong
            if (sectorCountRemainder != 0) {
                  sectorCount++;
                  distributedLoadLength = sectorCount * sectorSize;
                  activeThreads = (totalSize / distributedLoadLength)+1;
            }

            args = sliceIndexByWorkload(unformatted, 0, distributedLoadLength, false);
            if (threadCount == 1){ args[1] = 1; }
            //^ if only one sub-thread was used
            Task3.threadPool.enqueueJob( new
Task3JobNormalization(Task3.serializeIntArrArr(Arrays.copyOfRange(unformatted,0,args[1])),args[
2],distributedLoadLength) );
```

```
            //^ serialize queue the first job (the beginning)

            for (int index = 1; index < activeThreads; index++) {
                    //* serialize queue the other jobs (to the end)
                    args = sliceIndexByWorkload(unformatted, index*distributedLoadLength,
(index+1)*distributedLoadLength, false);
                    //Task4.threadPool.enqueueJob( new
Task4JobNormalization(Task4.serializeIntArrArr(unformatted),(index*distributedLoadLength),distr
ibutedLoadLength) );
                    if (args[1] == 0) { args[1] = unformatted.length; }
                    //^ for the last job
                    Task3.threadPool.enqueueJob( new
Task3JobNormalization(Task3.serializeIntArrArr(Arrays.copyOfRange(unformatted,args[0],args[1]))
,args[2],distributedLoadLength) );
            }

            //: wake up thread pool and get the sub-threads collective results
            threadPool.wakeUp();
            preMatrix = Task3.deserializeIntArrArr(threadPool.getResults());

            //: translate the result to the matrix due to the place-holderelements
            matrixIndex = 0;
            for (int preMatrixIndex = 0; preMatrixIndex < preMatrix.length; preMatrixIndex++)
{
                    boolean doneAllElements = false;
                    for (int jobResultIndex = 0; jobResultIndex <
preMatrix[preMatrixIndex].length; jobResultIndex += sectorSize ) {
                            matrix[matrixIndex] = Arrays.copyOfRange( preMatrix[preMatrixIndex],
jobResultIndex, jobResultIndex+sectorSize );
                            //^ get all elements for one row of the resulting matrix
                            matrixIndex++;
                            //^ next row
                            if ((matrixIndex*subArraysLength) >= totalSize) { doneAllElements =
true; break; }
                            //^ ignore the place-holders
                    }
                    if (doneAllElements) { break; }
            }

            threadPool.clear();
            //^ remove all now-redundant stored data from thread pool to minimise program's
heap memory usage
            return matrix;
            //^ return as the kernel application result normalized
    }
    private static void poisonPill() {
            int threadCount = Task3.threadPool.threadCount();
            for (int index = 0; index < threadCount; index++) {
                    //* fills the thread pool with Poison Pills to close all threads
                    Task3.threadPool.enqueueJob( new Task3JobPoisonPill() );
            }
            Task3.threadPool.wakeUp();
            //^ make thread pool send poison pill (as jobs) to the sub-theads to execute
    }
  private static byte[] serializeIntArrArr(int[][] intArrArr){
    //* To send serialized copies so referencing between nodes are limited to twice per
communication/message
      ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
      //^ holds and formats the byte data of the argument
      ObjectOutputStream outStream = null;
      //^ 'ObjectOutputStream' serialize and write objects.
      //^ Initialized as null because of the try-statement but null should never used
```

```
        try {
            //^ try-statement to deal with possible IOException errors (mandated by compiler)
                outStream = new ObjectOutputStream(byteOutStream);
                //^ 'ObjectOutputStream' initialized for serialization and writing to
'byteOutStream'
                outStream.writeObject(intArrArr);
                //^ serialize the argument and writes it to 'byteOutStream'
                outStream.close();
                //^ closes initialization to free up resources
        }
        catch (IOException e) {  e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
        return byteOutStream.toByteArray();
        //^ Returns a byte array containing the serialized data
    }
    private static int[][] deserializeIntArrArr(byte[] data){
       //* no 200ms network delay in this method's calls because 'main' and
       //* the thread pool are both in the same main thread
        ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
        //^ holds and formats the byte data of the argument
        ObjectInputStream inStream;
        //^ 'ObjectOutputStream' serialize and write objects.
        int[][] deserialized = null;
        //^ initialized as null because of the try-statement but null should never be returned

        try {
            //^ try-statement to deal with potential IOException or ClassNotFoundException
errors (mandated by compiler)
                inStream = new ObjectInputStream(byteInStream);
                // 'ObjectInputStream' initialized for deserialization and writing to
'byteInStream'
                deserialized = (int[][]) inStream.readObject();
                //^ deserialize the byte array argument and writes it to 'byteOutStream'.
                //^ The only code in this method that can potentially cause a
'ClassNotFoundException' error.
                inStream.close();
                //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {     e.printStackTrace(); }
        //^ for possible errors relating to deserialization, int[][] parsing, and the
'ObjectInputStream' instance
        return deserialized;
        //^ Returns a 2d int array containing the deserialized data
    }
}
```

Thread pool class

```
//: all relevant imports
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;

public class Task3ThreadPool {
       //* thread pool is used to a Thread-per-Node model (HPC cluster of computer nodes).

       private boolean sleep;
       //^ stops the thread pool from searching for threads to
       //^ message the jobs to when there are no jobs left.
       private Task3WorkerNode[] threads;
```

```
        //^ where the worker threads are referenced.
        //^ Each thread ID will be their index in the array
        private ArrayList<Task3Job> jobQueue;
        //^ FIFO stack structure (first in first out).
        //^ Contains the jobs as messages.
        private int[][] threadJobResults;
        //^ job results that are concurrently written to by the threads (atomic).
        //^ Each thread is to write to corresponding element, even if overwriting,
        //^ thus the index serves as the thread's ID (identification).
        private int maxJobs;
        //^ For code integrity purposes - if another developer changes initial
        //^ matrix size to a small size it will still work (STILL MUST BE ATLEAST 1x1).
        public Task3ThreadPool(int threadCount) {
                //: set-up the fields
                this.sleep = false;
                //^ activate the thread pool
                jobQueue = new ArrayList<Task3Job>();
                //^ for good practice and scalability - the data structure has dynamic size
                threadJobResults = new int[threadCount][];
                //^ is a jagged array, because jobs' returned int arrays' length are not
predefined

                //: set-up and fill the thread pool
                this.threads = new Task3WorkerNode[threadCount];
                for (int index = 0; index < threadCount; index++) {
                        this.threads[index] = new Task3WorkerNode();
                        //^ worker thread instance
                        this.threads[index].start();
                        //^ start the thread's execution
                }
        }

        //: public methods that are called by main ('Task3') class
        public int threadCount() /*getter method*/ { return this.threads.length; }
        public void wakeUp() { this.start(); }
        //^ notifies the thread pool - simulates the ".notify()" method call
        public void enqueueJob(Task3Job job) /*add method (FIFO style)*/ { jobQueue.add(job); }
        //^ enqueues a job by appending it to the end of the stack
        public byte[] getResults() /*getter method*/ { return
this.serializeIntArrArr(this.threadJobResults); }
        public void clear() /*reset method*/ {
                //* not exactly aligned with the Thread-per-Node Model but the program must not
exceed the Java
                //* default maximum heap memory limit, which cannot be changed in-program.
                //* The lower it is, the better the program's scalability.
                //: these data are, at this point of time, redundant so they must be deleted
                this.threadJobResults = new int[this.threads.length][];
                for (int index = 0; index < this.threads.length; index++) {
        threads[index].clear();    }
                //^ clears each worker node's stored result
                try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
                //^ to impose a delay of 200ms to simulate the speeds of modern computers and
cluster networks.
                //^ Tells all threads to delete their now redundant stored result.
        }

        private Task3Job dequeueJob() {
                //* follows the FIFO standard
                //: "First Out"
                Task3Job job = jobQueue.get(0);
                jobQueue.remove(0);
                return job;
                //^ to be sent, as a message, to the thread.
```

```
        }

        private void requestResults() {
                boolean allResults = false;
                //^ dictates if it thread pool should keep waiting for all results or not
                boolean[] alldone = new boolean[this.maxJobs];
                //^ only fetch results from the relevant threads to save processing time
                while (!allResults) {
                        //* far faster than the alternative of constantly checking a sub-thread job
until it
                        //* is done one at a time.
                        for (int index = 0; index < this.maxJobs; index++) {
                                //* get all results to if each thread has finished the job or not
                                alldone[index] = threads[index].isJobDone();
                                //^ store the each job status
                        }
                        try { Thread.sleep(200); } catch (InterruptedException e)
{ e.printStackTrace(); }
                        //^ to impose a delay of 200ms for each computational job executed to
simulate the
                        //^ speeds of modern computers and cluster networks.
                        //^ These node-to-node messages are to check if the all sub-thread has
finished
                        //^ their given jobs.
                        allResults = true;
                        //^ Assume that all threads have finished
                        for (int index = 0; index < this.maxJobs; index++) {
                                //* only gathers threads that were given relevant jobs
                                if(alldone[index] == false) {
                                        //^ current thread hasn't finished?
                                        allResults = false;
                                        //^ if one thread isn't finished then all threads are not
finished!
                                        break;
                                        //^ no point to continue so break off to save computational
time
                                }

                        }
                }
                for (int index = 0; index < this.maxJobs; index++) {
                        //* gather results of all relevant threads
                        threadJobResults[index] =
this.deserializeIntArr(this.threads[index].requestResult());
                        //^ deserialize raw data MPI message into the fetched int[] result to be
stored
                }
                try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
                //^ to impose a delay of 200ms for each computational job executed to simulate the
                //^ speeds of modern computers and cluster networks.
                //^ These node-to-node messages are to get the serialized results from the sub-
threads.
        }
        private void start() {
                //* called by ".wakeUp()" by other classes.
                //* Assign jobs and gather results.
                this.maxJobs = jobQueue.size();
                this.sleep = false;
                while (!this.sleep) {
                        try { Thread.sleep(1); } catch (InterruptedException e)
{ e.printStackTrace(); }
                        //^ to avoid busy-waiting
                        for (int index = 0; index < this.maxJobs; index++) {
```

```
                            //* assign a job per worker thread
                            if (this.jobQueue.size() == 0) { break; }
                            //^ cannot assign a non-existant job to a worker thread
                            threads[index].setJob(this.serializeJob(this.dequeueJob()));
                            //^ take from FIFO stack and into a serialized MPI message to the
worker thread
                    }
                    try { Thread.sleep(200); } catch (InterruptedException e)
{ e.printStackTrace(); }
                    //^ to impose a delay of 200ms for each computational job executed to
simulate the
                    //^ speeds of modern computers and cluster networks.
                    //^ this node-to-node message is the serialized job from the main thread to
the sub-thread.
                    //^ Thread-pool can send multiple messages without waiting for delay of the
sub-thread
                    //^ receiving the message.
                    if (this.jobQueue.size() == 0) { this.sleep = true; }
                    //^ go to sleep when there are currently no more jobs
             }
             this.requestResults();
             //^ after assigning jobs, we gather all results when all threads finish their
given jobs
       }
    private byte[] serializeJob(Task3Job job){
       //* between thread pool and worker nodes/threads - to reduce reference calls as much as
possible.
       //* To send serialized copies so referencing between nodes are limited to twice per
communication/message.
        ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
        //^ holds and formats the byte data of the argument
        ObjectOutputStream outStream = null;
        //^ 'ObjectOutputStream' serialize and write objects.
        //^ initialized as null because of the try-statement but null should never used

        try {
             //^ try-statement to deal with possible IOException errors (mandated by compiler)
              outStream = new ObjectOutputStream(byteOutStream);
              //^ 'ObjectOutputStream' initialized for serialization and writing to
'byteOutStream'
              outStream.writeObject(job);
              //^ serialize the argument and writes it to 'byteOutStream'
              outStream.close();
              //^ closes initialization to free up resources
        }
        catch (IOException e) {  e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance

        return byteOutStream.toByteArray();
        //^ Returns a byte array containing the serialized data
    }
    private int[] deserializeIntArr(byte[] data){
        ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
        //^ holds and formats the byte data of the argument.
        ObjectInputStream inStream;
        //^ 'ObjectOutputStream' serialize and write objects.
        int[] deserialized = null;
        //^ initialized as null because of the try-statement but null should never be returned

        try {
             //^ try-statement to deal with potential IOException or ClassNotFoundException
errors (mandated by compiler)
              inStream = new ObjectInputStream(byteInStream);
```

```
            //^ 'ObjectInputStream' initialized for deserialization and writing to
'byteInStream'
            deserialized = (int[]) inStream.readObject();
            //^ deserialize the byte array argument and writes it to 'byteOutStream'.
            //^ the only code in this method that can potentially cause a
'ClassNotFoundException' error.
            inStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {     e.printStackTrace(); }
        //^ for possible errors relating to deserialization, int[] parsing, and the
'ObjectInputStream' instance

        return deserialized;
        //^ Returns a int array containing the deserialized data
    }
    private byte[] serializeIntArrArr(int[][] array){
       //* between thread pool and main - same thread but still used for deep-copying
        ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
        //^ holds and formats the byte data of the argument
        ObjectOutputStream outStream = null;
        //^ 'ObjectOutputStream' serialize and write objects.
        //^ initialized as null because of the try-statement but null should never used

        try {
            //^ try-statement to deal with possible IOException errors (mandated by compiler)
            outStream = new ObjectOutputStream(byteOutStream);
            //^ 'ObjectOutputStream' initialized for serialization and writing to
'byteOutStream'
            outStream.writeObject(array);
            //^ serialize the 2d int argument and writes it to 'byteOutStream'
            outStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException e) {  e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance

        return byteOutStream.toByteArray();
        //^ Returns a byte array containing the serialized data
    }
}
```

Worker node class

```
//: import for serialization.
//: As compute nodes have copies of the built-in libraries, this is not shared
//: memory - sticking to the Thread-per-Node model (HPC cluster simulation).
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Task3WorkerNode extends Thread {
       //* simulates an individual compute node in a HPC cluster in accordance to
       //* the Thread-per-Node model.
       private Task3Job job;
       //^ stores the current job
       private boolean jobDone = true;
       //^ toggles when either a new job is given or the current job has finished
       private boolean running;
       //^ acts as a Poison Pill
       private int[] jobResult;
```

```
        //^ result of current job is caught and stored here

        public void setJob(byte[] job) {
                //^ called by main thread when a new job needs doing
                this.job = this.deserializeJob(job);
                this.jobDone = false;
        }
        public void done()/*setter method*/{ this.running = false; }
        //^ when thread is no longer needed - gets called by main thread
        public boolean isJobDone()/*getter method*/{ return this.jobDone; }
        //^ when thread finished current job - gets called by main thread
        public byte[] requestResult() { return this.serializeIntArr(this.jobResult); }
        //^ in compute clusters, results are stored in the worker nodes before retrieval
        public void clear() { this.jobResult = null; }
        //^ to help prevent the Java program from exceeding the default maximum heap size limit

        public void run() {
                this.running = true;
                //^ dictates if the thread should continue running or end
                while (this.running) {
                        //* while-loop runs constantly till thread must end
                        if (job != null && !this.jobDone) {
                                //^ execute when there is an existing job that is not completed
                                jobResult = this.job.perform();
                                //^ where the job actually gets done
                                if (jobResult.length == 0 ) { this.running = false; }
                                //^ shuts down thread (after this loop) if job is a Poison Pill
                                this.jobDone = true;
                                //^ prevents node from repeating the same job twice (clears
redundancy)
                                this.job = null;
                                //^ prevents node from running empty code before first job
                        }
                        try { Thread.sleep(1); } catch (InterruptedException e) { }
                        //^ to avoid busy-waiting
                }
        }
     private byte[] serializeIntArr(int[] array){
        //* to send the result as a serialized raw data message instead of pass by reference
          ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
          //^ holds and formats the byte data of the argument
          ObjectOutputStream outStream = null;
          //^ 'ObjectOutputStream' serialize and write objects.
          //^ initialized as null because of the try-statement but null should never used

          try {
                //^ try-statement to deal with possible IOException errors (mandated by compiler)
                  outStream = new ObjectOutputStream(byteOutStream);
                  //^ 'ObjectOutputStream' initialized for serialization and writing to
'byteOutStream'
                  outStream.writeObject(array);
                  //^ serialize the argument and writes it to 'byteOutStream'
                  outStream.close();
                  //^ closes initialization to free up resources
          }
          catch (IOException e) {  e.printStackTrace(); }
          //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
          return byteOutStream.toByteArray();
          //^ Returns a byte array containing the serialized data
     }
     private Task3Job deserializeJob(byte[] data){
        //* to deserialized the raw data message into a job
          ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
```

```
        //^ holds and formats the byte data of the argument
        ObjectInputStream inStream;
        //^ 'ObjectOutputStream' serialize and write objects.
        Task3Job deserialized = null;
        //^ initialized as null because of the try-statement but null should never be returned

        try {
             //^ try-statement to deal with potential IOException or ClassNotFoundException
errors (mandated by compiler)
                inStream = new ObjectInputStream(byteInStream);
                //^ 'ObjectInputStream' initialized for deserialization and writing to
'byteInStream'
                deserialized = (Task3Job) inStream.readObject();
                //^ deserialize the byte array argument and writes it to 'byteOutStream'.
                //^ the only code in this method that can potentially cause a
'ClassNotFoundException' error.
                inStream.close();
                //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {     e.printStackTrace(); }
        //^ for possible errors relating to deserialization, job parsing, and the
'ObjectInputStream' instance
        return deserialized;
        //^ Returns a job containing the deserialized data
    }
}
```

Job interface

```
import java.io.Serializable;
//^ For 'Task3Job' to implement 'Serializable'

public interface Task3Job extends Serializable{
       //^ any class implementing this interface also implements 'Serializable' so the classes'
       //^ instances are able to get serialized and deserialized.
       //* Interface allows the worker thread to accept any kind of job (deep copy and kernel
application).
       //* Implemented by the 'Task3JobKernelApplication', 'Task3JobNormalization', and
       //* 'Task3JobPoisonPill' classes.
       boolean isDone();
       //^ allows sub-thread to check if the job has finished yet (returns true if '.perform()'
finishes)
       int[] perform();
       //^ Executes the main body (in the 'perform' public method) of the job in the sub-thread
in
       //^ concurrency.
       int[][] deserializeIntArrArr(byte[] data);
       //^ Deserialises the 2d int array.
       //^ Due to the argument being serialized, reference calls are encapsulated inside of the
       //^ the method itself meaning that no "pass by reference" is done over the threads/nodes
       //^ in the computer network in the method's body.
       //^ Despite byte[] being an object, in a network simulation (Thread-per-Node model),
       //^ byte[] serves container for raw data for Messaging Passing Interface (MPI), rather
       //^ than a reference to the original data structure - thus the method call also does
       //^ not pass any direct reference calls between threads.
}
```

Kernel application job class

```
//: imports
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
```

```
public class Task3JobKernelApplication implements Task3Job {
	//: set-up.
	//: Kernels declared in this way to allow other developers to easily manipulate its
elements' value
	//: and program will still work as intended.
	private int[][] kernelSharpen = {
		//* element's values can be changed if wished by a developer
		{0, -1, 0},
		{-1, 5, -1},
		{0, -1, 0}
	};
	private int[][] kernelEdgeDetection = {
		//* element's values can be changed if wished by a developer
		{-1, -1, -1},
		{-1, 8, -1},
		{-1, -1, -1}
	};

	private boolean done = false;
	//^ notify main thread when the job has finished
	private int kernelType;
	//^ determines which kernel filter (Sharpen or Edge-Detection) will be used
	private int[][] matrixInput;
	//^ the matrix to apply the specified kernel filter on
	private int start;
	//^ to calculate the position of the first matrix element to apply the operation on.
	private int end;
	//^ to calculate the position of the last matrix element to apply the operation on.
	private int[] result;
	//^ where the result of the job is stored to be later fetched by the worker node/thread

	public Task3JobKernelApplication(int kernelType, byte[] matrixInput, int start, int end)
{
		this.kernelType = kernelType;
		this.matrixInput = this.deserializeIntArrArr(matrixInput);
		this.start = start;
		this.end = end;
	}
	@Override
	public boolean isDone()/*getter method*/{ return this.done; }
	@Override
	public int[] perform() {
		//: set-up
		int matrixElement;
		//^ where the result of applying kernel, on a element, is temporarily stored
		int loops = this.end - this.start;
		//^ tells the nested loop how many matrix elements to apply the operation on (is
calculated)
		this.result = new int[loops+1];
		//^ field assignment is based on local variable, hence it is here instead of in
the constructor
		boolean first = true;
		//: set up the starting coords of the matrix
		int yAxis = start / this.matrixInput[0].length;
		//^ the index of the array of int sub-arrays - calculated by "start" and matrix
dimensions
		int xAxis = start % this.matrixInput[0].length;
		//^ the index of the selected sub-array - calculated by "start" and dimensions

		//: apply kernel to each of the matrix elements
		for (int y = yAxis; y < this.matrixInput.length; y++) {
			if (loops < 0)/* thread finish its part of the matrix*/{ break; }
```

```
                    if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis =
0;}
                    first = !first;
                    //^ no longer the first applied sub-array

                    for (int x = xAxis; x < this.matrixInput[0].length; x++) {
                          loops--;
                          //: applies kernel of element depending on which kernel is selected.
                          //: Didn't use enumerations (enum) because it will make the
serialization process, to
                          //: prevent cross-node reference passes, more complex and takes more
                          //: computational time.
                          switch(this.kernelType) {
                          case 1: matrixElement = this.element(kernelSharpen,
this.matrixInput, y, x); break;
                          //^ '1' corresponds to applying the Sharpen kernel
                          default: matrixElement = this.element(kernelEdgeDetection,
this.matrixInput, y, x); break;
                          //^ when kernelType == 2.
                          //^ '2' corresponds to applying the Edge-Detection kernel.
                          //^ using default satisfy compiler so I do not need to initialize
'matrixElement' with
                          //^ a redundant value.
                          }

                          this.result[((result.length-1)-loops)-1] = matrixElement;
                          //^ immediately write the new element value to the array of int
arrays (concurrently)

                          if (loops < 0){ break; }
                          //^ if job has finished creating its part of the result
                    }
              }
              this.done = true;
              //^ job finished
              return this.result;
              //^ store result for worker thread to fetch
       }
       private int element(int[][] kernel, int[][] matrixInput, int yAxis, int xAxis) {
              int total = 0;
              //^ set-up
              //: apply kernel to the element with the element's neighbors
              for (int yAxisKernel = -1; yAxisKernel < kernel.length-1; yAxisKernel++) {
                    for (int xAxisKernel = -1; xAxisKernel < kernel.length-1; xAxisKernel++) {
                          try { total += kernel[yAxisKernel+1][xAxisKernel+1] *
matrixInput[yAxis+yAxisKernel][xAxis+xAxisKernel]; }
                          catch(ArrayIndexOutOfBoundsException error) { }
                          //^ works as zero-padding by not adding operations that include out-
of-bounds
                          //^ indexes as that will be that same as adding zero.
                    }
              }
              return total;
       }
       @Override
    public int[][] deserializeIntArrArr(byte[] data){
       //* to deserialize the raw data int[][] to operate on it
         ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
         //^ holds and formats the byte data of the argument
         ObjectInputStream inStream;
         //^ 'ObjectOutputStream' serialize and write objects.
         int[][] deserialized = null;
         //^ initialized as null because of the try-statement but null should never be returned
```

```
        try {
             //^ try-statement to deal with potential IOException or ClassNotFoundException
errors (mandated by compiler)
               inStream = new ObjectInputStream(byteInStream);
               //^ 'ObjectInputStream' initialized for deserialization and writing to
'byteInStream'
               deserialized = (int[][]) inStream.readObject();
               //^ deserialize the byte array argument and writes it to 'byteOutStream'.
               //^ the only code in this method that can potentially cause a
'ClassNotFoundException' error.
               inStream.close();
               //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {     e.printStackTrace(); }
        //^ for possible errors relating to deserialization, int[][] parsing, and the
'ObjectInputStream' instance

        return deserialized;
        //^ Returns a 2d int array containing the deserialized data
    }
}
```

Normalization job class

```
//: relevant imports
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class Task3JobNormalization implements Task3Job{
       private boolean done = false;
       //^ notify main thread when the job has finished
       private int[][] unformatted;
       //^ the matrix to apply the specified kernel filter on
       private int start;
       //^ to calculate the position of the first matrix element to apply the operation on.
       private int length;
       //^ tells the nested loop how many matrix elements to apply the operation on (is
calculated)
       private int[] result;
       //^ where the result of the job is stored to be later fetched by the worker node/thread

       public Task3JobNormalization(byte[] unformatted, int start, int length) {
              this.unformatted = this.deserializeIntArrArr(unformatted);
              //^ originally serialized it to prevent passing of references between
threads/nodes
              this.start = start;
              this.length = length;
              this.result = new int[length];
              //^ the result is based of how big of proportion of normalization we want this job
              //^ to do itself.
       }

       @Override
       public boolean isDone()/*getter method*/{ return this.done; }

       @Override
       public int[] perform(){
              //* to prevent the reference passed instead of the values (deep-copy)
              //:set-up
              boolean first = true;
              //^ for the stating index of the 2nd subject array and beyond
```

```
            int remaining = this.start;
            //: set up the starting index of the input (not all sub-arrays are always equal 
lengths)
            int yAxis = 0;
            //^ the starting index of the array of int sub-arrays - calculated by "start" and 
dimensions.
            //^ Assigned as zero for sake of initialization but the zero will not be read 
until change.
            int xAxis;
            //^ the starting index of the selected int sub-array - calculated by "start" and 
dimensions
            //^ (2nd dimensional index).
            int loops = this.length;
            //^ dictates how many elements left the job must operate on for its part of the 
normalization

            //: calculate the index of the int array of int arrays (yAxis)
            for(int index = 0; index < this.unformatted.length; index++) {
                  if (remaining <= this.unformatted[index].length-1) { yAxis = index; 
break; }
                  remaining -= (this.unformatted[index].length);
            }

            xAxis = remaining;
            //^ 2nd dimension index is based on 1st dimension (yAxis) calculation

            //: normalize part of the kernel's result based within specified range
            for (int y = yAxis; y < this.unformatted.length; y++) {
                  if (loops == 0)/* thread finish its part of the matrix*/{ break; }
                  if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis = 
0;}

                  if (first) { first = false; };
                  //^ after this iteration, the current iteration no longer be the first

                  for (int x = xAxis; x < this.unformatted[y].length; x++) {
                        //^ going via each element in current sub-array
                        loops--;
                        //^ one less element to process the current job's part of the 
normalization
                        if (!( y == this.unformatted.length || x == 
this.unformatted[y].length )) {
                              //^ prevent out-of-bounds error when going though the jagged 
array (as sub-array
                              //^ lengths are inconsistent and unknown).
                              this.result[(this.result.length-1)-loops] = 
this.unformatted[y][x];
                              //^ deep-copy the element's value to the array of atomic int 
arrays (concurrently)
                        }
                        else {
                              this.result[(this.result.length-1)-loops] = 0;
                              //^ We don't want resulting int[][] to also be jagged so we 
add placeholders as '0's
                        }
                        if (loops == 0){ break; }
                        //^ if job has finished creating its part of the result
                  }
            }

            this.done = true;
            //^ to tell the corresponding worker thread that the job has finished when it 
checks it
            return this.result;
```

```
            //^ stores result for the corresponding worker thread to fetch
        }
        @Override
    public int[][] deserializeIntArrArr(byte[] data){
            //^ has to be public because it is part of the 'Task3Job' interface
        //* to deserialize the raw data int[][] to operate on it
          ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
          //^ holds and formats the byte data of the argument
          ObjectInputStream inStream;
          //^ 'ObjectOutputStream' serialize and write objects.
          int[][] deserialized = null;
          //^ initialized as null because of the try-statement but null should never be returned

          try {
            //^ try-statement to deal with potential IOException or ClassNotFoundException
errors (mandated by compiler)
                inStream = new ObjectInputStream(byteInStream);
                //^ 'ObjectInputStream' initialized for deserialization and writing to
'byteInStream'
                deserialized = (int[][]) inStream.readObject();
                //^ deserialize the byte array argument and writes it to 'byteOutStream'.
                //^ the only code in this method that can potentially cause a
'ClassNotFoundException' error.
                inStream.close();
                //^ closes initialization to free up resources
          }
          catch (IOException|ClassNotFoundException e) {     e.printStackTrace(); }
          //^ for possible errors relating to deserialization, int[][] parsing, and the
'ObjectInputStream' instance

          return deserialized;
          //^ Returns a 2d int array containing the deserialized data
    }
}
```

Poison pill class

```
public class Task3JobPoisonPill implements Task3Job{
        //^ implements 'Task3Job' so it can be stored in the thread pool's
        //^ FIFO job queue.
        private boolean done = false;
        //^ simple set-up

        @Override
        public boolean isDone() { return done; }
        //^ returns true when '.perform()' gets executed, otherwise return false

        @Override
        public int[] perform() { this.done = true; return new int[] { }; }
        //^ when sub-thread catches result of job, it breaks while-loop and
        //^ closes thread if result is an empty int array.
        //^ It returns int[] as the other jobs returns int[] as well, hence
        //^ why so does the method in the 'Task3Job' interface.

        @Override
        public int[][] deserializeIntArrArr(byte[] data) { return null;   }
        //^ this method will never be called but is there because deserialization
        //^ is one of the main parts of a job (hence is in the job interface),
        //^ deserialized data to produce results, the Poison Pill enters sub-thread
        //^ but unlike jobs to process as a job to stop the sub-thread's while-loop
        //^ (inside 'start' method) to close the sub-threads otherwise program will
        //^ have zombie threads.
}
```

Code files exclusively used for task 4

Main code class

```
//: all relevant imports
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.Arrays;
import java.util.Scanner;

public class Task4 {
	private static Utilities utilities = new Utilities();
	//^ for printing matrixes and generating matrixes
	private static Task4ThreadPool threadPool;
	//^ stores the thread pool instance in the main class as they must be in the same thread
to reflect
	//^ the master node in the Thread-per-Node Model,
	private static int[][] matrix;
	//^ stores randomly generated matrix then used for the result of the Thread-per-Node
solution
	private static int[][] goldenStandardMatrix;
	//^ stores the matrix result of the golden solution
	private static int[][] unformatted;
	//^ stores the jagged array from each kernel application
	private static int failureRate;
	//^ stores failure rate in the main "Task4" class to be passed to the worker node
constructor
	//^ as a parameter/argument.

	public static void main(String[] args) {
		failureRate = Task4.inputFailureRate(utilities.scanner);
		//^ get failure rate from user.
		//^ Cannot open 2 scanners (even with "scanner.close()") so had to use the
utilities scanner.
		threadPool = new Task4ThreadPool( utilities.getProcessingEleCount(),
failureRate );
		//^ instantiate the thread pool with an appropriate number of sub-threads
determined by user
		GoldenStandard goldenStandard = new GoldenStandard();

		Task4.matrix = Task4.utilities.matrixGen();
		//^ generate the matrix of random numbers (0-9)
		//Task4.utilities.printMatrix(matrix, "Randomly generated matrix:");
		//^ prints the generated matrix (in a user-friendly manner)

		long startTime;
		//^ to store start time to time execution speeds

		System.out.println("Starting golden standard solution:");
		startTime = System.nanoTime();
		//^ get current time
		Task4.goldenStandardMatrix = goldenStandard.apply(matrix);
		//^ golden solution calculates result and gets caught into 'goldenStandardMatrix'
field
		System.out.println("Finished golden standard solution.");
		Task4.utilities.executionTime(startTime);
		//^ calculates and displays execution duration time based on started time
```

```
		//Task4.utilities.printMatrix(goldenStandardMatrix, "Resulting matrix from the
golden standard solution:");

		System.out.println("Starting compute cluster solution:");
		startTime = System.nanoTime();
		while (true) {
			//* To simulate possible MPI message corruptions disrupting the final
error.
			//: Record initial matrix dimensions for the normalization parts
			int martix1stDimentsion = matrix.length;
			int martix2ndDimentsion = matrix[0].length;

			//: apply sharpen kernel and normalize result
			Task4.unformatted = Task4.applyKernel(Task4.matrix, 1);
			System.out.println("=== compute cluster solution - Sharpen kernel
application applied ===");
			Task4.matrix = Task4.normalization(Task4.unformatted, martix1stDimentsion,
martix2ndDimentsion);
			System.out.println("=== compute cluster solution - applied matrix
normalized (1 of 2) ===");

			//: apply edge-detection kernel and normalize result
			Task4.unformatted = Task4.applyKernel(Task4.matrix, 2);
			System.out.println("=== compute cluster solution - Edge-detection kernel
application applied ===");
			Task4.matrix = Task4.normalization(Task4.unformatted, martix1stDimentsion,
martix2ndDimentsion);
			System.out.println("=== compute cluster solution - applied matrix
normalized (2 of 2) ===");

			Task4.poisonPill();
			//^ close all sub-threads
			System.out.println("compute cluster solution - Poison Pills administered");
			//: output result and duration time
			System.out.println("Total time of compute cluster solution:");
			Task4.utilities.executionTime(startTime);
			//Task4.utilities.printMatrix(Task4.matrix, "Resulting matrix from the
compute cluster solution:");

			System.out.println("Comparing both solutions:");
			String comparison = Task4.utilities.compareMatrixies(Task4.matrix,
goldenStandardMatrix);
			//^ comparison by simple subtractions
			System.out.println(comparison);
			if (comparison == "both matrixes are identical in both: dimensions and
elements' value") { break; }

			System.out.println("Redoing solution!");
		}
		System.out.println("Comparison successful, program finished.");
	}
	private static int[] sliceIndexByWorkload(int[][] matrix, int start, int end, boolean
applyKernel) {
		//* Necessary to minimize workload, instead of giving out whole copies (to nodes),
as much as possible
		//* to not exceed the default maximum memory heap leap limit - especially for
scaling matrix or
		//* sub-thread count (program scalability).
		//* Can work for both normalized and jagged arrays - for code integrity.
		//: all set-ups initialized to satisfy compiler but initialization should not be
part of the returned.
		int startSlice = 0;
		//^ starting sub-array's index
```

```
            int endSlice = 0;
            //^ ending sub-array's index
            int startSubArrayIndex = 0;
            //^ starting index inside the range sub-arrays
            int endSubArrayIndex = 0;
            //^ ending index inside the range sub-arrays

            //: determine which sub-arrays are needed and what range inside of them of them
            endSubArrayIndex = end - start;
            //^ assume the the startSubArrayIndex is 0
            for (int index = 0; index < matrix.length; index++) {
                //* calculates both sub-array start and beginning start range index
                if (start < matrix[index].length) {
                    startSubArrayIndex = start;
                    startSlice = index;
                    break;
                }
                start -= matrix[index].length;
            }
            endSubArrayIndex += startSubArrayIndex;
            //^ do not assume the the startSubArrayIndex is 0
            for (int index = 0; index < matrix.length; index++) {
                //* calculates just the sub-array end
                if (end < matrix[index].length) {
                    endSlice = index+1;
                    //^ adds 1 because array slicing does not include the ending index
in the slice
                    break;
                }
                end -= matrix[index].length;
            }

            if (applyKernel) {
                //^ only if applying kernels, not normalization.
                //: Make sure matrix part is surrounded with more the matrix for the kernel
applications
                //: to give accurate result.
                //: extra comparisons for code integrity - in case developer scales matrix
size down.
                if (startSlice != 0) {
                    startSlice -= 1;
                    //^ adds the values of above matrix sub-array.
                    //* Considers the shift in range indexes when pushing elements at
the start of the workload.
                    int indexShift = matrix[startSlice].length;
                    startSubArrayIndex += indexShift;
                    endSubArrayIndex += indexShift;
                }
                if (endSlice != matrix.length) {
                    endSlice += 1;
                    //^ adds the values of above matrix sub-array
                }
            }

            return new int[] {startSlice, endSlice, startSubArrayIndex, endSubArrayIndex};
        }
        private static int[][] applyKernel(int[][] matrix, int kernelType){
            int threadCount = Task4.threadPool.threadCount();
            //^ ask thread pool how many sub-threads is it currently handling
            int totalSize = matrix.length * matrix[0].length;
            //^ get element count to calculate distributed workload size
            int sectorSize = totalSize / threadCount;
            //^ calculate distributed workload size
```

```
            int[] args;

            //: For code integrity purposes - if another developer changes initial matrix size
to a very small size (STILL MUST BE ATLEAST 1x1).
            if (sectorSize == 0) {
                  //^ happens when there are more sub-threads than initial matrix element
count
                  sectorSize = 1;
                  //^ distribute smallest possible workload
                  threadCount = totalSize;
                  //^ to not bother giving redundant jobs to the extra sub-threads
            }

            //: determines parameter arguments for queued jobs - considers odd elements out
from workload divisions and certain sub-thread counts
            for (int index = 0; index < threadCount-1; index++) {
                  //* fills the rest of the thread pool and gives the data to the rest of the
threads to do the jobs
                  args = sliceIndexByWorkload(matrix, index*sectorSize,
((index+1)*sectorSize)-1, true);
                  Task4.threadPool.enqueueJob( new
Task4JobKernelApplication(kernelType,Task4.serializeIntArrArr(Arrays.copyOfRange(matrix,args[0]
,args[1])),args[2],args[3]) );
            }
            if (threadCount > 1) {
                  //* to account for the remainder matrix elements after division by number
of sub-threads.
                  args = sliceIndexByWorkload(matrix, (threadCount-1)*sectorSize,totalSize-1,
true);
                  Task4.threadPool.enqueueJob( new
Task4JobKernelApplication(kernelType,Task4.serializeIntArrArr(Arrays.copyOfRange(matrix,args[0]
,args[1])),args[2],args[3]) );
            }
            if (threadCount == 1) {
                  //* if user/developer wants to use only one sub-thread, for testing or
comparison reasons.
                  Task4.threadPool.enqueueJob( new
Task4JobKernelApplication(kernelType,Task4.serializeIntArrArr(matrix),0,totalSize-1) );
            }

            Task4.threadPool.wakeUp();
            //^ activate thread pool to let sub-threads fetch jobs

            int[][] result =
Arrays.copyOfRange(Task4.deserializeIntArrArr(Task4.threadPool.getResults()),0,threadCount);
            //^ the array slicing is for the edge-detection kernel application as results from
last thread pool wake-up will be brought along
            //^ in accordance to the 'code integrity purposes' mentioned in this method.

            //: for-loop is for the sharpen kernel application as results will include nulls
in accordance to the 'code integrity purposes' mentioned in this method.
            for (int index = 0; index < result.length; index++) {
                  //* For code integrity purposes (same as the previous integrity purposes)
                  if (result[index]==null) { return Arrays.copyOfRange(result, 0, index); }
            }

            threadPool.clear();
            //^ remove all now-redundant stored data from thread pool to minimise program's
heap memory usage
            return result;
            //^ returns the result as an jagged int array
      }
```

```
	private static int[][] normalization(int[][] unformatted, int subArraysCount, int
subArraysLength){
		int threadCount = Task4.threadPool.threadCount();
		//^ ask thread pool how many sub-threads is it currently handling
		int[][] matrix = new int[subArraysCount][subArraysLength];
		//^ creates new int[][] for the result with dimensions based on the original
matrix
		int[][] preMatrix =  new int[threadCount][];
		//^ where the jagged int array
		int totalSize = matrix.length * matrix[0].length;
		//^ get element count to calculate distributed workload size
		int sectorSize = subArraysLength;
		//^ better renaming for method;s body for easier code development and maintenance
		int sectorCount = subArraysCount / threadCount;
		//^ calculate how many sub-arrays (of the jagged array) each thread can handle
		int sectorCountRemainder = subArraysCount % threadCount;
		//^ remainder from "sectorCount" calculation
		int distributedLoadLength;
		//^ stores the length (element count) distributed load
		int matrixIndex;
		//^ index of the to-be-returned result matrix
		int activeThreads = threadCount;
		//^ better renaming for method's body for easier code development and maintenance
		int[] args;

		distributedLoadLength = sectorCount * sectorSize;
		//^ Assume there is no remainder in the "sectorCount" calculation

		//: increase workload if the "distributedLoadLength" assumption was wrong
		if (sectorCountRemainder != 0) {
			sectorCount++;
			distributedLoadLength = sectorCount * sectorSize;
			activeThreads = (totalSize / distributedLoadLength)+1;
		}

		args = sliceIndexByWorkload(unformatted, 0, distributedLoadLength, false);
		if (threadCount == 1){ args[1] = 1; }
		//^ if only one sub-thread was used
		Task4.threadPool.enqueueJob( new
Task4JobNormalization(Task4.serializeIntArrArr(Arrays.copyOfRange(unformatted,0,args[1])),args[
2],distributedLoadLength) );
		//^ serialize queue the first job (the beginning)

		for (int index = 1; index < activeThreads; index++) {
			//* serialize queue the other jobs (to the end)
			args = sliceIndexByWorkload(unformatted, index*distributedLoadLength,
(index+1)*distributedLoadLength, false);
			//Task4.threadPool.enqueueJob( new
Task4JobNormalization(Task4.serializeIntArrArr(unformatted),(index*distributedLoadLength),distr
ibutedLoadLength) );
			if (args[1] == 0) { args[1] = unformatted.length; }
			//^ for the last job
			Task4.threadPool.enqueueJob( new
Task4JobNormalization(Task4.serializeIntArrArr(Arrays.copyOfRange(unformatted,args[0],args[1]))
,args[2],distributedLoadLength) );
		}

		//: wake up thread pool and get the sub-threads collective results
		threadPool.wakeUp();
		preMatrix = Task4.deserializeIntArrArr(threadPool.getResults());

		//: translate the result to the matrix due to the place-holderelements
		matrixIndex = 0;
```

```
            for (int preMatrixIndex = 0; preMatrixIndex < preMatrix.length; preMatrixIndex++)
{
                    boolean doneAllElements = false;
                    for (int jobResultIndex = 0; jobResultIndex <
preMatrix[preMatrixIndex].length; jobResultIndex += sectorSize ) {
                            matrix[matrixIndex] = Arrays.copyOfRange( preMatrix[preMatrixIndex],
jobResultIndex, jobResultIndex+sectorSize );
                            //^ get all elements for one row of the resulting matrix
                            matrixIndex++;
                            //^ next row
                            if ((matrixIndex*subArraysLength) >= totalSize) { doneAllElements =
true; break; }
                            //^ ignore the place-holders
                    }
                    if (doneAllElements) { break; }
            }

            threadPool.clear();
            //^ remove all now-redundant stored data from thread pool to minimise program's
heap memory usage
            return matrix;
            //^ return as the kernel application result normalized
    }
    private static void poisonPill() {
            int threadCount = Task4.threadPool.threadCount();
            for (int index = 0; index < threadCount; index++) {
                    //* fills the thread pool with Poison Pills to close all threads
                    Task4.threadPool.enqueueJob( new Task4JobPoisonPill() );
            }
            Task4.threadPool.wakeUp();
            //^ make thread pool send poison pill (as jobs) to the sub-theads to execute
    }
    private static byte[] serializeIntArrArr(int[][] intArrArr){
        //* To send serialized copies so referencing between nodes are limited to twice per
communication/message
        ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
        //^ holds and formats the byte data of the argument
        ObjectOutputStream outStream = null;
        //^ 'ObjectOutputStream' serialize and write objects.
        //^ Initialized as null because of the try-statement but null should never used

        try {
            //^ try-statement to deal with possible IOException errors (mandated by compiler)
             outStream = new ObjectOutputStream(byteOutStream);
             //^ 'ObjectOutputStream' initialized for serialization and writing to
'byteOutStream'
             outStream.writeObject(intArrArr);
             //^ serialize the argument and writes it to 'byteOutStream'
             outStream.close();
             //^ closes initialization to free up resources
        }
        catch (IOException e) {  e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance

        return byteOutStream.toByteArray();
        //^ Returns a byte array containing the serialized data
    }
    private static int[][] deserializeIntArrArr(byte[] data){
        //* no 200ms network delay in this method's calls because 'main' and
        //* the thread pool are both in the same main thread
        ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
        //^ holds and formats the byte data of the argument
        ObjectInputStream inStream;
```

```
        //^ 'ObjectOutputStream' serialize and write objects.
        int[][] deserialized = null;
        //^ initialized as null because of the try-statement but null should never be returned

        try {
            //^ try-statement to deal with potential IOException or ClassNotFoundException
errors (mandated by compiler)
                inStream = new ObjectInputStream(byteInStream);
                // 'ObjectInputStream' initialized for deserialization and writing to
'byteInStream'
                deserialized = (int[][]) inStream.readObject();
                //^ deserialize the byte array argument and writes it to 'byteOutStream'.
                //^ The only code in this method that can potentially cause a
'ClassNotFoundException' error.
                inStream.close();
                //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {     e.printStackTrace(); }
        //^ for possible errors relating to deserialization, int[][] parsing, and the
'ObjectInputStream' instance

        return deserialized;
        //^ Returns a 2d int array containing the deserialized data
    }
        private static int inputFailureRate(Scanner scanner) {
            //* private method of the main Task4 class.
            //* Code taken and modified from "Utilities.getProcessingEleCount()".
            //: set-up
            boolean validCount = false;
            //^ stores the validity of users input
            int userInput = 0;
            //^ '0' is a placeholder, but assume user first meant 0% failure rate

            while(!validCount) {
                    //* gets a valid input (0-100 as percentage) from user
                    System.out.println("Enter failure rate for sub-threads before they return
results (in percentage between 0-100)");
                    if (!scanner.hasNextInt()) {
                            System.out.println("Error encounted on human counterpart: invalid
input - input must be an integer");
                            scanner.next();
                            continue;
                    }
                    userInput = scanner.nextInt();
                    if( userInput < 0) {
                            //* below percentage range
                            System.out.println("Error encounted on human counterpart: invalid
input - (integer) input must be positive");
                            scanner.next();
                            continue;
                    }
                    if( userInput > 100) {
                            //* above percentage range
                            System.out.println("Error encounted on human counterpart: invalid
input - connot be above 100%");
                            scanner.next();
                            continue;
                    }
                    //: success
                    System.out.println("Human counterpart compliant");
                    System.out.println( String.format("Failure rate is %d percent",
userInput) );
                    validCount = true;
```

```
            }

            //: success
            //scanner.close();
            return userInput;
        }
}
```

Thread pool class

```
//: all relevant imports
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;

public class Task4ThreadPool {
        //* thread pool is used to a Thread-per-Node model (HPC cluster of computer nodes).

        private boolean sleep;
        //^ stops the thread pool from searching for threads to
        //^ message the jobs to when there are no jobs left.
        private Task4WorkerNode[] threads;
        //^ where the worker threads are referenced.
        //^ Each thread ID will be their index in the array
        private ArrayList<Task4Job> jobQueue;
        //^ FIFO stack structure (first in first out).
        //^ Contains the jobs as messages.
        private int[][] threadJobResults;
        //^ job results that are concurrently written to by the threads (atomic).
        //^ Each thread is to write to corresponding element, even if overwriting,
        //^ thus the index serves as the thread's ID (identification).
        private int maxJobs;
        //^ For code integrity purposes - if another developer changes initial
        //^ matrix size to a small size it will still work (STILL MUST BE ATLEAST 1x1).
        private int failureRate;
        //^ Store the failure chance (in %) of a node failure to reflect possible node
        //^ failure in the node cluster of the Thread-Per-Node model
        private Task4Job[] jobHistory;
        //^ store copies of every job, until clearance, because of the assumption that
        //^ at least one node will fail.
        private int[] nodeInactivityCount;
        //^ time how many times the predicted completion duration has each node has run
        //^ the current node for.
        private int predictedCompletion;
        //^ predicted completion duration for the both kernel application and
        //^ normalizations separately (total divided by 4).

        public Task4ThreadPool(int threadCount, int failureRate) {
                //: set-up the fields
                this.sleep = false;
                //^ activate the thread pool
                this.jobQueue = new ArrayList<Task4Job>();
                //^ for good practice and scalability - the data structure has dynamic size
                this.threadJobResults = new int[threadCount][];
                //^ is a jagged array, because jobs' returned int arrays' length are not
predefined
                this.failureRate = failureRate;
                //^ stored because we assume at least one sub-thread will fail
                this.predictedCompletion = (((3*threadCount)+9)/4)*1000;
```

```
            //^ prediction time of a quarter of "y=3x+9" (for each set of jobs) but in
milliseconds
            //: set-up and fill the thread pool
            this.threads = new Task4WorkerNode[threadCount];
            for (int index = 0; index < threadCount; index++) {
                  this.threads[index] = new Task4WorkerNode(failureRate, index);
                  //^ worker thread instance, parameters are both primitives so no references
                  //^ are passed cross-node/thread.
                  this.threads[index].start();
                  //^ start the thread's execution
            }
      }

      //: public methods that are called by main ('Task4') class
      public int threadCount() /*getter method*/ { return this.threads.length; }
      public void wakeUp() { this.start(); }
      //^ notifies the thread pool - simulates the ".notify()" method call
      public void enqueueJob(Task4Job job) /*add method (FIFO style)*/ { jobQueue.add(job); }
      //^ enqueues a job by appending it to the end of the stack
      public byte[] getResults() /*getter method*/ { return
this.serializeIntArrArr(this.threadJobResults); }
      public void clear() /*reset method*/ {
            //* not exactly aligned with the Thread-per-Node Model but the program must not
exceed the Java
            //* default maximum heap memory limit, which cannot be changed in-program.
            //* The lower it is, the better the program's scalability.
            //: these data are, at this point of time, redundant so they must be deleted
            this.threadJobResults = new int[this.threads.length][];
            this.jobHistory = null;
            for (int index = 0; index < this.threads.length; index++) {
      threads[index].clear();   }
            //^ clears each worker node's stored result
            try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
            //^ to impose a delay of 200ms to simulate the speeds of modern computers and
cluster networks.
            //^ Tells all threads to delete their now redundant stored result.
      }

      private Task4Job dequeueJob() {
            //* follows the FIFO standard
            //: "First Out"
            Task4Job job = jobQueue.get(0);
            jobQueue.remove(0);
            return job;
            //^ to be sent, as a message, to the thread.
      }

      private void requestResults() {
            this.nodeInactivityCount = new int[this.threads.length];
            boolean allResults = false;
            //^ dictates if it thread pool should keep waiting for all results or not
            boolean[] alldone = new boolean[this.maxJobs];
            //^ only fetch results from the relevant threads to save processing time
            while (!allResults) {
                  //* far faster than the alternative of constantly checking a sub-thread job
until it
                  //* is done one at a time.
                  for (int index = 0; index < this.maxJobs; index++) {
                        //* get all results to if each thread has finished the job or not
                        alldone[index] = threads[index].isJobDone();
                        //^ store the each job status
                  }
```

```
                    try { Thread.sleep(200); } catch (InterruptedException e)
{ e.printStackTrace(); }
                    //^ to impose a delay of 200ms for each computational job executed to
simulate the
                    //^ speeds of modern computers and cluster networks.
                    //^ These node-to-node messages are to check if the all sub-thread has
finished
                    //^ their given jobs.
                    allResults = true;
                    //^ Assume that all threads have finished
                    for (int index = 0; index < this.maxJobs; index++) {
                        //* only gathers threads that were given relevant jobs
                        if(alldone[index] == false) {
                            //^ current thread hasn't finished?
                            allResults = false;
                            //^ if one thread isn't finished then all threads are not
finished!

                            //: check for dead threads by multiple periodic calls.
                            //: Assume that the main thread never dies.
                            nodeInactivityCount[index]++;
                            //^ sub-thread has not given result yet
                            if (nodeInactivityCount[index]*200 > this.predictedCompletion)
{
                                //^ comparison is milliseconds.
                                //^ Allow sub-thread to take up to double predicted
duration to takes
                                //^ int division and older/slower computers that run
this program into account.
                                //* If sub-thread takes too long, assume it died, then
recreate it with the missing work.
                                this.threads[index] = null;
                                this.threads[index] = new Task4WorkerNode(failureRate,
index);
                                //^ make new thread in the dead thread's place in the
thread pool

      this.threads[index].setJob(this.serializeJob(this.jobHistory[index]));
                                //^ assign missing work based on job assignment history
                                try { Thread.sleep(200); } catch (InterruptedException
e) { e.printStackTrace(); }
                                //^ to impose a delay of 200ms for another
computational job executed to simulate the
                                //^ speeds of modern computers and cluster networks.
                                //^ Sent to the node in place of the one that was
detected as dead.
                                System.out.println(String.format("Thread pool detected
dead node - node recreated and starting with former node's job (thread pool index: %d)",
index));
                                this.threads[index].start();
                                //^ Tell the new sub-thread to start executing jobs.
                                try { Thread.sleep(200); } catch (InterruptedException
e) { e.printStackTrace(); }
                                //^ to impose a delay of 200ms when telling the node to
execute the job.
                                //^ To simulate the speeds of modern computers and
cluster networks.
                                nodeInactivityCount[index] = 0;
                                //^ reset inactivity counter as thread is brand new
                            }

                        }
                    }
```

```
            }
            for (int index = 0; index < this.maxJobs; index++) {
                    //* gather results of all relevant threads
                    threadJobResults[index] =
this.deserializeIntArr(this.threads[index].requestResult());
                    //^ deserialize raw data MPI message into the fetched int[] result to be
stored
            }
            try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
            //^ to impose a delay of 200ms for each computational job executed to simulate the
            //^ speeds of modern computers and cluster networks.
            //^ These node-to-node messages are to get the serialized results from the sub-
threads.
    }
    private void start() {
            //* called by ".wakeUp()" by other classes.
            //* Assign jobs and gather results.
            this.maxJobs = jobQueue.size();
            jobHistory = jobQueue.toArray(new Task4Job[this.maxJobs]);
            this.sleep = false;
            while (!this.sleep) {
                    try { Thread.sleep(1); } catch (InterruptedException e)
{ e.printStackTrace(); }
                    //^ to avoid busy-waiting
                    for (int index = 0; index < this.maxJobs; index++) {
                            //* assign a job per worker thread
                            if (this.jobQueue.size() == 0) { break; }
                            //^ cannot assign a non-existant job to a worker thread
                            threads[index].setJob(this.serializeJob(this.dequeueJob()));
                            //^ take from FIFO stack and into a serialized MPI message to the
worker thread
                    }
                    try { Thread.sleep(200); } catch (InterruptedException e)
{ e.printStackTrace(); }
                    //^ to impose a delay of 200ms for each computational job executed to
simulate the
                    //^ speeds of modern computers and cluster networks.
                    //^ this node-to-node message is the serialized job from the main thread to
the sub-thread.
                    //^ Thread-pool can send multiple messages without waiting for delay of the
sub-thread
                    //^ receiving the message.
                    if (this.jobQueue.size() == 0) { this.sleep = true; }
                    //^ go to sleep when there are currently no more jobs
            }
            this.requestResults();
            //^ after assigning jobs, we gather all results when all threads finish their
given jobs
    }
  private byte[] serializeJob(Task4Job job){
    //* between thread pool and worker nodes/threads - to reduce reference calls as much as
possible.
    //* To send serialized copies so referencing between nodes are limited to twice per
communication/message.
        ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
        //^ holds and formats the byte data of the argument
        ObjectOutputStream outStream = null;
        //^ 'ObjectOutputStream' serialize and write objects.
        //^ initialized as null because of the try-statement but null should never used

        try {
            //^ try-statement to deal with possible IOException errors (mandated by compiler)
             outStream = new ObjectOutputStream(byteOutStream);
```

```
            //^ 'ObjectOutputStream' initialized for serialization and writing to 
'byteOutStream'
            outStream.writeObject(job);
            //^ serialize the argument and writes it to 'byteOutStream'
            outStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException e) {  e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance

        return byteOutStream.toByteArray();
        //^ Returns a byte array containing the serialized data
    }
    private int[] deserializeIntArr(byte[] data){
        ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
        //^ holds and formats the byte data of the argument.
        ObjectInputStream inStream;
        //^ 'ObjectOutputStream' serialize and write objects.
        int[] deserialized = null;
        //^ initialized as null because of the try-statement but null should never be returned

        try {
           //^ try-statement to deal with potential IOException or ClassNotFoundException 
errors (mandated by compiler)
            inStream = new ObjectInputStream(byteInStream);
            //^ 'ObjectInputStream' initialized for deserialization and writing to 
'byteInStream'
            deserialized = (int[]) inStream.readObject();
            //^ deserialize the byte array argument and writes it to 'byteOutStream'.
            //^ the only code in this method that can potentially cause a 
'ClassNotFoundException' error.
            inStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {     e.printStackTrace(); }
        //^ for possible errors relating to deserialization, int[] parsing, and the 
'ObjectInputStream' instance

        return deserialized;
        //^ Returns a int array containing the deserialized data
    }
    private byte[] serializeIntArrArr(int[][] array){
       //* between thread pool and main - same thread but still used for deep-copying
        ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
        //^ holds and formats the byte data of the argument
        ObjectOutputStream outStream = null;
        //^ 'ObjectOutputStream' serialize and write objects.
        //^ initialized as null because of the try-statement but null should never used

        try {
           //^ try-statement to deal with possible IOException errors (mandated by compiler)
            outStream = new ObjectOutputStream(byteOutStream);
            //^ 'ObjectOutputStream' initialized for serialization and writing to 
'byteOutStream'
            outStream.writeObject(array);
            //^ serialize the 2d int argument and writes it to 'byteOutStream'
            outStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException e) {  e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance

        return byteOutStream.toByteArray();
```

```
        //^ Returns a byte array containing the serialized data
    }
}
```

Worker node class

```
//: imports
import java.util.Random;
//^ to do percentage chances for potential node failure
//: import for serialization.
//: As compute nodes have copies of the built-in libraries, this is not shared
//: memory - sticking to the Thread-per-Node model (HPC cluster simulation).
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class Task4WorkerNode extends Thread {
      //* simulates an individual compute node in a HPC cluster in accordance to
      //* the Thread-per-Node model.
      private Task4Job job;
      //^ stores the current job
      private boolean jobDone = true;
      //^ toggles when either a new job is given or the current job has finished
      private boolean running;
      //^ acts as a Poison Pill
      private int[] jobResult;
      //^ result of current job is caught and stored here
      private int failureRate;
      //^ the probability of thread dying before fetching job's result
      private int threadPoolIndex;
      //^ let the user know which node/thread died when it does.

      public Task4WorkerNode(int failureRate, int threadPoolIndex) {
            //* constructor for the failure percentage change.
            //* Parameters are primitives so no references are passed between nodes.
            this.failureRate = failureRate;
            this.threadPoolIndex = threadPoolIndex;
      }

      private boolean nodeDeath() {
            //* calculation to decide if the node/thread was to live or die
            Random rand = new Random();
            boolean death = rand.nextInt(100) < this.failureRate;
            //^ based on percentage chance, does node/thread dies?
            if (death) {
                  //* node/thread dies
                  long threadId = Thread.currentThread().getId();
                  //^ does not interfere with node's job to actually simulate the node's
death - not realising that it is dead.
                  System.out.println(String.format("A node/thread has died (thread pool
index: %d, thread ID: %d) - unknown to the thread pool", this.threadPoolIndex, threadId));
                  //^ thread pool index is more relevant than thread id because index
correlate to what workload the specific sub-thread is handling
                  //^ but still including both in case if user wants both numbers.
                  //^ This tells user that sub-thread died but thread pool does not know that
yet.
            }
            return death;
            //^ will die?
      }
```

```
	public void run() {
		this.running = true;
		//^ dictates if the thread should continue running or end
		while (this.running) {
			//* while-loop runs constantly till thread must end

			if (Thread.interrupted()) { break; }
			//^ ending ".run()", when interrupted, to end thread by death.
			//^ Does not toggle "this.running" because node/thread cannot do
			//^ anything when it dies.

			if (job != null && !this.jobDone) {
				//^ execute when there is an existing job that is not completed

				int[] jobResultCache = this.job.perform();
				//^ where the job actually gets done but caught result is stored
				//^ in-scope until the thread survives the death probability.
				if (!this.nodeDeath()) { jobResult = jobResultCache; }
				//^ if thread does not die (by % probability), then store result
				//^ where it can be fetched by the thread pool and continue as
normal.
				else { this.interrupt(); continue; }
				//^ if thread dies then end ".run()" execution by interruption when
it gets checked

				if (jobResult.length == 0 ) { this.running = false; }
				//^ shuts down thread (after this loop) if job is a Poison Pill
				this.jobDone = true;
				//^ prevents node from repeating the same job twice (clears
redundancy)
				this.job = null;
				//^ prevents node from running empty code before first job
			}
			try { Thread.sleep(1); } catch (InterruptedException e) { }
			//^ to avoid busy-waiting
		}
	}

	public void setJob(byte[] job) {
		//^ called by main thread when a new job needs doing
		this.job = this.deserializeJob(job);
		this.jobDone = false;
	}
	public void done()/*setter method*/{ this.running = false; }
	//^ when thread is no longer needed - gets called by main thread
	public boolean isJobDone()/*getter method*/{ return this.jobDone; }
	//^ when thread finished current job - gets called by main thread
	public byte[] requestResult() { return this.serializeIntArr(this.jobResult); }
	//^ in compute clusters, results are stored in the worker nodes before retrieval
	public void clear() { this.jobResult = null; }
	//^ to help prevent the Java program from exceeding the default maximum heap size limit

    private byte[] serializeIntArr(int[] array){
	//* to send the result as a serialized raw data message instead of pass by reference
	  ByteArrayOutputStream byteOutStream = new ByteArrayOutputStream();
	  //^ holds and formats the byte data of the argument
	  ObjectOutputStream outStream = null;
	  //^ 'ObjectOutputStream' serialize and write objects.
	  //^ initialized as null because of the try-statement but null should never used

	  try {
		//^ try-statement to deal with possible IOException errors (mandated by compiler)
		  outStream = new ObjectOutputStream(byteOutStream);
```

```
            //^ 'ObjectOutputStream' initialized for serialization and writing to
'byteOutStream'
            outStream.writeObject(array);
            //^ serialize the argument and writes it to 'byteOutStream'
            outStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException e) {  e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
        return byteOutStream.toByteArray();
        //^ Returns a byte array containing the serialized data
    }
    private Task4Job deserializeJob(byte[] data){
      //* to deserialized the raw data message into a job
        ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
        //^ holds and formats the byte data of the argument
        ObjectInputStream inStream;
        //^ 'ObjectOutputStream' serialize and write objects.
        Task4Job deserialized = null;
        //^ initialized as null because of the try-statement but null should never be returned

        try {
             //^ try-statement to deal with potential IOException or ClassNotFoundException
errors (mandated by compiler)
            inStream = new ObjectInputStream(byteInStream);
            //^ 'ObjectInputStream' initialized for deserialization and writing to
'byteInStream'
            deserialized = (Task4Job) inStream.readObject();
            //^ deserialize the byte array argument and writes it to 'byteOutStream'.
            //^ the only code in this method that can potentially cause a
'ClassNotFoundException' error.
            inStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException|ClassNotFoundException e) {     e.printStackTrace(); }
        //^ for possible errors relating to deserialization, job parsing, and the
'ObjectInputStream' instance
        return deserialized;
        //^ Returns a job containing the deserialized data
    }
}
```

Job interface

```
import java.io.Serializable;
//^ For 'Task4Job' to implement 'Serializable'

public interface Task4Job extends Serializable{
     //^ any class implementing this interface also implements 'Serializable' so the classes'
     //^ instances are able to get serialized and deserialized.
     //* Interface allows the worker thread to accept any kind of job (deep copy and kernel
application).
     //* Implemented by the 'Task4JobKernelApplication', 'Task4JobNormalization', and
     //* 'Task4JobPoisonPill' classes.
     boolean isDone();
     //^ allows sub-thread to check if the job has finished yet (returns true if '.perform()'
finishes)
     int[] perform();
     //^ Executes the main body (in the 'perform' public method) of the job in the sub-thread
in
     //^ concurrency.
     int[][] deserializeIntArrArr(byte[] data);
     //^ Deserialises the 2d int array.
```

```
        //^ Due to the argument being serialized, reference calls are encapsulated inside of the
        //^ the method itself meaning that no "pass by reference" is done over the threads/nodes
        //^ in the computer network in the method's body.
        //^ Despite byte[] being an object, in a network simulation (Thread-per-Node model),
        //^ byte[] serves container for raw data for Messaging Passing Interface (MPI), rather
        //^ than a reference to the original data structure - thus the method call also does
        //^ not pass any direct reference calls between threads.
}
```

Kernel application job class

```
//: imports
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class Task4JobKernelApplication implements Task4Job {
        //: set-up.
        //: Kernels declared in this way to allow other developers to easily manipulate its
elements' value
        //: and program will still work as intended.
        private int[][] kernelSharpen = {
                //* element's values can be changed if wished by a developer
                {0, -1, 0},
                {-1, 5, -1},
                {0, -1, 0}
        };
        private int[][] kernelEdgeDetection = {
                //* element's values can be changed if wished by a developer
                {-1, -1, -1},
                {-1, 8, -1},
                {-1, -1, -1}
        };

        private boolean done = false;
        //^ notify main thread when the job has finished
        private int kernelType;
        //^ determines which kernel filter (Sharpen or Edge-Detection) will be used
        private int[][] matrixInput;
        //^ the matrix to apply the specified kernel filter on
        private int start;
        //^ to calculate the position of the first matrix element to apply the operation on.
        private int end;
        //^ to calculate the position of the last matrix element to apply the operation on.
        private int[] result;
        //^ where the result of the job is stored to be later fetched by the worker node/thread

        public Task4JobKernelApplication(int kernelType, byte[] matrixInput, int start, int end)
{
                this.kernelType = kernelType;
                this.matrixInput = this.deserializeIntArrArr(matrixInput);
                this.start = start;
                this.end = end;
        }
        @Override
        public boolean isDone()/*getter method*/{ return this.done; }
        @Override
        public int[] perform() {
                //: set-up
                int matrixElement;
                //^ where the result of applying kernel, on a element, is temporarily stored
                int loops = this.end - this.start;
```

```
            //^ tells the nested loop how many matrix elements to apply the operation on (is 
calculated)
            this.result = new int[loops+1];
            //^ field assignment is based on local variable, hence it is here instead of in 
the constructor
            boolean first = true;
            //: set up the starting coords of the matrix
            int yAxis = start / this.matrixInput[0].length;
            //^ the index of the array of int sub-arrays - calculated by "start" and matrix 
dimensions
            int xAxis = start % this.matrixInput[0].length;
            //^ the index of the selected sub-array - calculated by "start" and dimensions

            //: apply kernel to each of the matrix elements
            for (int y = yAxis; y < this.matrixInput.length; y++) {
                if (loops < 0)/* thread finish its part of the matrix*/{ break; }
                if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis = 
0;}
                first = !first;
                //^ no longer the first applied sub-array

                for (int x = xAxis; x < this.matrixInput[0].length; x++) {
                    loops--;
                    //: applies kernel of element depending on which kernel is selected.
                    //: Didn't use enumerations (enum) because it will make the 
serialization process, to
                    //: prevent cross-node reference passes, more complex and takes more
                    //: computational time.
                    switch(this.kernelType) {
                    case 1: matrixElement = this.element(kernelSharpen, 
this.matrixInput, y, x); break;
                    //^ '1' corresponds to applying the Sharpen kernel
                    default: matrixElement = this.element(kernelEdgeDetection, 
this.matrixInput, y, x); break;
                    //^ when kernelType == 2.
                    //^ '2' corresponds to applying the Edge-Detection kernel.
                    //^ using default satisfy compiler so I do not need to initialize 
'matrixElement' with
                    //^ a redundant value.
                    }

                    this.result[((result.length-1)-loops)-1] = matrixElement;
                    //^ immediately write the new element value to the array of int 
arrays (concurrently)

                    if (loops < 0){ break; }
                    //^ if job has finished creating its part of the result
                }
            }
            this.done = true;
            //^ job finished
            return this.result;
            //^ store result for worker thread to fetch
        }
        private int element(int[][] kernel, int[][] matrixInput, int yAxis, int xAxis) {
            int total = 0;
            //^ set-up
            //: apply kernel to the element with the element's neighbors
            for (int yAxisKernel = -1; yAxisKernel < kernel.length-1; yAxisKernel++) {
                for (int xAxisKernel = -1; xAxisKernel < kernel.length-1; xAxisKernel++) {
                    try { total += kernel[yAxisKernel+1][xAxisKernel+1] * 
matrixInput[yAxis+yAxisKernel][xAxis+xAxisKernel]; }
                    catch(ArrayIndexOutOfBoundsException error) { }
```

```
                            //^ works as zero-padding by not adding operations that include out-
of-bounds
                            //^ indexes as that will be that same as adding zero.
                        }
                }
                return total;
        }
        @Override
    public int[][] deserializeIntArrArr(byte[] data){
                //^ has to be public because it is part of the 'Task3Job' interface
        //* to deserialize the raw data int[][] to operate on it
         ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
         //^ holds and formats the byte data of the argument
         ObjectInputStream inStream;
         //^ 'ObjectOutputStream' serialize and write objects.
         int[][] deserialized = null;
         //^ initialized as null because of the try-statement but null should never be returned

         try {
                //^ try-statement to deal with potential IOException or ClassNotFoundException
errors (mandated by compiler)
                 inStream = new ObjectInputStream(byteInStream);
                 //^ 'ObjectInputStream' initialized for deserialization and writing to
'byteInStream'
                 deserialized = (int[][]) inStream.readObject();
                 //^ deserialize the byte array argument and writes it to 'byteOutStream'.
                 //^ the only code in this method that can potentially cause a
'ClassNotFoundException' error.
                 inStream.close();
                 //^ closes initialization to free up resources
         }
         catch (IOException|ClassNotFoundException e) {     e.printStackTrace(); }
         //^ for possible errors relating to deserialization, int[][] parsing, and the
'ObjectInputStream' instance

         return deserialized;
         //^ Returns a 2d int array containing the deserialized data
    }
}
```

Normalization job class

```
//: relevant imports
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class Task4JobNormalization implements Task4Job{
        private boolean done = false;
        //^ notify main thread when the job has finished
        private int[][] unformatted;
        //^ the matrix to apply the specified kernel filter on
        private int start;
        //^ to calculate the position of the first matrix element to apply the operation on.
        private int length;
        //^ tells the nested loop how many matrix elements to apply the operation on (is
calculated)
        private int[] result;
        //^ where the result of the job is stored to be later fetched by the worker node/thread

        public Task4JobNormalization(byte[] unformatted, int start, int length) {
                this.unformatted = this.deserializeIntArrArr(unformatted);
```

```
            //^ originally serialized it to prevent passing of references between
threads/nodes
            this.start = start;
            this.length = length;
            this.result = new int[length];
            //^ the result is based of how big of proportion of normalization we want this job
            //^ to do itself.
    }

    @Override
    public boolean isDone()/*getter method*/{ return this.done; }

    @Override
    public int[] perform(){
            //* to prevent the reference passed instead of the values (deep-copy)
            //:set-up
            boolean first = true;
            //^ for the stating index of the 2nd subject array and beyond
            int remaining = this.start;
            //: set up the starting index of the input (not all sub-arrays are always equal
lengths)
            int yAxis = 0;
            //^ the starting index of the array of int sub-arrays - calculated by "start" and
dimensions.
            //^ Assigned as zero for sake of initialization but the zero will not be read
until change.
            int xAxis;
            //^ the starting index of the selected int sub-array - calculated by "start" and
dimensions
            //^ (2nd dimensional index).
            int loops = this.length;
            //^ dictates how many elements left the job must operate on for its part of the
normalization

            //: calculate the index of the int array of int arrays (yAxis)
            for(int index = 0; index < this.unformatted.length; index++) {
                if (remaining <= this.unformatted[index].length-1) { yAxis = index;
break; }
                remaining -= (this.unformatted[index].length);
            }

            xAxis = remaining;
            //^ 2nd dimension index is based on 1st dimension (yAxis) calculation

            //: normalize part of the kernel's result based within specified range
            for (int y = yAxis; y < this.unformatted.length; y++) {
                if (loops == 0)/* thread finish its part of the matrix*/{ break; }
                if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis =
0;}
                if (first) { first = false; };
                //^ after this iteration, the current iteration no longer be the first

                for (int x = xAxis; x < this.unformatted[y].length; x++) {
                    //^ going via each element in current sub-array
                    loops--;
                    //^ one less element to process the current job's part of the
normalization
                    if (!( y == this.unformatted.length || x ==
this.unformatted[y].length )) {
                        //^ prevent out-of-bounds error when going though the jagged
array (as sub-array
                        //^ lengths are inconsistent and unknown).
```

```
                                this.result[(this.result.length-1)-loops] = 
this.unformatted[y][x];
                                //^ deep-copy the element's value to the array of atomic int 
arrays (concurrently)
                            }
                            else {
                                this.result[(this.result.length-1)-loops] = 0;
                                //^ We don't want resulting int[][] to also be jagged so we 
add placeholders as '0's
                            }
                            if (loops == 0){ break; }
                            //^ if job has finished creating its part of the result
                        }
                }

                this.done = true;
                //^ to tell the corresponding worker thread that the job has finished when it 
checks it
                return this.result;
                //^ stores result for the corresponding worker thread to fetch
        }
        @Override
    public int[][] deserializeIntArrArr(byte[] data){
                //^ has to be public because it is part of the 'Task3Job' interface
        //* to deserialize the raw data int[][] to operate on it
         ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
         //^ holds and formats the byte data of the argument
         ObjectInputStream inStream;
         //^ 'ObjectOutputStream' serialize and write objects.
         int[][] deserialized = null;
         //^ initialized as null because of the try-statement but null should never be returned

         try {
                //^ try-statement to deal with potential IOException or ClassNotFoundException 
errors (mandated by compiler)
                 inStream = new ObjectInputStream(byteInStream);
                 //^ 'ObjectInputStream' initialized for deserialization and writing to 
'byteInStream'
                 deserialized = (int[][]) inStream.readObject();
                 //^ deserialize the byte array argument and writes it to 'byteOutStream'.
                 //^ the only code in this method that can potentially cause a 
'ClassNotFoundException' error.
                 inStream.close();
                 //^ closes initialization to free up resources
         }
         catch (IOException|ClassNotFoundException e) {     e.printStackTrace(); }
         //^ for possible errors relating to deserialization, int[][] parsing, and the 
'ObjectInputStream' instance

         return deserialized;
         //^ Returns a 2d int array containing the deserialized data
    }
}
```

Poison pill class

```
public class Task4JobPoisonPill implements Task4Job{
        //^ implements 'Task4Job' so it can be stored in the thread pool's
        //^ FIFO job queue.
        private boolean done = false;
        //^ simple set-up

        @Override
```

```
        public boolean isDone() { return done; }
        //^ returns true when '.perform()' gets executed, otherwise return false

        @Override
        public int[] perform() { this.done = true; return new int[] { }; }
        //^ when sub-thread catches result of job, it breaks while-loop and
        //^ closes thread if result is an empty int array.
        //^ It returns int[] as the other jobs returns int[] as well, hence
        //^ why so does the method in the 'Task4Job' interface.

        @Override
        public int[][] deserializeIntArrArr(byte[] data) { return null;   }
        //^ this method will never be called but is there because deserialization
        //^ is one of the main parts of a job (hence is in the job interface),
        //^ deserialized data to produce results, the Poison Pill enters sub-thread
        //^ but unlike jobs to process as a job to stop the sub-thread's while-loop
        //^ (inside 'start' method) to close the sub-threads otherwise program will
        //^ have zombie threads.
}
```