

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: 7 th Jan 2024 14:00	EXPECTED FEEDBACK DATE: 31 st 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	4
Code explanation	4
Code's output.....	5
Speed	6
Task 2	6
Code explanation	6
Thread safety.....	13
Code's output.....	13
Speed comparison	15
Task 3	15
Code's explanation and justification.....	15
Thread safety.....	32
Code's output.....	33
Speed	35
Task 4	35
Defining node failure.....	35
Findings and fixes	37
Without error handling	38
Error handling fix.....	38
With error handling	45
Scalability	46
References.....	48
Appendixes	49
Code files used by multiple tasks	49
Golden Standard class	49
Utilities class.....	50
Code files exclusively used for task 1	52
Main code class.....	52
Code files exclusively used for task 2.....	53
Main code class.....	53
Worker thread class	55
Job class	55
Code files exclusively used for task 3.....	57
Main code class.....	57
Thread pool class.....	63
Worker node class	67
Job interface	69
Kernel application job class.....	69
Normalization job class.....	72
Poison pill class	74
Code files exclusively used for task 4.....	75
Main code class.....	75
Thread pool class.....	82
Worker node class	87
Job interface.....	89
Kernel application job class.....	90

U14437 – High Performance computing	Assessment 2 essay	
Normalization job class.....		92
Poison pill class		94

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:

```

1 public class Task1 {
2     public static void main(String[] args) {
3         //: Setting up by instantiating the classes
4         GoldenStandard goldenStandard = new GoldenStandard();
5         //^ to compare with tasks 2, 3, and 4
6         Utilities utilities = new Utilities();
7         //^ for printing a matrix and a generating matrix
8
9         int[][] matrix = utilities.matrixGen();
10        //^ generate the 10,000 by 10,000 random matrix
11        utilities.printMatrix(matrix, "Randomly generated matrix:");
12        //^ prints the initial matrix
13
14        System.out.println("Program is now doing the golden standard solution...");
15        int[][] matrixResult = goldenStandard.apply(matrix);
16        //^ applying one kernel filter one after the other (sharpen then edge-detection)
17        utilities.printMatrix(matrixResult, "Applying Sharpen kernel then Edge-Detection kernel:");
18        //^ prints the resulting matrix
19        System.out.println("...The golden standard solution has finished");
20    }
21 }

```

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:

```

15 public int[][] apply(int[][] matrixInput){
16     matrixInput = applyKernel(1, matrixInput);
17     //^ Apply the "Sharpen" kernel to the generated matrix
18     matrixInput = applyKernel(2, matrixInput);
19     //^ Apply the "Edge-Detection" kernel to the resulting
20     //^ matrix of the "Sharpen" kernel application.
21     return matrixInput;
22 }

```

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

```

24 public int[][] applyKernel(int kernelType, int[][] matrixInput) {
25     int[][] matrixResult = new int[matrixInput.length][matrixInput[0].length];
26     //^ set-up
27     //: apply kernel to each of the matrix elements
28     for (int yAxis = 0; yAxis < matrixInput.length; yAxis++) {
29         for (int xAxis = 0; xAxis < matrixInput[0].length; xAxis++) {
30             switch(kernelType) {
31                 case 1: matrixResult[yAxis][xAxis] = this.element(kernelSharpen, matrixInput, yAxis, xAxis); break;
32                 default: matrixResult[yAxis][xAxis] = this.element(kernelEdgeDetection, matrixInput, yAxis, xAxis); break;
33                 //^ when kernelType == 2
34             }
35         }
36     }
37     return matrixResult;
38 }

```

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:

```

2    //: set-up.
3    //: kernels declared in this way to allow other developers to easily manipulate its elements' value
4    private int[][] kernelSharpen = {
5        {0, -1, 0},
6        {-1, 5, -1},
7        {0, -1, 0}
8    };
9    private int[][] kernelEdgeDetection = {
10       //^ also called as "outline" kernel
11       {-1, -1, -1},
12       {-1, 8, -1},
13       {-1, -1, -1}
14    };

```

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:

```

39    private int element(int[][] kernel, int[][] matrixInput, int yAxis, int xAxis) {
40        int total = 0;
41        //^ set-up
42        //: apply kernel to the element with the element's neighbors
43        for (int yAxisKernel = -1; yAxisKernel < kernel.length-1; yAxisKernel++) {
44            for (int xAxisKernel = -1; xAxisKernel < kernel.length-1; xAxisKernel++) {
45                try { total += kernel[yAxisKernel+1][xAxisKernel+1] * matrixInput[yAxis+yAxisKernel][xAxis+xAxisKernel]; }
46                catch (ArrayIndexOutOfBoundsException error) { }
47                //^ works as zero-padding by not adding operations that include out-of-bounds
48                //^ indexes as that will be that same as adding zero.
49            }
50        }
51        return total;
52    }
53 }

```

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

```

17● public static void main(String[] args) {
18     int[][] matrix = Task2.utilities.matrixGen();
19     //^ generate the 10,000 by 10,000 random matrix
20     utilities.printMatrix(matrix, "Randomly generated matrix:");
21     //^ prints the initial matrix
22
23     //: set up the atomic two-dimensional int array fields
24     Task2.matrixResult2 = new int[matrix.length][matrix[0].length];
25     Task2.matrixResult2Sharpen = new int[matrix.length][matrix[0].length];
26
27     int[][] matrixResult = Task2.singleThreadedSolution(matrix, Task2.utilities);
28     //^ doing the golden standard (with execution duration) for comparison with the multi-threaded solution
29
30     //: threaded (task 2 solution)
31     System.out.println("Program is now doing the threaded solution...");
32     int threadCount = Task2.utilities.getProcessingEleCount();
33     //^ get the processor count from user (must be in range of 0 to maximum available processors)
34     long startTime = System.nanoTime();
35     //^ start execution time
36     Task2.multiThreadedSolution(matrix, threadCount);
37     //^ applying one kernel filter after the other (sharpen then edge-detection) but with parallel threads
38     Task2.utilities.executionTime(startTime);
39     //^ prints execution time of applying the kernel filters
40     utilities.printMatrix(Task2.matrixResult2, "multi threaded solution result:");
41     //^ prints the resulting matrix
42     System.out.println("...The threaded solution has finished");
43
44     System.out.println(Task2.utilities.compareMatrixes(matrixResult, Task2.matrixResult2));
45 }

```

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

```

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

```

The class's "multiThreadedSolution" method does the multi-threaded solution as seen:

```

46 private static void multiThreadedSolution(int[][] matrix, int threadCount){
47     //: set-up
48     int totalSize = matrix.length * matrix[0].length;
49     int sectorSize = totalSize / threadCount;
50     //^ calculate element count for each job distribution
51     Task2.threads = new Task2WorkerThread[threadCount];
52     //^ set-up the thread pool
53
54     for (int index = 0; index < Task2.threads.length; index++) {
55         Task2.threads[index] = new Task2WorkerThread();
56         Task2.threads[index].start();
57     }
58
59     multiThreadedJob(1, matrix, threadCount, sectorSize, totalSize);
60     //^ apply the sharpen kernel
61
62     //: deep-copy the result to pass the values, instead of reference, to the next job
63     Task2.toFormat = true;
64     multiThreadedJob(0, new int [][]{{}}, threadCount, sectorSize, totalSize);
65     //^ values of first 2 arguments do not matter as they will not be used
66     Task2.toFormat = false;
67
68     multiThreadedJob(2, Task2.matrixResult2Sharpen, threadCount, sectorSize, totalSize);
69     //^ apply the edge-detection kernel
70
71     for (int index = 0; index < Task2.threads.length; index++) { Task2.threads[index].done(); }
72     //^ acts like a Poison Pill
73     for (int index = 0; index < Task2.threads.length; index++) {
74         try { Task2.threads[index].join(); } catch (InterruptedException e) { e.printStackTrace(); }
75     }
76 }

```

It calls the “multiThreadedJob” method to pass the jobs as seen:

```

46 private static void multiThreadedSolution(int[][] matrix, int threadCount){
47     //: set-up
48     int totalSize = matrix.length * matrix[0].length;
49     int sectorSize = totalSize / threadCount;
50     //^ calculate element count for each job distribution
51     Task2.threads = new Task2WorkerThread[threadCount];
52     //^ set-up the thread pool
53
54     for (int index = 0; index < Task2.threads.length; index++) {
55         Task2.threads[index] = new Task2WorkerThread();
56         Task2.threads[index].start();
57     }
58
59     multiThreadedJob(1, matrix, threadCount, sectorSize, totalSize);
60     //^ apply the sharpen kernel
61
62     //: deep-copy the result to pass the values, instead of reference, to the next job
63     Task2.toFormat = true;
64     multiThreadedJob(0, new int [][]{{}}, threadCount, sectorSize, totalSize);
65     //^ values of first 2 arguments do not matter as they will not be used
66     Task2.toFormat = false;
67
68     multiThreadedJob(2, Task2.matrixResult2Sharpen, threadCount, sectorSize, totalSize);
69     //^ apply the edge-detection kernel
70
71     for (int index = 0; index < Task2.threads.length; index++) { Task2.threads[index].done(); }
72     for (int index = 0; index < Task2.threads.length; index++) {
73         try { Task2.threads[index].join(); } catch (InterruptedException e) { e.printStackTrace(); }
74     }
75 }

```

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


```

1 public class Task2Job{
2     //: set-up.
3     //: kernels declared in this way to allow other developers to easily manipulate its elements' value
4     private int[][] kernelSharpen = {
5         {0, -1, 0},
6         {-1, 5, -1},
7         {0, -1, 0}
8     };
9     private int[][] kernelEdgeDetection = {
10        {-1, -1, -1},
11        {-1, 8, -1},
12        {-1, -1, -1}
13    };
14
15    private boolean done = false;
16    //^ notify main thread when the job has finished
17    private int kernelType;
18    //^ determines which kernel filter (Sharpen or Edge-Detection) will be used
19    private int[][] matrixInput;
20    //^ the matrix to apply the specified kernel filter on
21    private int start;
22    //^ to calculate the position of the first matrix element to apply the operation on.
23    private int end;
24    //^ to calculate the position of the last matrix element to apply the operation on.
25    private int loops;
26    //^ tells the nested loop how many matrix elements to apply the operation on (is calculated)
27    private int xAxis;
28    //^ the index of the array of int sub-arrays - calculated by "start" and matrix dimensions
29    private int yAxis;
30    //^ the index of the selected sub-array - calculated by "start" and dimensions
31
32    public Task2Job(int kernelType, int[][] matrixInput, int start, int end) {
33        this.kernelType = kernelType;
34        this.matrixInput = matrixInput;
35        this.start = start;
36        this.end = end;
37        this.loops = this.end - this.start;
38        this.yAxis = start / Task2.matrixResult2.length;
39        this.xAxis = start % Task2.matrixResult2.length;
40    }

```

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

```
1 public class Task2WorkerThread extends Thread{
2     private Task2Job job;
3     private boolean jobDone;
4     private boolean running;
5
6     public void setJob(Task2Job job) {
7         //^ called by main thread when a new job needs doing
8         this.job = job;
9         this.jobDone = false;
10    }
11    public void done() { this.running = false; }
12    //^ when thread is no longer needed - gets called by main thread
13    public boolean isJobDone() { return this.jobDone; }
14    //^ when thread finished current job - gets called by main thread
15    public void run() {
16        this.running = true;
17        while (this.running) {
18            if (job != null && !job.isDone()) {
19                //^ execute when there is an existing job that is not completed
20                this.job.perform();
21                //^ where the job actually gets done
22                this.jobDone = true;
23                this.job = null;
24            }
25            try { Thread.sleep(10); } catch (InterruptedException e) { }
26            //^ to avoid busy-waiting
27        }
28    }
29 }
```

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

```

42● public void perform() {
43     if(Task2.toFormat) { this.deepCopyMatrix(Task2.matrixResult2, Task2.matrixResult2Sharpen); return; }
44     //: set-up
45     int matrixElement;
46     int loops = this.loops;
47     boolean first = true;
48     //: set up the starting coords of the matrix
49     int yAxis = this.yAxis;
50     int xAxis = this.xAxis;
51
52     //: apply kernel to each of the matrix elements
53     for (int y = yAxis; y < this.matrixInput.length; y++) {
54         if (loops < 0)/* thread finish its part of the matrix*/{ break; }
55         if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis = 0;}
56         first = !first;
57
58         for (int x = xAxis; x < this.matrixInput[0].length; x++) {
59             loops--;
60             switch(this.kernelType) {
61                 case 1: matrixElement = this.element(kernelSharpen, this.matrixInput, y, x); break;
62                 default: matrixElement = this.element(kernelEdgeDetection, this.matrixInput, y, x); break;
63                 //^ when kernelType == 2
64
65             }
66
67             Task2.matrixResult2[y][x] = matrixElement;
68             //^ immediately write the new element value to the array of int arrays (concurrently)
69
70             if (loops < 0){ break; }
71             //^ if job has finished creating its part of the result
72         }
73     }
74 }

```

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

```

65● private int element(int[][] kernel, int[][] matrixInput, int yAxis, int xAxis) {
66     int total = 0;
67     //^ set-up
68     //: apply kernel to the element with the element's neighbors
69     for (int yAxisKernel = -1; yAxisKernel < kernel.length-1; yAxisKernel++) {
70         for (int xAxisKernel = -1; xAxisKernel < kernel.length-1; xAxisKernel++) {
71             try { total += kernel[yAxisKernel+1][xAxisKernel+1] * matrixInput[yAxis+yAxisKernel][xAxis+xAxisKernel]; }
72             catch(ArrayIndexOutOfBoundsException error) { }
73             //^ works as zero-padding by not adding operations that include out-of-bounds
74             //^ indexes as that will be that same as adding zero.
75         }
76     }
77     return total;
78 }

```

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

```

93 private void deepCopyMatrix(int[][] matrixCopy, int[][] matrixPaste){
94     /* to prevent the reference passed instead of the values (deep-copy)
95     //:set-up
96     int loops = this.loops;
97     boolean first = true;
98     //: set up the starting coords of the matrix
99     int yAxis = this.yAxis;
100    int xAxis = this.xAxis;
101
102    //: deep-copy each of the matrix elements within specified range
103    for (int y = yAxis; y < matrixCopy.length; y++) {
104        if (loops < 0)/* thread finish its part of the matrix*/{ break; }
105        if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis = 0;}
106        first = !first;
107
108        for (int x = xAxis; x < matrixCopy[0].length; x++) {
109            loops--;
110            matrixPaste[y][x] = matrixCopy[y][x];
111            //^ DEEP-copy the element's value to the array of atomic int arrays (concurrently)
112
113            if (loops < 0){ break; }
114            //^ if job has finished creating its part of the result
115        }
116    }
117    this.done = true;
118 }
119 }

```

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

```

83 private static int[][] singleThreadedSolution(int[][] matrix, Utilities utilities)/*golden standard*/{
84     GoldenStandard goldenStandard = new GoldenStandard();
85     //^ instantiating the class that does the golden standard solution
86
87     System.out.println("Program is now doing the golden standard solution...");
88     long startTime = System.nanoTime();
89     //^ start execution time
90     int[][] matrixResult = goldenStandard.apply(matrix);
91     //^ applying one kernel filter after the other (sharpen then edge-detection)
92     utilities.executionTime(startTime);
93     //^ prints execution time of applying the kernel filters
94     //utilities.printMatrix(matrixResult, "golden standard solution result:");
95     //^ prints the resulting matrix
96     System.out.println("...The golden standard solution has finished");
97
98     return matrixResult;
99 }
100 }

```

Matrixes are compared by calling “utilities.compareMatrixies” as seen:

```

43 public String compareMatrixies(int[][] matrix1, int[][] matrix2) {
44     int[] result = sameMatrixies(matrix1, matrix2);
45     //^ set-up
46     if (result[0] == -2) {return "non-identical matrixes - matrix dimensions are different";}
47     else if (result[0] == -1) {return "both matrixes are identical in both: dimensions' value";}
48     else {return String.format("non-identical matrixes - atleast one element's value is unequal. First unequal at [%d][%d]",result[0],result[1]);}
49 }

```

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:

```

20● private int[] sameMatrixes(int[][] matrix1, int[][] matrix2) {
21     int [][] matrixResult = new int[matrix1.length][matrix1[0].length];
22     //^ set-up for the subtraction proof
23     /* {-1,0} means same, otherwise matrixes are not the same.
24     /* {-2,0} means different dimensions.
25     //: subtraction proof
26     if ( (matrix1.length != matrix2.length) || (matrix1[0].length != matrix2[0].length) ) { return new int[] {-2,0}; }
27     //^ check dimensions
28     for (int yAxis = 0; yAxis < matrix1.length; yAxis++) {
29         for (int xAxis = 0; xAxis < matrix1[0].length; xAxis++) {
30             matrixResult[yAxis][xAxis] = matrix1[yAxis][xAxis] - matrix2[yAxis][xAxis];
31             //^ subtract each individual element value from the corresponding one from the other matrix at a time
32         }
33     }
34     printMatrix(matrixResult, "subtraction proof (a simple subtraction between both matrixes:)");
35     //^ print the subtraction proof
36
37     //: check the subtraction proof
38     for (int yAxis = 0; yAxis < matrix1.length; yAxis++) {
39         for (int xAxis = 0; xAxis < matrix1[0].length; xAxis++) {
40             if (matrixResult[yAxis][xAxis] != 0) { return new int[] {yAxis,xAxis}; }
41             //^ return location of first non-identical element value
42         }
43     }
44     return new int[] {-1,0};
45     //^ executes if matrixes are identical
46 }

```

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: VW
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: VW
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]
[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.

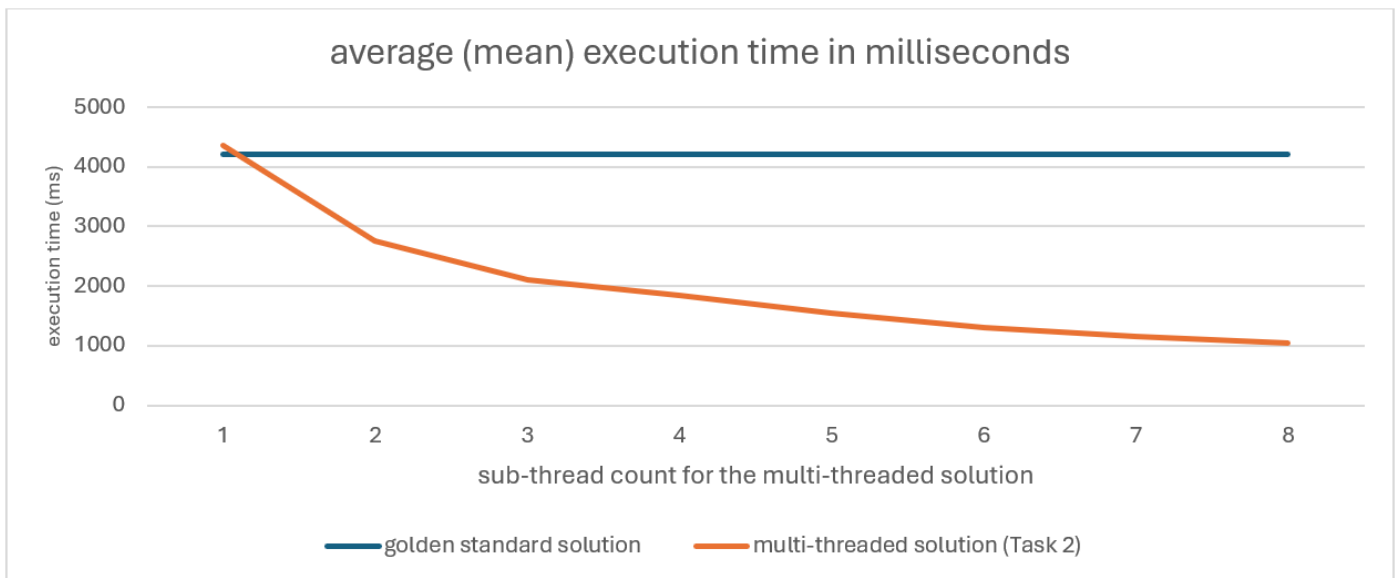
Speed comparison

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 (Delparte-Gallet et al., 2003) in to in favour with the Message-Passing Model with the Message Passing Interface (MPI) (Delparte-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 (Delparte-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 stored 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).

```

1 import java.io.Serializable;
2 //^ For 'Task3Job' to implement 'Serializable'
3
4 public interface Task3Job extends Serializable{
5     //^ any class implementing this interface also implements 'Serializable' so the classes'
6     //^ instances are able to get serialized and deserialized.
7     /* Interface allows the worker thread to accept any kind of job (deep copy and kernel application).
8     /* Implemented by the 'Task3JobKernelApplication', 'Task3JobNormalization', and
9     /* 'Task3JobPoisonPill' classes.
10    boolean isDone();
11    //^ allows sub-thread to check if the job has finished yet (returns true if '.perform()' finishes)
12    int[] perform();
13    //^ Executes the main body (in the 'perform' public method) of the job in the sub-thread in
14    //^ concurrency.
15    int[][] deserializeIntArrArr(byte[] data);
16    //^ Deserialises the 2d int array.
17    //^ Due to the argument being serialized, reference calls are encapsulated inside of the
18    //^ the method itself meaning that no "pass by reference" is done over the threads/nodes
19    //^ in the computer network in the method's body.
20    //^ Despite byte[] being an object, in a network simulation (Thread-per-Node model),
21    //^ byte[] serves container for raw data for Messaging Passing Interface (MPI), rather
22    //^ than a reference to the original data structure - thus the method call also does
23    //^ not pass any direct reference calls between threads.
24 }

```

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:


```

1  //: all relevant imports
2  import java.io.ByteArrayInputStream;
3  import java.io.ByteArrayOutputStream;
4  import java.io.IOException;
5  import java.io.ObjectInputStream;
6  import java.io.ObjectOutputStream;
7  import java.util.ArrayList;
8
9  public class Task3ThreadPool {
10     /** thread pool is used to a Thread-per-Node model (HPC cluster of computer nodes).
11
12     private boolean sleep;
13     /** stops the thread pool from searching for threads to
14     /** message the jobs to when there are no jobs left.
15     private Task3WorkerNode[] threads;
16     /** where the worker threads are referenced.
17     /** Each thread ID will be their index in the array
18     private ArrayList<Task3Job> jobQueue;
19     /** FIFO stack structure (first in first out).
20     /** Contains the jobs as messages.
21     private int[][] threadJobResults;
22     /** job results that are concurrently written to by the threads (atomic).
23     /** Each thread is to write to corresponding element, even if overwriting,
24     /** thus the index serves as the thread's ID (identification).
25     private int maxJobs;
26     /** For code integrity purposes - if another developer changes initial
27     /** matrix size to a small size it will still work (STILL MUST BE ATLEAST 1x1).
28     public Task3ThreadPool(int threadCount) {
29         //: set-up the fields
30         this.sleep = false;
31         /** activate the thread pool
32         jobQueue = new ArrayList<Task3Job>();
33         /** for good practice and scalability - the data structure has dynamic size
34         threadJobResults = new int[threadCount][];
35         /** is a jagged array, because jobs' returned int arrays' length are not predefined
36
37         //: set-up and fill the thread pool
38         this.threads = new Task3WorkerNode[threadCount];
39         for (int index = 0; index < threadCount; index++) {
40             this.threads[index] = new Task3WorkerNode();
41             /** worker thread instance
42             this.threads[index].start();
43             /** start the thread's execution
44         }
45     }

```

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:

```

47  //: public methods that are called by main ('Task3') class
48  public int threadCount() /*getter method*/ { return this.threads.length; }
49  public void wakeUp() { this.start(); }
50  //^ notifies the thread pool - simulates the ".notify()" method call
51  public void enqueueJob(Task3Job job) /*add method (FIFO style)*/ { jobQueue.add(job); }
52  //^ enqueues a job by appending it to the end of the stack
53  public byte[] getResults() /*getter method*/ { return this.serializeIntArrArr(this.threadJobResults); }
54  public void clear() /*reset method*/ {
55      //^ not exactly aligned with the Thread-per-Node Model but the program must not exceed the Java
56      //^ default maximum heap memory limit, which cannot be changed in-program.
57      //^ The lower it is, the better the program's scalability.
58      //: these data are, at this point of time, redundant so they must be deleted
59      this.threadJobResults = new int[this.threads.length][];
60      for (int index = 0; index < this.threads.length; index++) { threads[index].clear(); }
61      //^ clears each worker node's stored result
62      try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
63      //^ to impose a delay of 200ms to simulate the speeds of modern computers and cluster networks.
64      //^ Tells all threads to delete their now redundant stored result.
65  }

```

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:

```

118  private void start() {
119      //^ called by ".wakeUp()" by other classes.
120      //^ Assign jobs and gather results.
121      this.maxJobs = jobQueue.size();
122      this.sleep = false;
123      while (!this.sleep) {
124          try { Thread.sleep(1); } catch (InterruptedException e) { e.printStackTrace(); }
125          //^ to avoid busy-waiting
126          for (int index = 0; index < this.maxJobs; index++) {
127              //^ assign a job per worker thread
128              if (this.jobQueue.size() == 0) { break; }
129              //^ cannot assign a non-existent job to a worker thread
130              threads[index].setJob(this.serializeJob(this.dequeueJob()));
131              //^ take from FIFO stack and into a serialized MPI message to the worker thread
132          }
133          try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
134          //^ to impose a delay of 200ms for each computational job executed to simulate the
135          //^ speeds of modern computers and cluster networks.
136          //^ this node-to-node message is the serialized job from the main thread to the sub-thread.
137          //^ Thread-pool can send multiple messages without waiting for delay of the sub-thread
138          //^ receiving the message.
139          if (this.jobQueue.size() == 0) { this.sleep = true; }
140          //^ go to sleep when there are currently no more jobs
141      }
142      this.requestResults();
143      //^ after assigning jobs, we gather all results when all threads finish their given jobs
144  }

```

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

```

76 private void requestResults() {
77     boolean allResults = false;
78     //^ dictates if it thread pool should keep waiting for all results or not
79     boolean[] alldone = new boolean[this.maxJobs];
80     //^ only fetch results from the relevant threads to save processing time
81     while (!allResults) {
82         //^ far faster than the alternative of constantly checking a sub-thread job until it
83         //^ is done one at a time.
84         for (int index = 0; index < this.maxJobs; index++) {
85             //^ get all results to if each thread has finished the job or not
86             alldone[index] = threads[index].isJobDone();
87             //^ store the each job status
88         }
89         try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
90         //^ to impose a delay of 200ms for each computational job executed to simulate the
91         //^ speeds of modern computers and cluster networks.
92         //^ These node-to-node messages are to check if the all sub-thread has finished
93         //^ their given jobs.
94         allResults = true;
95         //^ Assume that all threads have finished
96         for (int index = 0; index < this.maxJobs; index++) {
97             //^ only gathers threads that were given relevant jobs
98             if(alldone[index] == false) {
99                 //^ current thread hasn't finished?
100                 allResults = false;
101                 //^ if one thread isn't finished then all threads are not finished!
102                 break;
103                 //^ no point to continue so break off to save computational time
104             }
105         }
106     }
107 }
108 for (int index = 0; index < this.maxJobs; index++) {
109     //^ gather results of all relevant threads
110     threadJobResults[index] = this.deserializeIntArr(this.threads[index].requestResult());
111     //^ deserialize raw data MPI message into the fetched int[] result to be stored
112 }
113 try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
114 //^ to impose a delay of 200ms for each computational job executed to simulate the
115 //^ speeds of modern computers and cluster networks.
116 //^ These node-to-node messages are to get the serialized results from the sub-threads.
117 }

```

When passing jobs, it first serialises them as seen:

```

145 private byte[] serializeJob(Task3Job job){
146     //^ between thread pool and worker nodes/threads - to reduce reference calls as much as possible.
147     //^ To send serialized copies so referencing between nodes are limited to twice per communication/message.
148     ByteArrayOutputStream byteOutputStream = new ByteArrayOutputStream();
149     //^ holds and formats the byte data of the argument
150     ObjectOutputStream outputStream = null;
151     //^ 'ObjectOutputStream' serialize and write objects.
152     //^ initialized as null because of the try-statement but null should never used
153
154     try {
155         //^ try-statement to deal with possible IOException errors (mandated by compiler)
156         outputStream = new ObjectOutputStream(byteOutputStream);
157         //^ 'ObjectOutputStream' initialized for serialization and writing to 'byteOutputStream'
158         outputStream.writeObject(job);
159         //^ serialize the argument and writes it to 'byteOutputStream'
160         outputStream.close();
161         //^ closes initialization to free up resources
162     }
163     catch (IOException e) { e.printStackTrace(); }
164     //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
165
166     return byteOutputStream.toByteArray();
167     //^ Returns a byte array containing the serialized data
168 }

```

It also deserialize the job's results as seen:

```

169 private int[] deserializeIntArr(byte[] data){
170     ByteArrayInputStream byteInputStream = new ByteArrayInputStream(data);
171     //^ holds and formats the byte data of the argument.
172     ObjectInputStream inStream;
173     //^ 'ObjectOutputStream' serialize and write objects.
174     int[] deserialized = null;
175     //^ initialized as null because of the try-statement but null should never be returned
176
177     try {
178         //^ try-statement to deal with potential IOException or ClassNotFoundException errors (mandated by compiler)
179         inStream = new ObjectInputStream(byteInputStream);
180         //^ 'ObjectInputStream' initialized for deserialization and writing to 'byteInputStream'
181         deserialized = (int[]) inStream.readObject();
182         //^ deserializes the byte array argument and writes it to 'byteOutputStream'.
183         //^ the only code in this method that can potentially cause a 'ClassNotFoundException' error.
184         inStream.close();
185         //^ closes initialization to free up resources
186     }
187     catch (IOException|ClassNotFoundException e) { e.printStackTrace(); }
188     //^ for possible errors relating to deserialization, int[] parsing, and the 'ObjectInputStream' instance
189
190     return deserialized;
191     //^ Returns a int array containing the deserialized data
192 }

```

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 by reference, it does it only for deep copying as seen:

```

181 private byte[] serializeIntArrArr(int[][] array){
182     //^ between thread pool and main - same thread but still used for deep-copying
183     ByteArrayOutputStream byteOutputStream = new ByteArrayOutputStream();
184     //^ holds and formats the byte data of the argument
185     ObjectOutputStream outStream = null;
186     //^ 'ObjectOutputStream' serialize and write objects.
187     //^ initialized as null because of the try-statement but null should never be used
188
189     try {
190         //^ try-statement to deal with possible IOException errors (mandated by compiler)
191         outStream = new ObjectOutputStream(byteOutputStream);
192         //^ 'ObjectOutputStream' initialized for serialization and writing to 'byteOutputStream'
193         outStream.writeObject(array);
194         //^ serialize the 2d int array argument and writes it to 'byteOutputStream'
195         outStream.close();
196         //^ closes initialization to free up resources
197     }
198     catch (IOException e) { e.printStackTrace(); }
199     //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
200
201     return byteOutputStream.toByteArray();
202     //^ Returns a byte array containing the serialized data
203 }
204 }

```

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):

```

1 //: import for serialization.
2 //: As compute nodes have copies of the built-in libraries, this is not shared
3 //: memory - sticking to the Thread-per-Node model (HPC cluster simulation).
4 import java.io.ByteArrayInputStream;
5 import java.io.ByteArrayOutputStream;
6 import java.io.IOException;
7 import java.io.ObjectInputStream;
8 import java.io.ObjectOutputStream;
9
10 public class Task3WorkerNode extends Thread {
11     /* simulates an individual compute node in a HPC cluster in accordance to
12     /* the Thread-per-Node model.
13     private Task3Job job;
14     /* stores the current job
15     private boolean jobDone = true;
16     /* toggles when either a new job is given or the current job has finished
17     private boolean running;
18     /* acts as a Poison Pill
19     private int[] jobResult;
20     /* result of current job is caught and stored here
21
22     public void setJob(byte[] job) {
23         /* called by main thread when a new job needs doing
24         this.job = this.deserializeJob(job);
25         this.jobDone = false;
26     }
27     public void done()/*setter method*/{ this.running = false; }
28     /* when thread is no longer needed - gets called by main thread
29     public boolean isJobDone()/*getter method*/{ return this.jobDone; }
30     /* when thread finished current job - gets called by main thread
31     public byte[] requestResult() { return this.serializeIntArr(this.jobResult); }
32     /* in compute clusters, results are stored in the worker nodes before retrieval
33     public void clear() { this.jobResult = null; }
34     /* 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:

```

36     public void run() {
37         this.running = true;
38         /* dictates if the thread should continue running or end
39         while (this.running) {
40             /* while-loop runs constantly till thread must end
41             if (job != null && !this.jobDone) {
42                 /* execute when there is an existing job that is not completed
43                 jobResult = this.job.perform();
44                 /* where the job actually gets done
45                 if (jobResult.length == 0 ) { this.running = false; }
46                 /* shuts down thread (after this loop) if job is a Poison Pill
47                 this.jobDone = true;
48                 /* prevents node from repeating the same job twice (clears redundancy)
49                 this.job = null;
50                 /* prevents node from running empty code before first job
51             }
52             try { Thread.sleep(1); } catch (InterruptedException e) { }
53             /* to avoid busy-waiting
54         }
55     }

```

Despite the methods not being called outside class, it remains public because so does the "Thread" 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:

```

56 private byte[] serializeIntArr(int[] array){
57     /* to send the result as a serialized raw data message instead of pass by reference
58     ByteArrayOutputStream byteOutputStream = new ByteArrayOutputStream();
59     //^ holds and formats the byte data of the argument
60     ObjectOutputStream outputStream = null;
61     //^ 'ObjectOutputStream' serialize and write objects.
62     //^ initialized as null because of the try-statement but null should never used
63
64     try {
65         //^ try-statement to deal with possible IOException errors (mandated by compiler)
66         outputStream = new ObjectOutputStream(byteOutputStream);
67         //^ 'ObjectOutputStream' initialized for serialization and writing to 'byteOutputStream'
68         outputStream.writeObject(array);
69         //^ serialize the argument and writes it to 'byteOutputStream'
70         outputStream.close();
71         //^ closes initialization to free up resources
72     }
73     catch (IOException e) { e.printStackTrace(); }
74     //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
75     return byteOutputStream.toByteArray();
76     //^ Returns a byte array containing the serialized data
77 }
78 private Task3Job deserializeJob(byte[] data){
79     /* to deserialize the raw data message into a job
80     ByteArrayInputStream byteInputStream = new ByteArrayInputStream(data);
81     //^ holds and formats the byte data of the argument
82     ObjectInputStream inputStream;
83     //^ 'ObjectOutputStream' serialize and write objects.
84     Task3Job deserialized = null;
85     //^ initialized as null because of the try-statement but null should never be returned
86
87     try {
88         //^ try-statement to deal with potential IOException or ClassNotFoundException errors (mandated by compiler)
89         inputStream = new ObjectInputStream(byteInputStream);
90         //^ 'ObjectInputStream' initialized for deserialization and writing to 'byteInputStream'
91         deserialized = (Task3Job) inputStream.readObject();
92         //^ deserialize the byte array argument and writes it to 'byteOutputStream'.
93         //^ the only code in this method that can potentially cause a 'ClassNotFoundException' error.
94         inputStream.close();
95         //^ closes initialization to free up resources
96     }
97     catch (IOException|ClassNotFoundException e) { e.printStackTrace(); }
98     //^ for possible errors relating to deserialization, job parsing, and the 'ObjectInputStream' instance
99     return deserialized;
100    //^ Returns a job containing the deserialized data
101 }
102 }

```

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

Kenel application:


```

1  //: imports
2  import java.io.ByteArrayInputStream;
3  import java.io.IOException;
4  import java.io.ObjectInputStream;
5
6  public class Task3JobKernelApplication implements Task3Job {
7      //: set-up.
8      //: Kernels declared in this way to allow other developers to easily manipulate its elements' value
9      //: and program will still work as intended.
10     private int[][] kernelSharpen = {
11         /* element's values can be changed if wished by a developer
12         {0, -1, 0},
13         {-1, 5, -1},
14         {0, -1, 0}
15     };
16     private int[][] kernelEdgeDetection = {
17         /* element's values can be changed if wished by a developer
18         {-1, -1, -1},
19         {-1, 8, -1},
20         {-1, -1, -1}
21     };
22
23     private boolean done = false;
24     //^ notify main thread when the job has finished
25     private int kernelType;
26     //^ determines which kernel filter (Sharpen or Edge-Detection) will be used
27     private int[][] matrixInput;
28     //^ the matrix to apply the specified kernel filter on
29     private int start;
30     //^ to calculate the position of the first matrix element to apply the operation on.
31     private int end;
32     //^ to calculate the position of the last matrix element to apply the operation on.
33     private int[] result;
34     //^ where the result of the job is stored to be later fetched by the worker node/thread
35
36     public Task3JobKernelApplication(int kernelType, byte[] matrixInput, int start, int end) {
37         this.kernelType = kernelType;
38         this.matrixInput = this.deserializeIntArray(matrixInput);
39         this.start = start;
40         this.end = end;
41     }
42     @Override
43     public boolean isDone()/*getter method*/{ return this.done; }

```

```

45     public int[] perform() {
46         //: set-up
47         int matrixElement;
48         //^ where the result of applying kernel, on a element, is temporarily stored
49         int loops = this.end - this.start;
50         //^ tells the nested loop how many matrix elements to apply the operation on (is calculated)
51         this.result = new int[loops+1];
52         //^ field assignment is based on local variable, hence it is here instead of in the constructor
53         boolean first = true;
54         //: set up the starting coords of the matrix
55         int yAxis = start / this.matrixInput[0].length;
56         //^ the index of the array of int sub-arrays - calculated by "start" and matrix dimensions
57         int xAxis = start % this.matrixInput[0].length;
58         //^ the index of the selected sub-array - calculated by "start" and dimensions
59
60         //: apply kernel to each of the matrix elements
61         for (int y = yAxis; y < this.matrixInput.length; y++) {
62             if (loops < 0)/* thread finish its part of the matrix*/{ break; }
63             if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis = 0;}
64             first = !first;
65             //^ no longer the first applied sub-array
66
67             for (int x = xAxis; x < this.matrixInput[0].length; x++) {
68                 loops--;
69                 //: applies kernel of element depending on which kernel is selected.
70                 //: Didn't use enumerations (enum) because it will make the serialization process, to
71                 //: prevent cross-node reference passes, more complex and takes more
72                 //: computational time.
73                 switch(this.kernelType) {
74                     case 1: matrixElement = this.element(kernelSharpen, this.matrixInput, y, x); break;
75                     //^ '1' corresponds to applying the Sharpen kernel
76                     default: matrixElement = this.element(kernelEdgeDetection, this.matrixInput, y, x); break;
77                     //^ when kernelType == 2.
78                     //^ '2' corresponds to applying the Edge-Detection kernel.
79                     //^ using default satisfy compiler so I do not need to initialize 'matrixElement' with
80                     //^ a redundant value.
81                 }
82
83                 this.result[((result.length-1)-loops)-1] = matrixElement;
84                 //^ immediately write the new element value to the array of int arrays (concurrently)
85
86                 if (loops < 0){ break; }
87                 //^ if job has finished creating its part of the result
88             }
89         }
90         this.done = true;
91         //^ job finished
92         return this.result;
93         //^ store result for worker thread to fetch
94     }

```



```

95 private int element(int[][] kernel, int[][] matrixInput, int yAxis, int xAxis) {
96     int total = 0;
97     //^ set-up
98     //: apply kernel to the element with the element's neighbors
99     for (int yAxisKernel = -1; yAxisKernel < kernel.length-1; yAxisKernel++) {
100         for (int xAxisKernel = -1; xAxisKernel < kernel.length-1; xAxisKernel++) {
101             try { total += kernel[yAxisKernel+1][xAxisKernel+1] * matrixInput[yAxis+yAxisKernel][xAxis+xAxisKernel]; }
102             catch(ArrayIndexOutOfBoundsException error) { }
103             //^ works as zero-padding by not adding operations that include out-of-bounds
104             //^ indexes as that will be that same as adding zero.
105         }
106     }
107     return total;
108 }
109 @Override
110 public int[][] deserializeIntArrArr(byte[] data){
111     /* to deserialize the raw data int[][] to operate on it
112     ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
113     //^ holds and formats the byte data of the argument
114     ObjectInputStream inStream;
115     //^ 'ObjectOutputStream' serialize and write objects.
116     int[][] deserialized = null;
117     //^ initialized as null because of the try-statement but null should never be returned
118
119     try {
120         //^ try-statement to deal with potential IOException or ClassNotFoundException errors (mandated by compiler)
121         inStream = new ObjectInputStream(byteInStream);
122         //^ 'ObjectInputStream' initialized for deserialization and writing to 'byteInStream'
123         deserialized = (int[][]) inStream.readObject();
124         //^ deserialize the byte array argument and writes it to 'byteOutStream'.
125         //^ the only code in this method that can potentially cause a 'ClassNotFoundException' error.
126         inStream.close();
127         //^ closes initialization to free up resources
128     }
129     catch (IOException|ClassNotFoundException e) { e.printStackTrace(); }
130     //^ for possible errors relating to deserialization, int[][] parsing, and the 'ObjectInputStream' instance
131
132     return deserialized;
133     //^ Returns a 2d int array containing the deserialized data
134 }
135 }

```

Normalization:

```

1 //: relevant imports
2 import java.io.ByteArrayInputStream;
3 import java.io.IOException;
4 import java.io.ObjectInputStream;
5
6 public class Task3JobNormalization implements Task3Job{
7     private boolean done = false;
8     //^ notify main thread when the job has finished
9     private int[][] unformatted;
10    //^ the matrix to apply the specified kernel filter on
11    private int start;
12    //^ to calculate the position of the first matrix element to apply the operation on.
13    private int length;
14    //^ tells the nested loop how many matrix elements to apply the operation on (is calculated)
15    private int[] result;
16    //^ where the result of the job is stored to be later fetched by the worker node/thread
17
18    public Task3JobNormalization(byte[] unformatted, int start, int length) {
19        this.unformatted = this.deserializeIntArrArr(unformatted);
20        //^ originally serialized it to prevent passing of references between threads/nodes
21        this.start = start;
22        this.length = length;
23        this.result = new int[length];
24        //^ the result is based of how big of proportion of normalization we want this job
25        //^ to do itself.
26    }
27
28    @Override
29    public boolean isDone()/*getter method*/{ return this.done; }
30

```

```

31● @Override
32 public int[] perform(){
33     /* to prevent the reference passed instead of the values (deep-copy)
34     //:set-up
35     boolean first = true;
36     //^ for the starting index of the 2nd subject array and beyond
37     int remaining = this.start;
38     //: set up the starting index of the input (not all sub-arrays are always equal lengths)
39     int yAxis = 0;
40     //^ the starting index of the array of int sub-arrays - calculated by "start" and dimensions.
41     //^ Assigned as zero for sake of initialization but the zero will not be read until change.
42     int xAxis;
43     //^ the starting index of the selected int sub-array - calculated by "start" and dimensions
44     //^ (2nd dimensional index).
45     int loops = this.length;
46     //^ dictates how many elements left the job must operate on for its part of the normalization
47
48     //: calculate the index of the int array of int arrays (yAxis)
49     for(int index = 0; index < this.unformatted.length; index++) {
50         if (remaining <= this.unformatted[index].length-1) { yAxis = index; break; }
51         remaining -= (this.unformatted[index].length);
52     }
53
54     xAxis = remaining;
55     //^ 2nd dimension index is based on 1st dimension (yAxis) calculation
56
57     //: normalize part of the kernel's result based within specified range
58     for (int y = yAxis; y < this.unformatted.length; y++) {
59         if (loops == 0)/* thread finish its part of the matrix*/{ break; }
60         if(!first)/* in next sub-arrays, always start with index of 0*/{ xAxis = 0;}
61         if (first) { first = false; };
62         //^ after this iteration, the current iteration no longer be the first
63
64         for (int x = xAxis; x < this.unformatted[y].length; x++) {
65             //^ going via each element in current sub-array
66             loops--;
67             //^ one less element to process the current job's part of the normalization
68             if (!(y == this.unformatted.length || x == this.unformatted[y].length)) {
69                 //^ prevent out-of-bounds error when going though the jagged array (as sub-array
70                 //^ lengths are inconsistent and unknown).
71                 this.result[(this.result.length-1)-loops] = this.unformatted[y][x];
72                 //^ deep-copy the element's value to the array of atomic int arrays (concurrently)
73             }
74             else {
75                 this.result[(this.result.length-1)-loops] = 0;
76                 //^ We don't want resulting int[][] to also be jagged so we add placeholders as '0's
77             }

```

```

78         if (loops == 0){ break; }
79         //^ if job has finished creating its part of the result
80     }
81 }
82
83 this.done = true;
84 //^ to tell the corresponding worker thread that the job has finished when it checks it
85 return this.result;
86 //^ stores result for the corresponding worker thread to fetch
87 }
88
89 @Override
90 public int[][] deserializeIntArrArr(byte[] data){
91     //^ has to be public because it is part of the 'Task3Job' interface
92     //^ to deserialize the raw data int[][] to operate on it
93     ByteArrayInputStream byteInStream = new ByteArrayInputStream(data);
94     //^ holds and formats the byte data of the argument
95     ObjectInputStream inStream;
96     //^ 'ObjectOutputStream' serialize and write objects.
97     int[][] deserialized = null;
98     //^ initialized as null because of the try-statement but null should never be returned
99
100     try {
101         //^ try-statement to deal with potential IOException or ClassNotFoundException errors (mandated by compiler)
102         inStream = new ObjectInputStream(byteInStream);
103         //^ 'ObjectInputStream' initialized for deserialization and writing to 'byteInStream'
104         deserialized = (int[][]) inStream.readObject();
105         //^ deserialize the byte array argument and writes it to 'byteOutStream'.
106         //^ the only code in this method that can potentially cause a 'ClassNotFoundException' error.
107         inStream.close();
108         //^ closes initialization to free up resources
109     }
110     catch (IOException|ClassNotFoundException e) { e.printStackTrace(); }
111     //^ for possible errors relating to deserialization, int[][] parsing, and the 'ObjectInputStream' instance
112
113     return deserialized;
114     //^ Returns a 2d int array containing the deserialized data
115 }

```

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

```

1  //: all relevant imports
2  import java.io.ByteArrayInputStream;
3  import java.io.ByteArrayOutputStream;
4  import java.io.IOException;
5  import java.io.ObjectInputStream;
6  import java.io.ObjectOutputStream;
7  import java.util.Arrays;
8
9  public class Task3 {
10     private static Utilities utilities = new Utilities();
11     //^ for printing matrixes and generating matrixes
12     private static Task3ThreadPool threadPool;
13     //^ stores the thread pool instance in the main class as they must be in the same thread to reflect
14     //^ the master node in the Thread-per-Node Model,
15     private static int[][] matrix;
16     //^ stores randomly generated matrix then used for the result of the Thread-per-Node solution
17     private static int[][] goldenStandardMatrix;
18     //^ stores the matrix result of the golden solution
19     private static int[][] unformatted;
20     //^ stores the jagged array from each kernel application
21
22     public static void main(String[] args) {
23         threadPool = new Task3ThreadPool( utilities.getProcessingEleCount() );
24         //^ instantiate the thread pool with an appropriate number of sub-threads determined by user
25         GoldenStandard goldenStandard = new GoldenStandard();
26
27         Task3.matrix = Task3.utilities.matrixGen();
28         //^ generate the matrix of random numbers (0-9)
29         Task3.utilities.printMatrix(matrix, "Randomly generated matrix:");
30         //^ prints the generated matrix (in a user-friendly manner)
31
32         long startTime;
33         //^ to store start time to time execution speeds
34
35         System.out.println("Starting golden standard solution:");
36         startTime = System.nanoTime();
37         //^ get current time
38         Task3.goldenStandardMatrix = goldenStandard.apply(matrix);
39         //^ golden solution calculates result and gets caught into 'goldenStandardMatrix' field
40         System.out.println("Finished golden standard solution.");
41         System.out.println("Total time golden standard solution:");
42         Task3.utilities.executionTime(startTime);
43         //^ calculates and displays execution duration time based on started time
44         Task3.utilities.printMatrix(goldenStandardMatrix, "Resulting matrix from the golden standard solution:");
45
46         System.out.println("Starting compute cluster solution:");
47         startTime = System.nanoTime();
48         //: record initial matrix dimensions for the normalization parts
49         int martix1stDimentsion = matrix.length;
50         int martix2ndDimentsion = matrix[0].length;
51
52         //: apply sharpen kernel and normalize result
53         Task3.unformatted = Task3.applyKernel(Task3.matrix, 1);
54         Task3.matrix = Task3.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);
55
56         //: apply edge-detection kernel and normalize result
57         Task3.unformatted = Task3.applyKernel(Task3.matrix, 2);
58         Task3.matrix = Task3.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);
59
60         Task3.poisonPill();
61         //^ close all sub-threads
62         //: output result and duration time
63         System.out.println("Finished compute cluster solution.");
64         System.out.println("Total time of compute cluster solution:");

```

```

65     Task3.utilities.executionTime(startTime);
66     Task3.utilities.printMatrix(Task3.matrix, "Resulting matrix from the compute cluster solution:");
67
68     System.out.println("Comparing both solutions:");
69     System.out.println(Task3.utilities.compareMatrixies(Task3.matrix, goldenStandardMatrix));
70     //^ comparison by simple subtractions
71     System.out.println("Comparison done, program finished.");
72 }
73 private static int[] sliceIndexByWorkload(int[][] matrix, int start, int end, boolean applyKernel) {
74     /** Necessary to minimize workload, instead of giving out whole copies (to nodes), as much as possible
75     /** to not exceed the default maximum memory heap leap limit - especially for scaling matrix or
76     /** sub-thread count (program scalability).
77     /** Can work for both normalized and jagged arrays - for code integrity.
78     /**: all set-ups initialized to satisfy compiler but initialization should not be part of the returned.
79     int startSlice = 0;
80     //^ starting sub-array's index
81     int endSlice = 0;
82     //^ ending sub-array's index
83     int startSubArrayIndex = 0;
84     //^ starting index inside the range sub-arrays
85     int endSubArrayIndex = 0;
86     //^ ending index inside the range sub-arrays
87
88     /**: determine which sub-arrays are needed and what range inside of them of them
89     endSubArrayIndex = end - start;
90     //^ assume the the startSubArrayIndex is 0
91     for (int index = 0; index < matrix.length; index++) {
92         /** calculates both sub-array start and beginning start range index
93         if (start < matrix[index].length) {
94             startSubArrayIndex = start;
95             startSlice = index;
96             break;
97         }
98         start -= matrix[index].length;
99     }
100     endSubArrayIndex += startSubArrayIndex;
101     //^ do not assume the the startSubArrayIndex is 0
102     for (int index = 0; index < matrix.length; index++) {
103         /** calculates just the sub-array end
104         if (end < matrix[index].length) {
105             endSlice = index+1;
106             //^ adds 1 because array slicing does not include the ending index in the slice
107             break;
108         }
109         end -= matrix[index].length;
110     }
111
112     if (applyKernel) {
113         //^ only if applying kernels, not normalization.
114         /**: Make sure matrix part is surrounded with more the matrix for the kernel applications
115         /**: to give accurate result.
116         /**: extra comparisons for code integrity - in case developer scales matrix size down.
117         if (startSlice != 0) {
118             startSlice -= 1;
119             //^ adds the values of above matrix sub-array.
120             /** Considers the shift in range indexes when pushing elements at the start of the workload.
121             int indexShift = matrix[startSlice].length;
122             startSubArrayIndex += indexShift;
123             endSubArrayIndex += indexShift;
124         }
125         if (endSlice != matrix.length) {
126             endSlice += 1;
127             //^ adds the values of above matrix sub-array
128         }
129     }

```

```

130
131     return new int[] {startSlice, endSlice, startSubArrayIndex, endSubArrayIndex};
132 }
133 private static int[][] applyKernel(int[][] matrix, int kernelType){
134     int threadCount = Task3.threadPool.threadCount();
135     //^ ask thread pool how many sub-threads is it currently handling
136     int totalSize = matrix.length * matrix[0].length;
137     //^ get element count to calculate distributed workload size
138     int sectorSize = totalSize / threadCount;
139     //^ calculate distributed workload size
140     int[] args;
141
142     //: For code integrity purposes - if another developer changes initial matrix size to a very small size (STILL MUST BE ATLEAST 1x1).
143     if (sectorSize == 0) {
144         //^ happens when there are more sub-threads than initial matrix element count
145         sectorSize = 1;
146         //^ distribute smallest possible workload
147         threadCount = totalSize;
148         //^ to not bother giving redundant jobs to the extra sub-threads
149     }
150
151     //: determines parameter arguments for queued jobs - considers odd elements out from workload divisions and certain sub-thread counts
152     for (int index = 0; index < threadCount-1; index++) {
153         //^ fills the rest of the thread pool and gives the data to the rest of the threads to do the jobs
154         args = sliceIndexByWorkLoad(matrix, index*sectorSize, ((index+1)*sectorSize)-1, true);
155         Task3.threadPool.enqueueJob( new Task3JobKernelApplication(kernelType,Task3.serializeIntArray(Arrays.copyOfRange(matrix,args[0],args[1])),args[2],args[3]) );
156     }
157     if (threadCount > 1) {
158         //^ to account for the remainder matrix elements after division by number of sub-threads.
159         args = sliceIndexByWorkLoad(matrix, (threadCount-1)*sectorSize,totalSize-1, true);
160         Task3.threadPool.enqueueJob( new Task3JobKernelApplication(kernelType,Task3.serializeIntArray(Arrays.copyOfRange(matrix,args[0],args[1])),args[2],args[3]) );
161     }
162     if (threadCount == 1) {
163         //^ if user/developer wants to use only one sub-thread, for testing or comparison reasons.
164         Task3.threadPool.enqueueJob( new Task3JobKernelApplication(kernelType,Task3.serializeIntArray(matrix),0,totalSize-1) );
165     }
166
167     Task3.threadPool.wakeUp();
168     //^ activate thread pool to let sub-threads fetch jobs
169
170     int[][] result = Arrays.copyOfRange(Task3.deserializeIntArray(Task3.threadPool.getResults()),0,threadCount);
171     //^ the array slicing is for the edge-detection kernel application as results from last thread pool wake-up will be brought along
172     //^ in accordance to the 'code integrity purposes' mentioned in this method.
173
174     //: for-loop is for the sharpen kernel application as results will include nulls in accordance to the 'code integrity purposes' mentioned in this method.
175     for (int index = 0; index < result.length; index++) {
176         //^ For code integrity purposes (same as the previous integrity purposes)
177         if (result[index]==null) { return Arrays.copyOfRange(result, 0, index); }
178     }
179
180     threadPool.clear();
181     //^ remove all now-redundant stored data from thread pool to minimise program's heap memory usage
182     return result;
183     //^ returns the result as an jagged int array
184 }

```

```

185 private static int[][] normalization(int[][] unformatted, int subArraysCount, int subArraysLength){
186     int threadCount = Task3.threadPool.threadCount();
187     //^ ask thread pool how many sub-threads is it currently handling
188     int[][] matrix = new int[subArraysCount][subArraysLength];
189     //^ creates new jagged array for the result with dimensions based on the original matrix
190     int[][] preMatrix = new int[threadCount][];
191     //^ where the jagged array
192     int totalSize = matrix.length * matrix[0].length;
193     //^ get element count to calculate distributed workload size
194     int sectorSize = subArraysLength;
195     //^ better renaming for methods; body for easier code development and maintenance
196     int sectorCount = subArraysCount / threadCount;
197     //^ calculate how many sub-arrays (of the jagged array) each thread can handle
198     int sectorCountRemainder = subArraysCount % threadCount;
199     //^ remainder from "sectorCount" calculation
200     int distributedLoadLength;
201     //^ stores the length (element count) distributed load
202     int matrixIndex;
203     //^ index of the to-be-returned result matrix
204     int activeThreads = threadCount;
205     //^ better renaming for method's body for easier code development and maintenance
206     int[] args;
207
208     distributedLoadLength = sectorCount * sectorSize;
209     //^ Assume there is no remainder in the "sectorCount" calculation
210
211     //: increase workload if the "distributedLoadLength" assumption was wrong
212     if (sectorCountRemainder != 0) {
213         sectorCount++;
214         distributedLoadLength = sectorCount * sectorSize;
215         activeThreads = (totalSize / distributedLoadLength)+1;
216     }
217
218     args = sliceIndexByWorkLoad(unformatted, 0, distributedLoadLength, false);
219     if (threadCount == 1){ args[1] = 1; }
220     //^ if only one sub-thread was used
221     Task3.threadPool.enqueueJob( new Task3JobNormalization(Task3.serializeIntArrArr(Arrays.copyOfRange(unformatted,0,args[1])),args[2],distributedLoadLength) );
222     //^ serialize queue the first job (the beginning)
223
224     for (int index = 1; index < activeThreads; index++) {
225         //^ serialize queue the other jobs (to the end)
226         args = sliceIndexByWorkLoad(unformatted, index*distributedLoadLength, (index+1)*distributedLoadLength, false);
227         //Task4.threadPool.enqueueJob( new Task4JobNormalization(Task4.serializeIntArrArr(unformatted),(index*distributedLoadLength),distributedLoadLength) );
228         if (args[1] == 0) { args[1] = unformatted.length; }
229         //^ for the last job
230         Task3.threadPool.enqueueJob( new Task3JobNormalization(Task3.serializeIntArrArr(Arrays.copyOfRange(unformatted,args[0],args[1])),args[2],distributedLoadLength) );
231     }
232
233     //: wake up thread pool and get the sub-threads collective results
234     threadPool.wakeUp();
235     preMatrix = Task3.deserializeIntArrArr(threadPool.getResults());
236
237     //: translate the result to the matrix due to the place-holder elements
238     matrixIndex = 0;
239     for (int preMatrixIndex = 0; preMatrixIndex < preMatrix.length; preMatrixIndex++) {
240         boolean doneAllElements = false;
241         for (int jobResultIndex = 0; jobResultIndex < preMatrix[preMatrixIndex].length; jobResultIndex += sectorSize) {
242             matrix[matrixIndex] = Arrays.copyOfRange( preMatrix[preMatrixIndex], jobResultIndex, jobResultIndex+sectorSize );
243             //^ get all elements for one row of the resulting matrix
244             matrixIndex++;
245             //^ next row
246             if ((matrixIndex*subArraysLength) >= totalSize) { doneAllElements = true; break; }
247             //^ ignore the place-holders
248         }
249         if (doneAllElements) { break; }

```



```

250     }
251
252     threadPool.clear();
253     //^ remove all now-redundant stored data from thread pool to minimise program's heap memory usage
254     return matrix;
255     //^ return as the kernel application result normalized
256 }
257 private static void poisonPill() {
258     int threadCount = Task3.threadPool.threadCount();
259     for (int index = 0; index < threadCount; index++) {
260         //^ fills the thread pool with Poison Pills to close all threads
261         Task3.threadPool.enqueueJob( new Task3JobPoisonPill() );
262     }
263     Task3.threadPool.wakeup();
264     //^ make thread pool send poison pill (as jobs) to the sub-threads to execute
265 }
266 private static byte[] serializeIntArrArr(int[][] intArrArr){
267     //^ To send serialized copies so referencing between nodes are limited to twice per communication/message
268     ByteArrayOutputStream byteOutputStream = new ByteArrayOutputStream();
269     //^ holds and formats the byte data of the argument
270     ObjectOutputStream outputStream = null;
271     //^ 'ObjectOutputStream' serialize and write objects.
272     //^ Initialized as null because of the try-statement but null should never be used
273
274     try {
275         //^ try-statement to deal with possible IOException errors (mandated by compiler)
276         outputStream = new ObjectOutputStream(byteOutputStream);
277         //^ 'ObjectOutputStream' initialized for serialization and writing to 'byteOutputStream'
278         outputStream.writeObject(intArrArr);
279         //^ serialize the argument and writes it to 'byteOutputStream'
280         outputStream.close();
281         //^ closes initialization to free up resources
282     }
283     catch (IOException e) { e.printStackTrace(); }
284     //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
285     return byteOutputStream.toByteArray();
286     //^ Returns a byte array containing the serialized data
287 }
288 private static int[][] deserializeIntArrArr(byte[] data){
289     //^ no 200ms network delay in this method's calls because 'main' and
290     //^ the thread pool are both in the same main thread
291     ByteArrayInputStream byteInputStream = new ByteArrayInputStream(data);
292     //^ holds and formats the byte data of the argument
293     ObjectInputStream inputStream;
294     //^ 'ObjectOutputStream' serialize and write objects.
295     int[][] deserialized = null;
296     //^ initialized as null because of the try-statement but null should never be returned
297
298     try {
299         //^ try-statement to deal with potential IOException or ClassNotFoundException errors (mandated by compiler)
300         inputStream = new ObjectInputStream(byteInputStream);
301         // 'ObjectInputStream' initialized for deserialization and writing to 'byteInputStream'
302         deserialized = (int[][]) inputStream.readObject();
303         //^ deserialize the byte array argument and writes it to 'byteOutputStream'.
304         //^ The only code in this method that can potentially cause a 'ClassNotFoundException' error.
305         inputStream.close();
306         //^ closes initialization to free up resources
307     }
308     catch (IOException|ClassNotFoundException e) { e.printStackTrace(); }
309     //^ for possible errors relating to deserialization, int[][] parsing, and the 'ObjectInputStream' instance
310     return deserialized;
311     //^ Returns a 2d int array containing the deserialized data
312 }
313 }

```

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:


```

2 public class Task3JobPoisonPill implements Task3Job{
3     //^ implements 'Task3Job' so it can be stored in the thread pool's
4     //^ FIFO job queue.
5     private boolean done = false;
6     //^ simple set-up
7
8     @Override
9     public boolean isDone() { return done; }
10    //^ returns true when '.perform()' gets executed, otherwise return false
11
12    @Override
13    public int[] perform() { this.done = true; return new int[] { }; }
14    //^ when sub-thread catches result of job, it breaks while-loop and
15    //^ closes thread if result is an empty int array.
16    //^ It returns int[] as the other jobs returns int[] as well, hence
17    //^ why so does the method in the 'Task3Job' interface.
18
19    @Override
20    public int[][] deserializeIntArrArr(byte[] data) { return null; }
21    //^ this method will never be called but is there because deserialization
22    //^ is one of the main parts of a job (hence is in the job interface),
23    //^ deserialized data to produce results, the Poison Pill enters sub-thread
24    //^ but unlike jobs to process as a job to stop the sub-thread's while-loop
25    //^ (inside 'start' method) to close the sub-threads otherwise program will
26    //^ have zombie threads.
27 }

```

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: VW
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: VW
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:

```

22     private static int failureRate;
23     //^ stores failure rate in the main "Task4" class to be passed to the worker node constructor
24     //^ as a parameter/argument.
25
26     public static void main(String[] args) {
27         failureRate = Task4.inputFailureRate(utilities.scanner);
28         //^ get failure rate from user.
29         //^ Cannot open 2 scanners (even with "scanner.close()") so had to use the utilities scanner.
30         threadPool = new Task4ThreadPool( utilities.getProcessingEleCount(), failureRate );
31         //^ instantiate the thread pool with an appropriate number of sub-threads determined by user

```

Method of getting failure rate is as seen:

```

244 private static int inputFailureRate() {
245     /** private method of the main Task4 class.
246     /** Code taken and modified from "Utilities.getProcessingEleCount()".
247     /**: set-up
248     Scanner scanner = new Scanner(System.in);
249     boolean validCount = false;
250     /** stores the validity of users input
251     int userInput = 0;
252     /** '0' is a placeholder, but assume user first meant 0% failure rate
253
254     while(!validCount) {
255         /** gets a valid input (0-100 as percentage) from user
256         System.out.println("Enter failure rate for sub-threads before they return results (in percentage between 0-100)");
257         if (!scanner.hasNextInt()) {
258             System.out.println("Error encountered on human counterpart: invalid input - input must be an integer");
259             continue;
260         }
261         userInput = scanner.nextInt();
262         if( userInput < 1) {
263             /** below percentage range
264             System.out.println("Error encountered on human counterpart: invalid input - (integer) input must be positive");
265             continue;
266         }
267         if( userInput > 100) {
268             /** above percentage range
269             System.out.println("Error encountered on human counterpart: invalid input - cannot be above 100%");
270             continue;
271         }
272         /**: success
273         System.out.println("Human counterpart compliant");
274         System.out.println( String.format("Failure rate is %d%", userInput) );
275         validCount = true;
276     }
277
278     /**: success
279     scanner.close();
280     return userInput;
281 }
282 }

```

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

```

29 public Task4ThreadPool(int threadCount, int failureRate) {
30     /**: set-up the fields
31     this.sleep = false;
32     /** activate the thread pool
33     jobQueue = new ArrayList<>();
34     /** for good practice and scalability - the data structure has dynamic size
35     threadJobResults = new int[threadCount][];
36     /** is a jagged array, because jobs' returned int arrays' length are not predefined
37
38     /**: set-up and fill the thread pool
39     this.threads = new Task4WorkerNode[threadCount];
40     for (int index = 0; index < threadCount; index++) {
41         this.threads[index] = new Task4WorkerNode(failureRate, index);
42         /** worker thread instance, parameters are both primitives so no references
43         /** are passed cross-node/thread.
44         this.threads[index].start();
45         /** start the thread's execution
46     }
47 }

```

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

```

20 private int failureRate;
21 //^ the probability of thread dying before fetching job's result
22 private int threadPoolIndex;
23 //^ let the user know which node/thread died when it does.
24
25 public Task4WorkerNode(int failureRate, int threadPoolIndex) {
26     //^ constructor for the failure percentage change.
27     //^ Parameters are primitives so no references are passed between nodes.
28     this.failureRate = failureRate;
29     this.threadPoolIndex = threadPoolIndex;
30 }
31
32 private boolean nodeDeath() {
33     //^ calculation to decide if the node/thread was to live or die
34     Random rand = new Random();
35     boolean death = rand.nextInt(100) < this.failureRate;
36     //^ based on percentage chance, does node/thread dies?
37     if (death) {
38         //^ node/thread dies
39         long threadId = Thread.currentThread().getId();
40         //^ does not interfere with node's job to actually simulate the node's death - not realising that it is dead.
41         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));
42         //^ thread pool index is more relevant than thread id because index correlate to what workload the specific sub-thread is handling
43         //^ but still including both in case if user wants both numbers.
44         //^ This tells user that sub-thread died but thread pool does not know that yet.
45     }
46     return death;
47     //^ will die?
48 }
49
50 public void run() {
51     this.running = true;
52     //^ dictates if the thread should continue running or end
53     while (this.running) {
54         //^ while-loop runs constantly till thread must end
55
56         if (Thread.interrupted()) { break; }
57         //^ ending ".run()", when interrupted, to end thread by death.
58         //^ Does not toggle "this.running" because node/thread cannot do
59         //^ anything when it dies.
60
61         if (job != null && !this.jobDone) {
62             //^ execute when there is an existing job that is not completed
63
64             int[] jobResultCache = this.job.perform();
65             //^ where the job actually gets done but caught result is stored
66             //^ in-scope until the thread survives the death probability.
67             if (!this.nodeDeath()) { jobResult = jobResultCache; }
68             //^ if thread does not die (by % probability), then store result
69             //^ where it can be fetched by the thread pool and continue as normal.
70             else { this.interrupt(); continue; }
71             //^ if thread dies then end ".run()" execution by interruption when it gets checked
72         }

```

Findings and fixes

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

```

57     //: apply sharpen kernel and normalize result
58     Task4.unformatted = Task4.applyKernel(Task4.matrix, 1);
59     System.out.println("compute cluster solution - Sharpen kernel application applied");
60     Task4.matrix = Task4.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);
61     System.out.println("compute cluster solution - applied matrix normalized (1 of 2)");
62
63     //: apply edge-detection kernel and normalize result
64     Task4.unformatted = Task4.applyKernel(Task4.matrix, 2);
65     System.out.println("compute cluster solution - Edge-detection kernel application applied");
66     Task4.matrix = Task4.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);
67     System.out.println("compute cluster solution - applied matrix normalized (2 of 2)");
68
69     Task4.poisonPill();
70     //^ close all sub-threads
71     System.out.println("compute cluster solution - Poison Pills administered");
72     //: output result and duration time
73     System.out.println("Total time of compute cluster solution:");
74     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: WV
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:

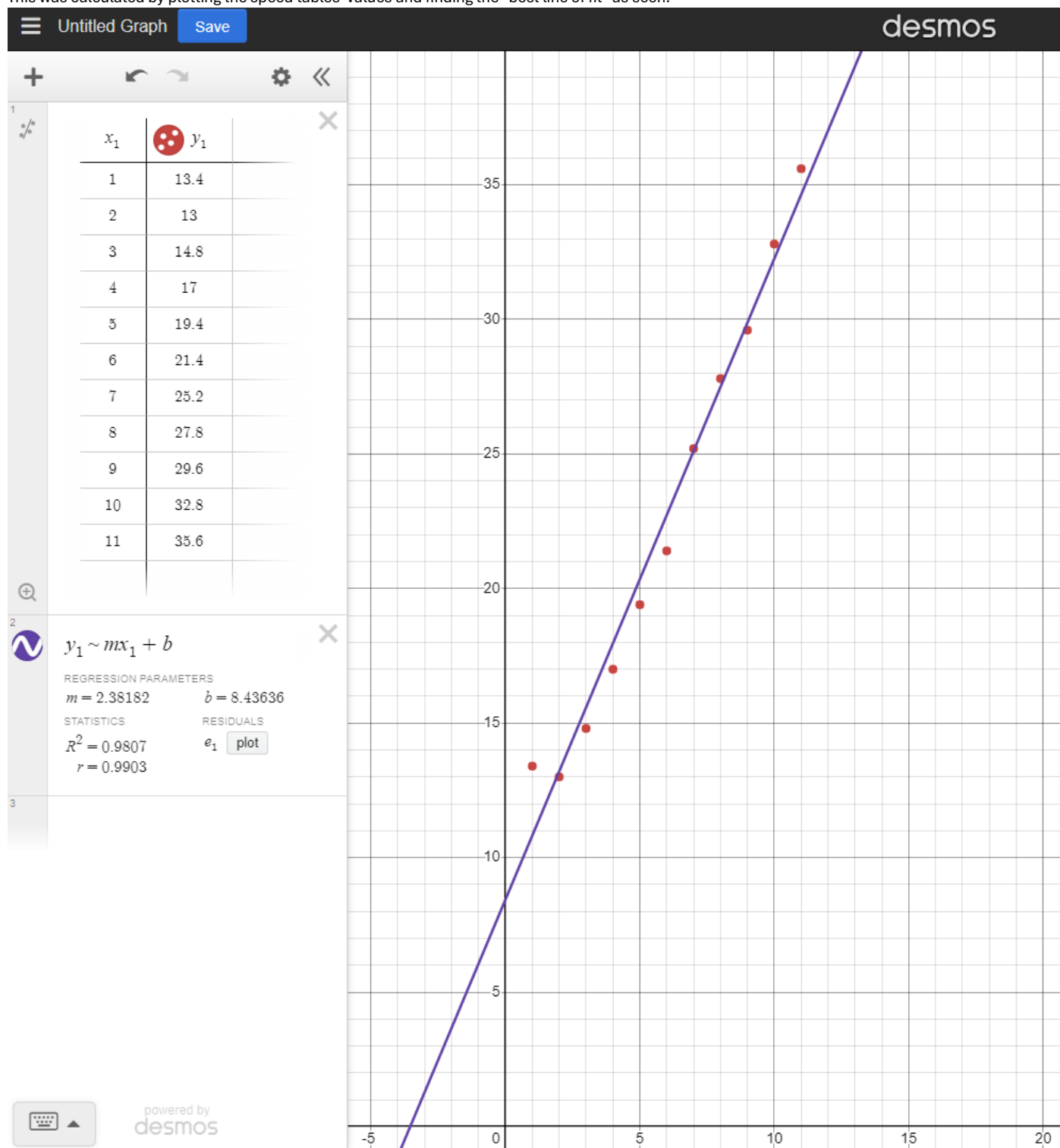
(seconds to the lower whole integer)	9 sub-threads	10 sub-threads	11 sub-threads
trail #1	30	33	34
trail #2	29	32	36
trail #3	30	33	36
trail #4	29	33	36
trail #5	30	33	36
mean	29.6	32.8	35.6

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:


```

51     //: apply sharpen kernel and normalize result
52     startTime = System.nanoTime();
53     Task3.unformatted = Task3.applyKernel(Task3.matrix, 1);
54     Task3.utilities.executionTime(startTime);
55     startTime = System.nanoTime();
56     Task3.matrix = Task3.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);
57     Task3.utilities.executionTime(startTime);
58
59     //: apply edge-detection kernel and normalize result
60     startTime = System.nanoTime();
61     Task3.unformatted = Task3.applyKernel(Task3.matrix, 2);
62     Task3.utilities.executionTime(startTime);
63     startTime = System.nanoTime();
64     Task3.matrix = Task3.normalization(unformatted, martix1stDimentsion, martix2ndDimentsion);
65     Task3.utilities.executionTime(startTime);
66     startTime = System.nanoTime();
67
68     Task3.poisonPill();
69     Task3.utilities.executionTime(startTime);
70     //^ close all sub-threads
71     //: output result and duration time
72     System.out.println("Total time of compute cluster solution:");
73     Task3.utilities.executionTime(startTime2);
74     //Task3.utilities.printMatrix(Task3.matrix, "Resulting matrix from the compute cluster solution:");
75
76     System.out.println("Comparing both solutions:");
77     System.out.println(Task3.utilities.compareMatrixies(Task3.matrix, goldenStandardMatrix));
78     //^ comparison by simple subtractions
79     System.out.println("Comparison done, program finished.");

```

Console X

<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: WV

4226079300, 4226, 4

Starting compute cluster solution:

Execution time in nanoseconds, milliseconds, and seconds respectfully: WV

9137876900, 9137, 9

Execution time in nanoseconds, milliseconds, and seconds respectfully: WV

8647034000, 8647, 8

Execution time in nanoseconds, milliseconds, and seconds respectfully: WV

8256288000, 8256, 8

Execution time in nanoseconds, milliseconds, and seconds respectfully: WV

8242678600, 8242, 8

Execution time in nanoseconds, milliseconds, and seconds respectfully: WV

624861000, 624, 0

Total time of compute cluster solution:

Execution time in nanoseconds, milliseconds, and seconds respectfully: WV

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:


```

95 private void requestResults() {
96     this.nodeInactivityCount = new int[this.threads.length];
97     boolean allResults = false;
98     //^ dictates if it thread pool should keep waiting for all results or not
99     boolean[] alldone = new boolean[this.maxJobs];
100     //^ only fetch results from the relevant threads to save processing time
101     while (!allResults) {
102         //^ far faster than the alternative of constantly checking a sub-thread job until it
103         //^ is done one at a time.
104         for (int index = 0; index < this.maxJobs; index++) {
105             //^ get all results to if each thread has finished the job or not
106             alldone[index] = threads[index].isJobDone();
107             //^ store the each job status
108         }
109         try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
110         //^ to impose a delay of 200ms for each computational job executed to simulate the
111         //^ speeds of modern computers and cluster networks.
112         //^ These node-to-node messages are to check if the all sub-thread has finished
113         //^ their given jobs.
114         allResults = true;
115         //^ Assume that all threads have finished
116         for (int index = 0; index < this.maxJobs; index++) {
117             //^ only gathers threads that were given relevant jobs
118             if(alldone[index] == false) {
119                 //^ current thread hasn't finished?
120                 allResults = false;
121                 //^ if one thread isn't finished then all threads are not finished!
122             }
123             //: check for dead threads by multiple periodic calls.
124             //: Assume that the main thread never dies.
125             nodeInactivityCount[index]++;
126             //^ sub-thread has not given result yet
127             if (nodeInactivityCount[index]*200 > this.predictedCompletion) {
128                 //^ comparison is milliseconds.
129                 //^ Allow sub-thread to take up to double predicted duration to takes
130                 //^ int division and older/slower computers that run this program into account.
131                 //^ If sub-thread takes too long, assume it died, then recreate it with the missing work.
132                 this.threads[index] = null;
133                 this.threads[index] = new Task4WorkerNode(failureRate, index);
134                 //^ make new thread in the dead thread's place in the thread pool
135                 this.threads[index].setJob(this.serializeJob(this.jobHistory[index]));
136                 //^ assign missing work based on job assignment history
137                 try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
138                 //^ to impose a delay of 200ms for another computational job executed to simulate the
139                 //^ speeds of modern computers and cluster networks.
140                 //^ Sent to the node in place of the one that was detected as dead.
141                 System.out.println(String.format("Thread pool detected dead node - node recreated and starting with former node's job (thread pool index: %d)", index));
142                 this.threads[index].start();
143                 //^ Tell the new sub-thread to start executing jobs.
144                 try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
145                 //^ to impose a delay of 200ms when telling the node to execute the job.
146                 //^ To simulate the speeds of modern computers and cluster networks.
147                 nodeInactivityCount[index] = 0;
148                 //^ reset inactivity counter as thread is brand new
149             }
150         }
151     }
152 }
153 }
154 for (int index = 0; index < this.maxJobs; index++) {
155     //^ gather results of all relevant threads
156     threadJobResults[index] = this.deserializeIntArr(this.threads[index].requestResult());
157     //^ deserialize raw data MPI message into the fetched int[] result to be stored
158 }
159 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: WV
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 “byteOutputStream.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:

```

67 public void clear() /*reset method*/ {
68     /* not exactly aligned with the Thread-per-Node Model but the program must not exceed the Java
69     /* default maximum heap memory limit, which cannot be changed in-program.
70     /* The lower it is, the better the program's scalability.
71     /*: these data are, at this point of time, redundant so they must be deleted
72     this.threadJobResults = new int[this.threads.length][];
73     this.jobHistory = null;
74     for (int index = 0; index < this.threads.length; index++) { threads[index].clear(); }
75     /*^ clears each worker node's stored result
76     try { Thread.sleep(200); } catch (InterruptedException e) { e.printStackTrace(); }
77     /*^ to impose a delay of 200ms to simulate the speeds of modern computers and cluster networks.
78     /*^ Tells all threads to delete their now redundant stored result.
79 }

```

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: WV
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: WV
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: WV
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: WV
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:

```

52     System.out.println("Starting compute cluster solution:");
53     startTime = System.nanoTime();
54     while (true) {
55         /* To simulate possible MPI message corruptions disrupting the final error.
56         //: Record initial matrix dimensions for the normalization parts
57         int martix1stDimention = matrix.length;
58         int martix2ndDimention = matrix[0].length;
59
60         //: apply sharpen kernel and normalize result
61         Task4.unformatted = Task4.applyKernel(Task4.matrix, 1);
62         System.out.println("=== compute cluster solution - Sharpen kernel application applied ===");
63         Task4.matrix = Task4.normalization(Task4.unformatted, martix1stDimention, martix2ndDimention);
64         System.out.println("=== compute cluster solution - applied matrix normalized (1 of 2) ===");
65
66         //: apply edge-detection kernel and normalize result
67         Task4.unformatted = Task4.applyKernel(Task4.matrix, 2);
68         System.out.println("=== compute cluster solution - Edge-detection kernel application applied ===");
69         Task4.matrix = Task4.normalization(Task4.unformatted, martix1stDimention, martix2ndDimention);
70         System.out.println("=== compute cluster solution - applied matrix normalized (2 of 2) ===");
71
72         Task4.poisonPill();
73         //^ close all sub-threads
74         System.out.println("compute cluster solution - Poison Pills administered");
75         //: output result and duration time
76         System.out.println("Total time of compute cluster solution:");
77         Task4.utilities.executionTime(startTime);
78         Task4.utilities.printMatrix(Task4.matrix, "Resulting matrix from the compute cluster solution:");
79
80         System.out.println("Comparing both solutions:");
81         String comparison = Task4.utilities.compareMatrixies(Task4.matrix, goldenStandardMatrix);
82         //^ comparison by simple subtractions
83         System.out.println(comparison);
84         if (comparison == "both matrixes are identical in both: dimensions and elements' value") { break; }
85
86         System.out.println("Redoing solution!");
87     }
88     System.out.println("Comparison successful, program finished.");
89 }

```

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: VW
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: VW
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: VW
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: VW
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.

```

57     public int[][] matrixGen() { return fill(new int[10000][10000]); }
58     //^ setting up matrix's dimensions (10,000x10,000) and generating it.
59     //^ 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: VW
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: VW
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.

```



```
57     public int[][] matrixGen() { return fill(new int[12500][12500]); }
58     //^ setting up matrix's dimensions (10,000x10,000) and generating it.
59     //^ 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: VW
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: VW
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:

```
6 public class Task4JobKernelApplication implements Task4Job {
7     //: set-up.
8     //: Kernels declared in this way to allow other developers to easily manipulate its elements' value
9     //: and program will still work as intended.
10    private int[][] kernelSharpen = {
11        /* element's values can be changed if wished by a developer
12        {0, -1, 0},
13        {-1, 5, -1},
14        {0, -1, 0}
15    };
16    private int[][] kernelEdgeDetection = {
17        /* element's values can be changed if wished by a developer
18        {-1, -1, -1},
19        {-1, 8, -1},
20        {-1, -1, -1}
21    };
```

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: VVW");
    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-dimensional 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.compareMatrixes(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 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.serializeIntArray(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.serializeIntArray(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.serializeIntArray(matrix),0,totalSize-1) );
    }
}

```

```

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

        int[][] result =
Arrays.copyOfRange(Task3.deserializeIntArray(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.serializeIntArray(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-holder elements
    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-threads to execute
}

private static byte[] serializeIntArrArr(int[][] intArrArr){
    /* To send serialized copies so referencing between nodes are limited to twice per
communication/message
    ByteArrayOutputStream byteOutputStream = new ByteArrayOutputStream();
    //^ holds and formats the byte data of the argument
    ObjectOutputStream outputStream = 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)
        outputStream = new ObjectOutputStream(byteOutputStream);
        //^ 'ObjectOutputStream' initialized for serialization and writing to
        'byteOutputStream'
        outputStream.writeObject(intArrArr);
        //^ serialize the argument and writes it to 'byteOutputStream'
        outputStream.close();
        //^ closes initialization to free up resources
    }
    catch (IOException e) { e.printStackTrace(); }
    //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
    return byteOutputStream.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 byteInputStream = 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(byteInputStream);
        // 'ObjectInputStream' initialized for deserialization and writing to
        'byteInputStream'
        deserialized = (int[][] inStream.readObject());
        //^ deserialize the byte array argument and writes it to 'byteOutputStream'.
        //^ 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-existent 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.
the sub-thread.
    //^ this node-to-node message is the serialized job from the main thread to
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 byteOutputStream = new ByteArrayOutputStream();
    //^ holds and formats the byte data of the argument
    ObjectOutputStream outputStream = null;
    //^ 'ObjectOutputStream' serialize and write objects.
    //^ initialized as null because of the try-statement but null should never be used

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

    return byteOutputStream.toByteArray();
    //^ Returns a byte array containing the serialized data
}
private int[] deserializeIntArr(byte[] data){
    ByteArrayInputStream byteInputStream = new ByteArrayInputStream(data);
    //^ holds and formats the byte data of the argument.
    ObjectInputStream inStream;
    //^ 'ObjectInputStream' 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(byteInputStream);

```

```

        //^ '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 byteOutputStream = new ByteArrayOutputStream();
    //^ holds and formats the byte data of the argument
    ObjectOutputStream outputStream = 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)
        outputStream = new ObjectOutputStream(byteOutputStream);
        //^ 'ObjectOutputStream' initialized for serialization and writing to
        'byteOutStream'
        outputStream.writeObject(array);
        //^ serialize the 2d int argument and writes it to 'byteOutStream'
        outputStream.close();
        //^ closes initialization to free up resources
    }
    catch (IOException e) { e.printStackTrace(); }
    //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance

    return byteOutputStream.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 byteOutputStream = new ByteArrayOutputStream();
    //^ holds and formats the byte data of the argument
    ObjectOutputStream outputStream = 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)
        outputStream = new ObjectOutputStream(byteOutputStream);
        //^ 'ObjectOutputStream' initialized for serialization and writing to
'byteOutputStream'
        outputStream.writeObject(array);
        //^ serialize the argument and writes it to 'byteOutputStream'
        outputStream.close();
        //^ closes initialization to free up resources
    }
    catch (IOException e) { e.printStackTrace(); }
    //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
    return byteOutputStream.toByteArray();
    //^ Returns a byte array containing the serialized data
}

private Task3Job deserializeJob(byte[] data){
    /* to deserialized the raw data message into a job
    ByteArrayInputStream byteInputStream = 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 'byteOutputStream'.
        //^ 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.deserializeIntArray(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 'byteOutputStream'.
        //^ 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 martix1stDimention = matrix.length;
    int martix2ndDimention = 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, martix1stDimention,
martix2ndDimention);
    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, martix1stDimention,
martix2ndDimention);
    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.compareMatrixes(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-holder elements
    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-threads to execute
    }

    private static byte[] serializeIntArrArr(int[][] intArrArr){
        /* To send serialized copies so referencing between nodes are limited to twice per
communication/message
        ByteArrayOutputStream byteOutputStream = new ByteArrayOutputStream();
        //^ holds and formats the byte data of the argument
        ObjectOutputStream outputStream = 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)
            outputStream = new ObjectOutputStream(byteOutputStream);
            //^ 'ObjectOutputStream' initialized for serialization and writing to
'byteOutputStream'
            outputStream.writeObject(intArrArr);
            //^ serialize the argument and writes it to 'byteOutputStream'
            outputStream.close();
            //^ closes initialization to free up resources
        }
        catch (IOException e) { e.printStackTrace(); }
        //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance

        return byteOutputStream.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 byteInputStream = new ByteArrayInputStream(data);
        //^ holds and formats the byte data of the argument
        ObjectInputStream inputStream;

```

```

    //^ '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 encountered 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 encountered on human counterpart: invalid
            input - (integer) input must be positive");
            scanner.next();
            continue;
        }
        if( userInput > 100) {
            /* above percentage range
            System.out.println("Error encountered on human counterpart: invalid
            input - cannot 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.serializeIntArray(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-existent 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 byteOutputStream = new ByteArrayOutputStream();
    /*^ holds and formats the byte data of the argument
    ObjectOutputStream outputStream = null;
    /*^ 'ObjectOutputStream' serialize and write objects.
    /*^ initialized as null because of the try-statement but null should never be used

    try {
        /*^ try-statement to deal with possible IOException errors (mandated by compiler)
        outputStream = new ObjectOutputStream(byteOutputStream);

```

```

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

    return byteOutputStream.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 'byteOutputStream'.
        //^ 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 byteOutputStream = new ByteArrayOutputStream();
    //^ holds and formats the byte data of the argument
    ObjectOutputStream outputStream = 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)
        outputStream = new ObjectOutputStream(byteOutputStream);
        //^ 'ObjectOutputStream' initialized for serialization and writing to
        'byteOutputStream'
        outputStream.writeObject(array);
        //^ serialize the 2d int argument and writes it to 'byteOutputStream'
        outputStream.close();
        //^ closes initialization to free up resources
    }
    catch (IOException e) { e.printStackTrace(); }
    //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance

    return byteOutputStream.toByteArray();

```

```

    }
}

```

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 byteOutputStream = new ByteArrayOutputStream();
    //^ holds and formats the byte data of the argument
    ObjectOutputStream outputStream = 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)
        outputStream = new ObjectOutputStream(byteOutputStream);

```

```

        //^ 'ObjectOutputStream' initialized for serialization and writing to
        'byteOutputStream'
        outputStream.writeObject(array);
        //^ serialize the argument and writes it to 'byteOutputStream'
        outputStream.close();
        //^ closes initialization to free up resources
    }
    catch (IOException e) { e.printStackTrace(); }
    //^ for possible errors relating to serialization and the 'ObjectOutputStream' instance
    return byteOutputStream.toByteArray();
    //^ Returns a byte array containing the serialized data
}

private Task4Job deserializeJob(byte[] data){
    /* to deserialize 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 'byteOutputStream'.
        //^ 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 'byteOutputStream'.
        //^ 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.
}
```