DISPARP Lab Book

Introduction

The second part of DISPARP is focused more on utilising parallelism to solve problems faster or execute simulation type programs much quicker than its sequential counterpart. The first half of this lab report will be records about all the lab assignments during the weekly practical session. The remaining half will be a matrix multiplication development project. The aim of the development project is to incorporate parallelism in square matrix multiplication to reduce the amount of time it takes to calculate the new matrix for multiplication of two square matrices with a large size of N by N when N is larger than 100.

Lab assignments

Week 13 - Java Threads for Parallelism – Reprise

The first week of teaching block two introduced parallel programming by benchmarking two different versions, a sequential approach and a parallel approach, on how to approximate π using the rectangle rule. It provided an insight into the use and advantages of parallelism to solve computation problems like approximating π.

The sequential approach demonstrates the approximation is very close to know approximations such as the one shown in Wikipedia. The constant N represents the value of the variable numsteps and with the larger value of N, the approximation will become more accurate and closer to the truth value. A larger value of numsteps requires more computation power as more loops are carried out in the computation which leads to more time needed to calculate the value.

Comparison between the estimated value from SequentialPi function and known estimates such as the one from wiki

Sequential	
Source	Value
SequentialPi	3.14159265358973
Wiki	3.14159265358979
Difference	-0.00000000000006

The second approach uses parallelism to tackle the problem of more computation power is needed when N becomes larger and larger, this is achieved by splitting the sum into two parts and calculate two sums in separate threads with the use of thread class. The variable of begin and end are used to split the numsteps into two equal parts.

```
thread1.begin = 0 ;               thread2.begin = numSteps / 2 ;
thread1.end = numSteps / 2 ;      thread2.end = numSteps ;
```

The total number of steps is the same as for the sequential approach, but half and half of the steps carried out at the same time leads to approximately two times faster in the calculated time and this is shown in the results.

Avearage	Time(milliseconds)		
SequentialPi	89.1	Parallel speedup	n steps = 10m
ParallelPi	48.6	ParallelPi	1.833333333

Different amount of numsteps are used to demonstrate how the parallel speed up varies in different problem size, changing the value of numsteps to 1 million and 1 billion.

Parallel speedup	n steps = 10m	n steps = 1m	n steps = 1b
ParallelPi	1.833333333	0.966386555	2.018693994

From the results above, the bigger the problem size the better the efficiency is for the parallel speedup. The parallel speedup is the run time of sequential time over parallel time.

Exercise

1) The "quadcore" version is similar to the ParallelPi but it divides the numsteps into four equal parts instead of two and with two extra threads to run those two extra parts.

```
QuadPi thread1 = new QuadPi();
thread1.begin = 0 ;
thread1.end = numSteps / 4 ;

QuadPi thread2 = new QuadPi();
thread2.begin = numSteps / 4 ;
thread2.end = numSteps / 2 ;

QuadPi thread3 = new QuadPi();
thread3.begin = numSteps / 2 ;
thread3.end = numSteps * 3/4 ;

QuadPi thread4 = new QuadPi();
thread4.begin = numSteps * 3/4 ;
thread4.end = numSteps ;
```

This version has a better parallel speedup but not by a lot since an increasing number of processors decreases the efficiency.

Parallel speedup	n steps = 10m	n steps = 1m
ParallelPi	1.833333333	0.966386555
QuadPi	2.516949153	0.787671233

From the results, the speedup is not twice as fast as the ParallelPi even the range is divided into four equal parts.

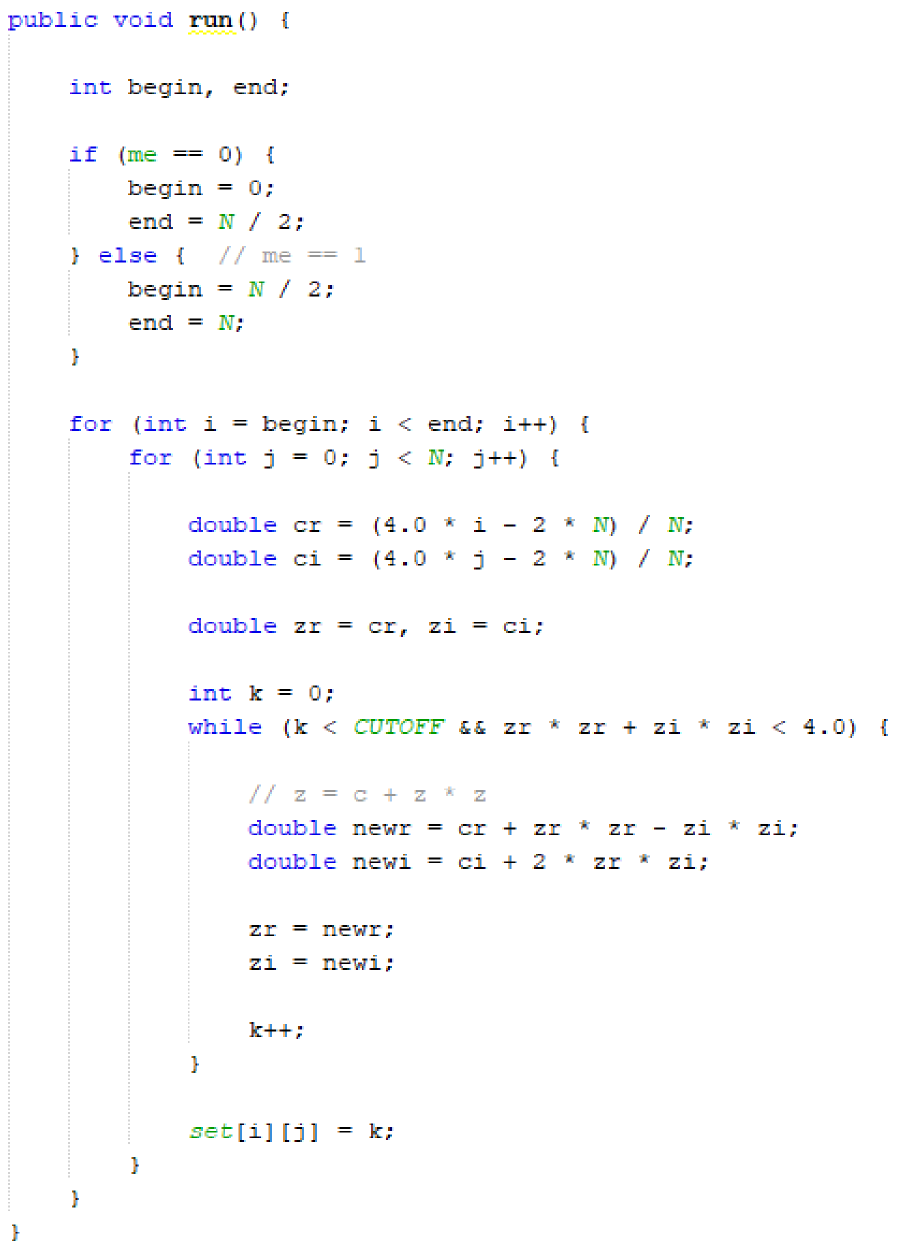
2) Results aren't more accurate but in another unit nanoseconds instead of milliseconds.

	Time(All value converted back into milliseconds)									
ParallelPi	1	2	3	4	5	6	7	8	9	10
nano	49	48	50	48	49	49	48	49	48	49
milli	47	49	48	49	49	49	48	48	50	49
Avearge	Time(milliseconds)									
nano	48.7									
milli	48.6									

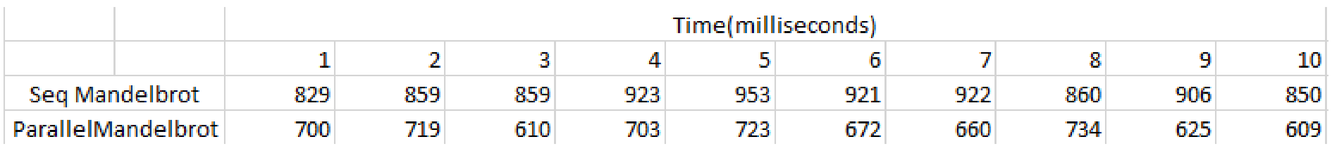
Week 14 - More Parallel Programming - Threads and Data(s)

The second week's lab script carried on from the first week to speed up the execution of problem-solving programs, in this case, it is a program for calculating the Mandelbrot set.

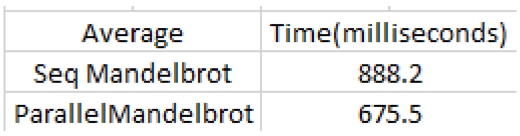
One of the tasks is to parallelise the sequential version to enhance the execution speed. The method is similar to the first week ParallelPi by using begin and end variables to split the range into two equal parts and moving the main calculation loop into the run method, so the threads are running the calculation. The for-loop range is then replaced with begin and end variables.



```
public void run() {

    int begin, end;

    if (me == 0) {
        begin = 0;
        end = N / 2;
    } else {  // me == 1
        begin = N / 2;
        end = N;
    }

    for (int i = begin; i < end; i++) {
        for (int j = 0; j < N; j++) {

            double cr = (4.0 * i - 2 * N) / N;
            double ci = (4.0 * j - 2 * N) / N;

            double zr = cr, zi = ci;

            int k = 0;
            while (k < CUTOFF && zr * zr + zi * zi < 4.0) {

                // z = c + z * z
                double newr = cr + zr * zr - zi * zi;
                double newi = ci + 2 * zr * zi;

                zr = newr;
                zi = newi;

                k++;
            }

            set[i][j] = k;
        }
    }
}
```



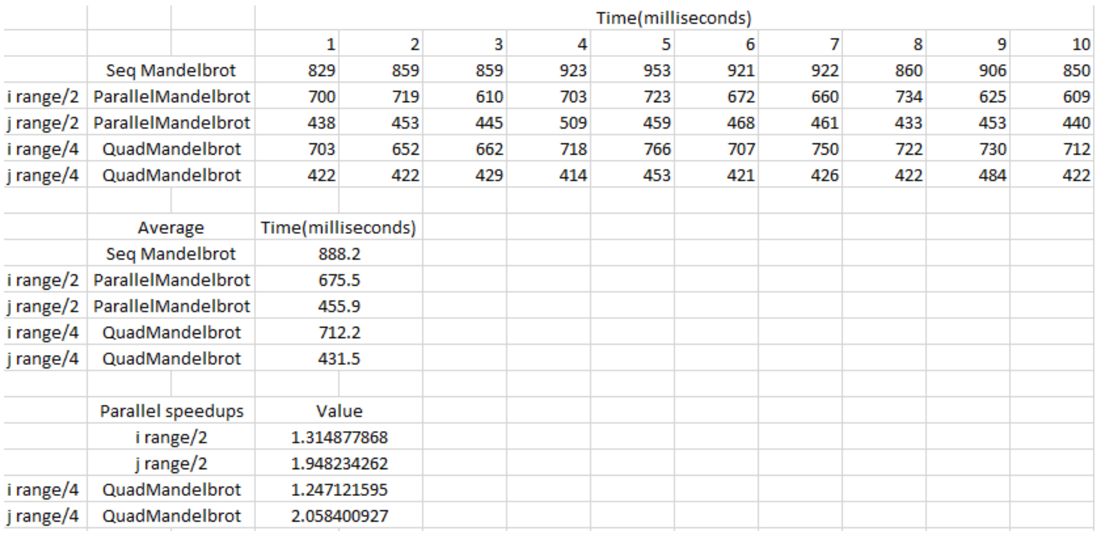
	Time(milliseconds)									
	1	2	3	4	5	6	7	8	9	10
Seq Mandelbrot	829	859	859	923	953	921	922	860	906	850
ParallelMandelbrot	700	719	610	703	723	672	660	734	625	609



Average	Time(milliseconds)
Seq Mandelbrot	888.2
ParallelMandelbrot	675.5

Comparing the results above, there isn't much improvement from the parallel version with about 24% faster than the sequential version. Last week's ParallelPi runs nearly twice as fast as the sequential version.

Exercise



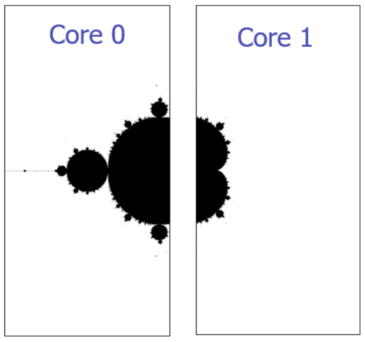
		Time(milliseconds)									
		1	2	3	4	5	6	7	8	9	10
	Seq Mandelbrot	829	859	859	923	953	921	922	860	906	850
i range/2	ParallelMandelbrot	700	719	610	703	723	672	660	734	625	609
j range/2	ParallelMandelbrot	438	453	445	509	459	468	461	433	453	440
i range/4	QuadMandelbrot	703	652	662	718	766	707	750	722	730	712
j range/4	QuadMandelbrot	422	422	429	414	453	421	426	422	484	422

	Average	Time(milliseconds)
	Seq Mandelbrot	888.2
i range/2	ParallelMandelbrot	675.5
j range/2	ParallelMandelbrot	455.9
i range/4	QuadMandelbrot	712.2
j range/4	QuadMandelbrot	431.5

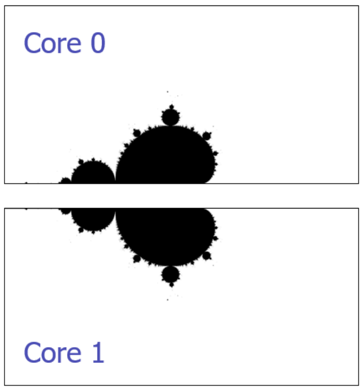
	Parallel speedups	Value
	i range/2	1.314877868
	j range/2	1.948234262
i range/4	QuadMandelbrot	1.247121595
j range/4	QuadMandelbrot	2.058400927

1) The parallel speedup for dividing up the horizontal i range is 1.31.
2) The parallel speedup for dividing up the horizontal j range is 1.94 and it is different from the one dividing the i range because of the perfect load balancing.

   This is how the workload spread across two cores when dividing up the horizontal i range.

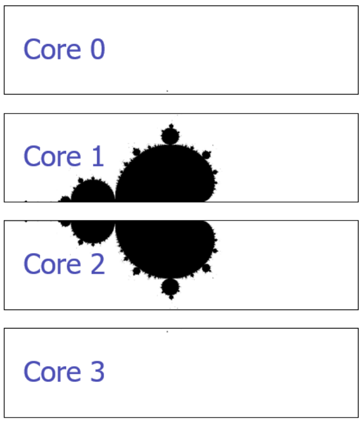


Here is how the workload spread across two cores when dividing up the horizontal j range.



For the two images above, it demonstrates how the workload is divided across two cores when dividing either the i range or j range. The first one has a poor load balancing because the workload isn't divided evenly across two cores as core 0 has more workload. On the other hand, the second one has a better load balancing because the workload is split evenly across two cores and that is why it has a parallel speedup around two.

3) The load balancing isn't better by splitting the range into four equal parts and this is because not all cores have the same amount of workload.



This is how the workload is split across four cores and from the image above, core 0 and core 3 have barely any workload and most of the work is done by core 1 and core 2. This disastrous load balancing is the reason why the whole process isn't finished twice as fast as the two cores version. This is further demonstrating in the results, i range/ 4 got worse parallel speedup than i range/2 as well as j range/4 has a similar parallel speedup as j range/2.

Week 15 - Workload Decompositions, and a "Simulation"

The third week's lab script split into two parts:

The first part was an extension of last week's lab by applying block decomposition and cyclic decomposition to ParallelMandelbrot and benchmarking the results.

Before the two decompositions, the program threads' creation is generalised by using variable P to represents the number of threads and using for-loop to call the thread.start() and thread.join():

```
final static int P = 4;
```

```
ParallelMandelbrot[] threads = new ParallelMandelbrot[P];
for (int me = 0; me < P; me++) {
    threads[me] = new ParallelMandelbrot(me);
    threads[me].start();
}

for (int me = 0; me < P; me++) {
    threads[me].join();
}
```

For block decomposition, variables begin and end are changed to:

```
int b = N / P;   // block size

begin = me * b;
end = begin + b;
```

For cyclic decomposition, variables begin and end are removed and the main calculation for-loop is modified:

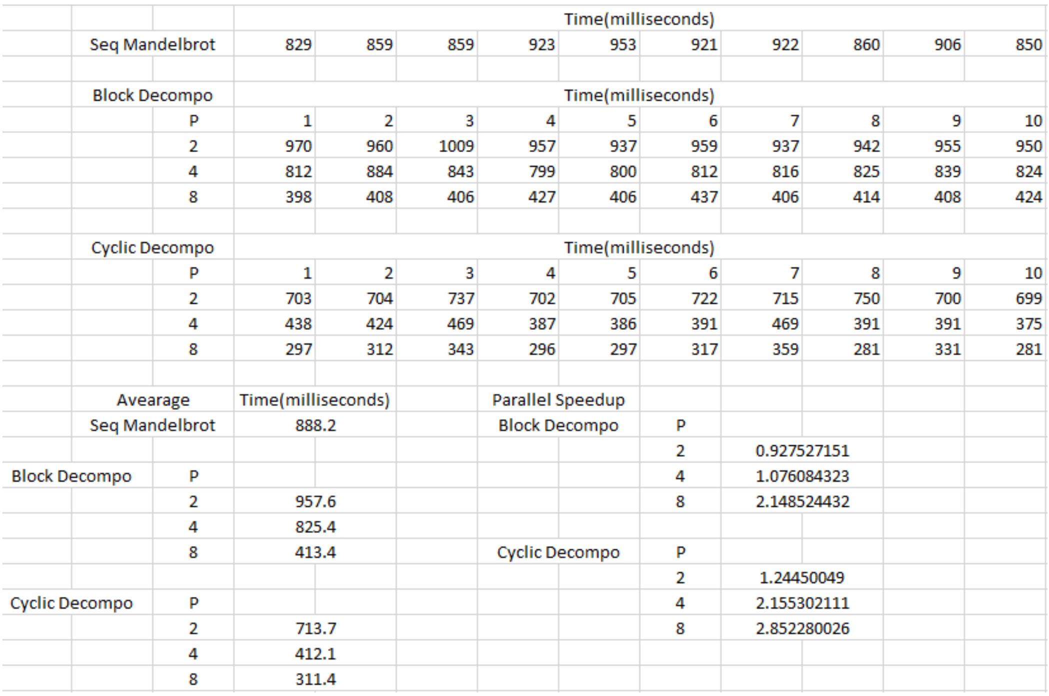
From:

```
for (int i = begin; i < end; i++) {
```

To:

```
for(int i = me ; i < N ; i+=P) {
```

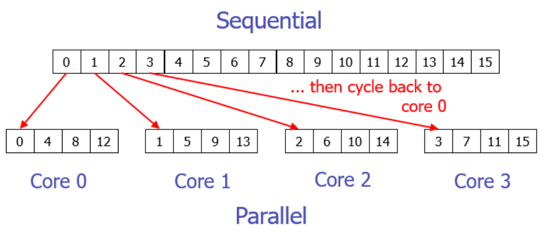
Here are the results of both the decomposition in comparison to the sequential version

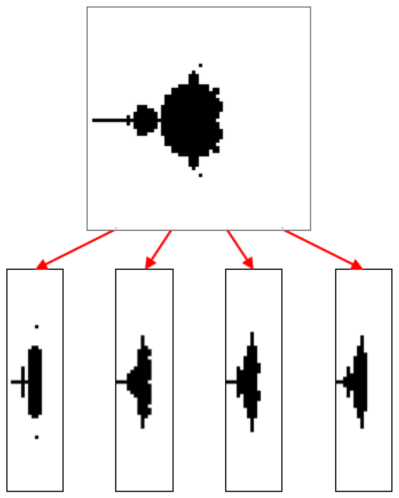


							Time(milliseconds)					
	Seq Mandelbrot		829	859	859	923	953	921	922	860	906	850
	Block Decompo						Time(milliseconds)					
		P	1	2	3	4	5	6	7	8	9	10
		2	970	960	1009	957	937	959	937	942	955	950
		4	812	884	843	799	800	812	816	825	839	824
		8	398	408	406	427	406	437	406	414	408	424
	Cyclic Decompo						Time(milliseconds)					
		P	1	2	3	4	5	6	7	8	9	10
		2	703	704	737	702	705	722	715	750	700	699
		4	438	424	469	387	386	391	469	391	391	375
		8	297	312	343	296	297	317	359	281	331	281
	Avearage		Time(milliseconds)			Parallel Speedup						
	Seq Mandelbrot		888.2			Block Decompo		P				
								2	0.927527151			
Block Decompo		P						4	1.076084323			
		2	957.6					8	2.148524432			
		4	825.4									
		8	413.4			Cyclic Decompo		P				
								2	1.24450049			
Cyclic Decompo		P						4	2.155302111			
		2	713.7					8	2.852280026			
		4	412.1									
		8	311.4									

From the results, block decomposition didn't provide much improvement over than the parallel version in last week's lab. When there were 8 threads, cyclic decomposition improved the parallel speedup to 2.8 nearly 3 times faster than the sequential version.

The reason behind all of this is the way how cyclic decomposition break down the workload of the Mandelbrot program. Instead of splitting the range into four equal parts, it cycles through the workload then added to each core, the process is similar to the diagram below.





With much more break-even of the set, it allows much better load balancing over any other attempted methods used before.

The second part was applying parallelism to simulation type programs and in this case, it was the cellular automaton called Conway's Game of Life.

Exercise

The whole process of parallelising the life program is similar to the last week's but instead of using cyclic decomposition this week's life program will use block decomposition with barrier synchronisation to counter the race conditions.

The run() is created to hold all the calculations, updates and the repaint().

The main calculation loop is moved into run() and variables begin and end are used in the break down the workload to each threads.

```
int begin = me * B;
int end = begin + B;
```

```
for (int i = begin; i < end; i++) {
```

```
for (int i = begin; i < end; i++) {
```

The synch() is used in-between the calculation and update as well as in-between update and the display.repaint().

```
static void synch() {
    try {
        barrier.await();
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}
```

The two synch() are essential because:

- The first one makes sure no thread goes to modify the cell array until all threads have finished using its old value to calculate their elements of the sums array.

- The second one makes sure that no thread goes on to calculate the sums array in the next generation until all threads have finished calculating the new values of the cells array.

Week 16 - Parallel Programs with Interacting Threads

The fourth week's lab script introduced a new example, Laplace's Equation, to see how to use barrier synchronisation across threads.

Exercise

The exercise is to parallelise the sequential version of Laplace's Equation.

The two variables being and end are added to divide the workload for each thread

```
int begin = me * B ;
int end = begin + B ;

if(me == 0)
    begin = 1 ;

if(me == P-1)
    end = N - 1 ;
```

Similar to last week's lab the ParallelLife program, synch() is used after each decomposed loop in the thread to climate race conditions.

The run() is created and holds all the calculations of phi and updates of phi as well as the repaint() after each loop is executed.

The threads' creation was similar to the one in week 15, a variable P holds the number of threads.

Here are the results:

		Time(milliseconds)									
		1	2	3	4	5	6	7	8	9	10
Seq Laplace		12571	11330	12356	12445	11456	12789	12837	12598	11987	11226
ParallelLaplace											
	P										
	2	6460	6157	6362	6898	6891	6751	6746	6304	6478	6529
	4	7281	7584	7793	7365	7428	7596	7123	7245	7321	7359
	8	11607	12036	11912	11770	11657	12681	12250	11948	12319	12484
Average		Time(milliseconds)									
Seq Laplace		12159.5									
ParallelLaplace											
	P										
	2	6557.6									
	4	7409.5									
	8	12066.4									
Parallel speedup											
	P										
	2	1.854260705									
	4	1.641068898									
	8	1.00771564									

Week 17 - Running MPJ Programs

The fifth week's lab script introduced a different kind of parallel programming, distributed memory parallel programming. All the ones in previous labs were shared memory programming using the thread class in Java.

The activities in the week lab script consist of running the previous program using MPJ such as the approximate pi program as well as simple program like hello world in Window's command prompt.

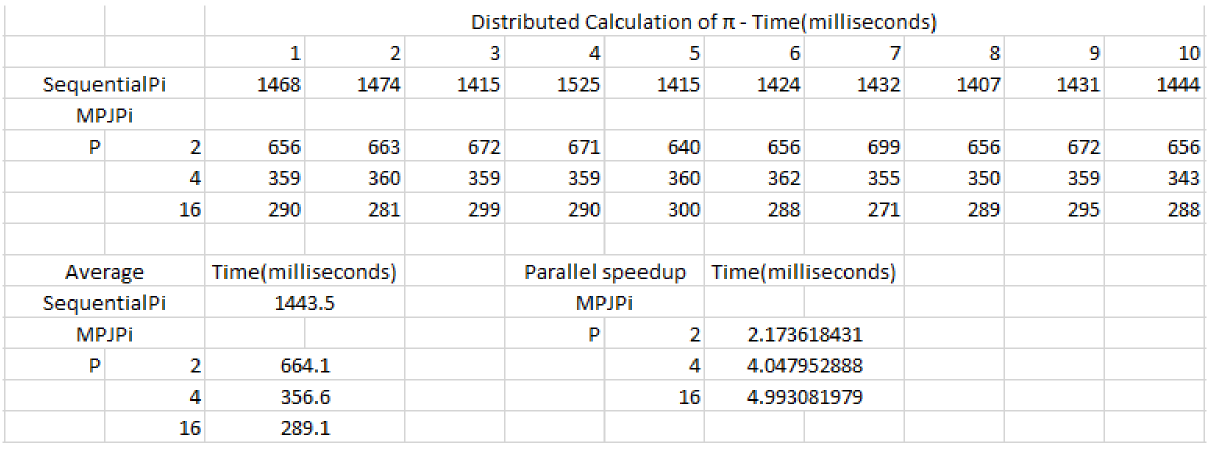
There are two modes that MPJ can run to parallelise programs, multicore mode, and cluster mode. In multicore mode, MPJ is simply running on the cores of the local processors and this is the default mode. The cluster mode is much more complicated to set but the principle is utilising the computation power of more than one machine to perform better parallel speed.

Exercise

This week's exercise is to achieve a parallel speedup of more than four for the MPJPI program.

In order to get the result, the -np value was set to 4 as well as 16 for better parallel speedup.

Here is the result:

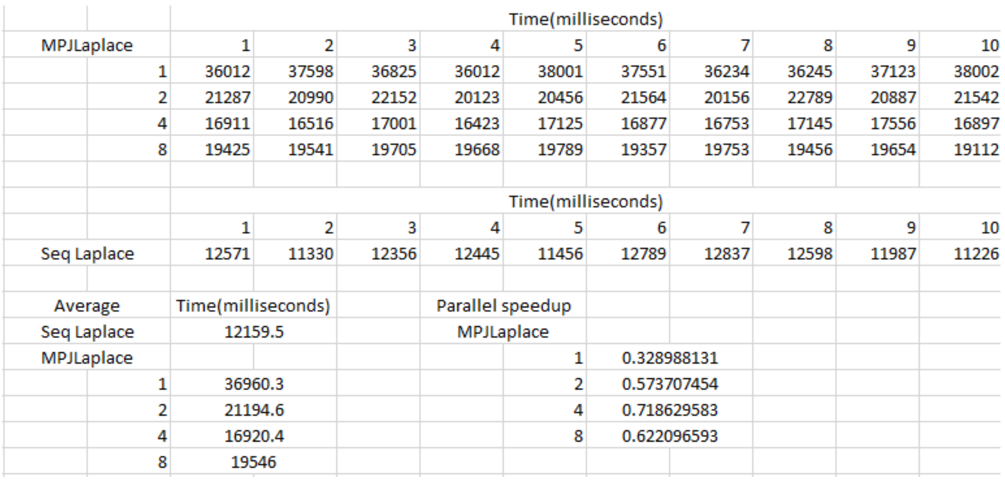


		Distributed Calculation of π - Time(milliseconds)									
		1	2	3	4	5	6	7	8	9	10
SequentialPi		1468	1474	1415	1525	1415	1424	1432	1407	1431	1444
MPJPi											
P	2	656	663	672	671	640	656	699	656	672	656
	4	359	360	359	359	360	362	355	350	359	343
	16	290	281	299	290	300	288	271	289	295	288
Average		Time(milliseconds)			Parallel speedup		Time(milliseconds)				
SequentialPi		1443.5			MPJPi						
MPJPi					P	2	2.173618431				
P	2	664.1				4	4.047952888				
	4	356.6				16	4.993081979				
	16	289.1									

Week 18 - MPJ Communication

The sixth week's lab script is concentrated on much more complex version of Laplace's Equation.

The new version of the Laplace's Equation added ghost regions and edge swap communications. It is slightly more complicated because a fix is needed for the boundary conditions at the edges of the system interact with the ghost regions.



		Time(milliseconds)									
MPJLaplace		1	2	3	4	5	6	7	8	9	10
	1	36012	37598	36825	36012	38001	37551	36234	36245	37123	38002
	2	21287	20990	22152	20123	20456	21564	20156	22789	20887	21542
	4	16911	16516	17001	16423	17125	16877	16753	17145	17556	16897
	8	19425	19541	19705	19668	19789	19357	19753	19456	19654	19112
		Time(milliseconds)									
		1	2	3	4	5	6	7	8	9	10
Seq Laplace		12571	11330	12356	12445	11456	12789	12837	12598	11987	11226
Average		Time(milliseconds)		Parallel speedup							
Seq Laplace		12159.5		MPJLaplace							
MPJLaplace					1	0.328988131					
	1	36960.3			2	0.573707454					
	2	21194.6			4	0.718629583					
	4	16920.4			8	0.622096593					
	8	19546									

From the results above, it shows that MPJLaplace with 1, 2, 4 or 8 threads didn't yield a perfect parallel speedup. It ran much slower than the sequential version.

		Time(milliseconds)									
		1	2	3	4	5	6	7	8	9	10
Seq Laplace		12571	11330	12356	12445	11456	12789	12837	12598	11987	11226
ParallelLaplace											
	P										
	2	6460	6157	6362	6898	6891	6751	6746	6304	6478	6529
	4	7281	7584	7793	7365	7428	7596	7123	7245	7321	7359
	8	11607	12036	11912	11770	11657	12681	12250	11948	12319	12484

Average		Time(milliseconds)
Seq Laplace		12159.5
ParallelLaplace		
	P	
	2	6557.6
	4	7409.5
	8	12066.4
Parallel speedup		
	P	
	2	1.854260705
	4	1.641068898
	8	1.00771564

The parallel version provided a much better parallel speedup than the MPJLaplace, nearly ran twice as fast as the sequential version.

Week 19 - An MPJ Task Farm

The seventh week's lab script provided an insight into a different and more dynamic approach of parallelism, task farm.

The example used in this week lab is Slow Mandelbrot, a Mandelbrot with its cut off value set to very high to increase the workload.

The activity of this week lab was to compare the new sequential version to the MPJ counterpart by benchmarking in both multicore mode as well as cluster mode. The aim is to explore the limits of parallel speedup by running it across multiple hosts.

		Time(milliseconds)									
		1	2	3	4	5	6	7	8	9	10
	Seq	43466	42218	43344	42569	41589	43516	41523	42217	42288	42667
	Parallel										
	2	46350	44321	42153	41567	45123	44653	41234	46541	42354	43298
	4	15983	17381	16401	16091	15830	16414	15663	15856	16107	16083
	8	13494	12765	13919	14292	13512	12960	13570	13237	12543	14048
3 PCs Cluster		5800	5539	5807	5313	4476	4535	4723	6734	6320	4449
Average		Time(milliseconds)			Parallel speedup						
	Seq	42539.7				Parallel					
	Parallel					2	0.972127132				
	2	43759.4				4	2.629007039				
	4	16180.9				8	3.166569897				
	8	13434			3 PCs Cluster		7.922321961				
3 PCs Cluster		5369.6									

From the results, the multicore version provided significant parallel speedup especially with 8 threads in comparison with the sequential version, but it is not as much as the one run in cluster mode.

The 3 PCs ran in cluster mode provided a much better parallel speedup, approximately eight times faster than the sequential version. It demonstrates the parallel speedup limit of a single machine with one quadcore processor and how much improvement can be obtained by utilising multiple processors.

Development project

Before the application of parallelism to the square matrix multiplication, the sequential version must be created first for a comparison of the performance between sequential and parallel as well as the calculation of parallel speedup.

Two matrices that are compatible for matrix multiplication when the first matrix's columns count matches the second matrix's rows count. The product will have the same number of rows as the first matrix and the same number of columns as the second matrix.

The calculation process is to take the first row of the first matrix multiply by the first column of the second matrix, element by element, and then repeat the same process to all the other columns of the second matrix. After the first row has been applied to every column in the second matrix, repeat the process above with the second row of the first matrix and so on until each row in the first matrix has multiplied to each column in the second matrix.

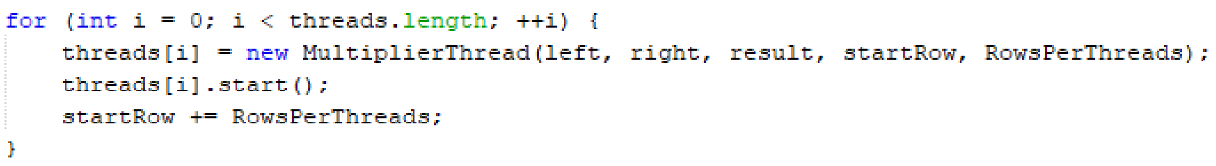
Left = first matrix   right = second matrix

The sequential version was easier to implement by utilising three for-loops, one for going through the left matrix's rows, one for iterating the right matrix's columns and the last one for all the iteration of values to produce the dot product.

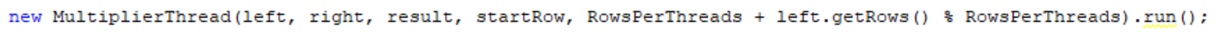
```
Matrix result = new Matrix(left.getRows(), right.getColumns());

for (int y = 0; y < left.getRows(); ++y) {
    for (int x = 0; x < right.getColumns(); ++x) {
        double sum = 0.0;

        for (int i = 0; i < left.getColumns(); ++i) {
            sum += left.get(i, y) * right.get(x, i);
        }

        result.set(x, y, sum);

    }
}
return result;
```

For parallelising the sequential version of the square matrix multiplication program, I broke down all the left matrix's row into sections and apply to each available thread on the machine.

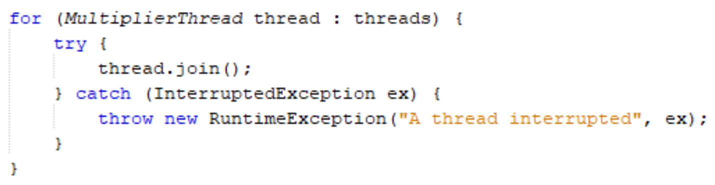
```
int RowsPerThreads = left.getRows() / numberOfThreads;
```



```
for (int i = 0; i < threads.length; ++i) {
    threads[i] = new MultiplierThread(left, right, result, startRow, RowsPerThreads);
    threads[i].start();
    startRow += RowsPerThreads;
}
```



```
new MultiplierThread(left, right, result, startRow, RowsPerThreads + left.getRows() % RowsPerThreads).run();
```

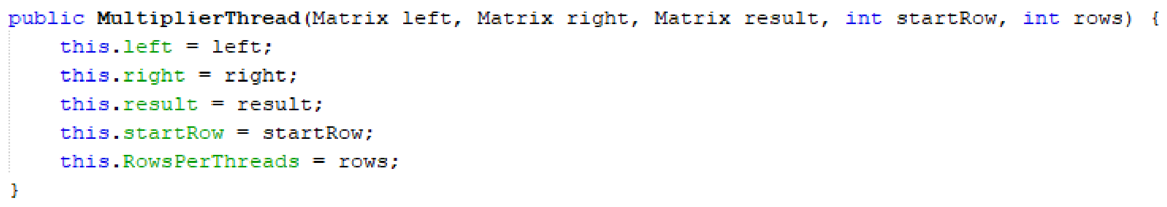


```
for (MultiplierThread thread : threads) {
    try {
        thread.join();
    } catch (InterruptedException ex) {
        throw new RuntimeException("A thread interrupted", ex);
    }
}
```

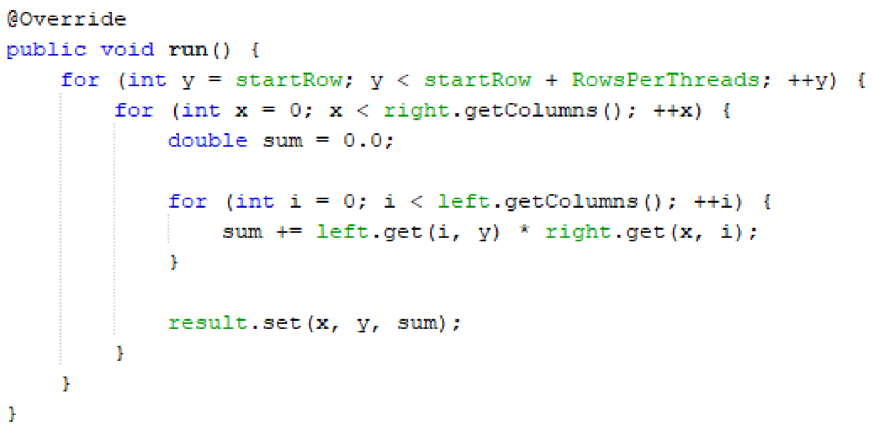
Each available thread will responsible for a section of the left matrix's rows with the use of variables startRow and RowsPerThreads. The startRow tells where to start and RowsPerThreads tells where to stop. The next thread will have a different startRow value by adding the RowsPerThreads to previous startRow value, so the range is split evenly as possible.

The modular division is used to ensure if there is any row left over from the RowsPerThread split is all cover in the thread.

The main calculation is similar to the sequential version but uses startRow and RowsPerThread to set the range for each thread in the first for-loop as it is responsible for the rows in the left matrix. This is inside the MultiplierThread class which is then extended from the Thread class.



```
public MultiplierThread(Matrix left, Matrix right, Matrix result, int startRow, int rows) {
    this.left = left;
    this.right = right;
    this.result = result;
    this.startRow = startRow;
    this.RowsPerThreads = rows;
}
```



```
@Override
public void run() {
    for (int y = startRow; y < startRow + RowsPerThreads; ++y) {
        for (int x = 0; x < right.getColumns(); ++x) {
            double sum = 0.0;

            for (int i = 0; i < left.getColumns(); ++i) {
                sum += left.get(i, y) * right.get(x, i);
            }

            result.set(x, y, sum);
        }
    }
}
```

The for-loop of Thread.join() is used to join all the calculations together to produce a full matrix.

The number of columns in the second matrix can be divided across all threads evenly to achieve better parallel speedup. In this way, if the device has 4 threads then each will responsible for a quarter of the output matrix

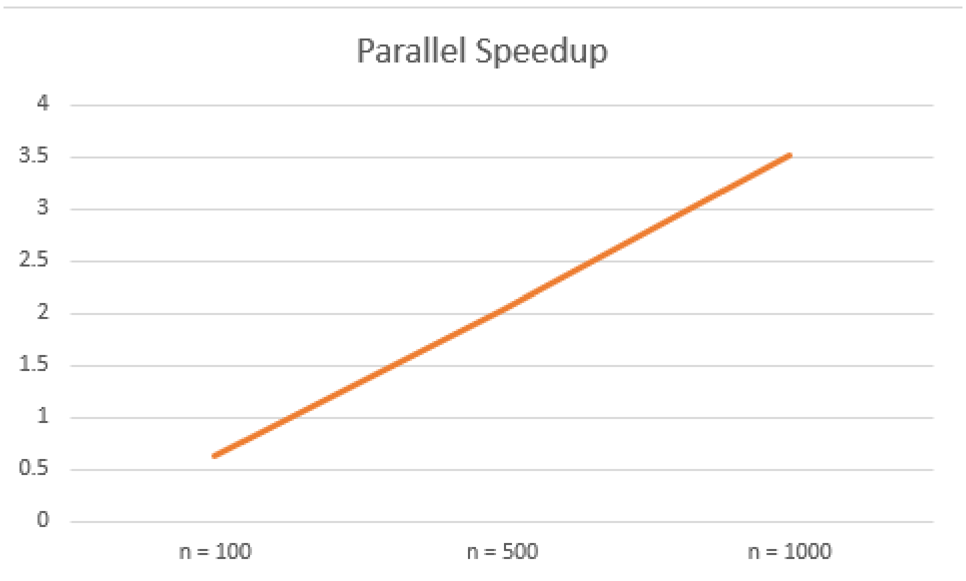
Here are the results of different size N :

	Time(milliseconds)									
n = 100	1	2	3	4	5	6	7	8	9	10
Sequential	16	15	16	16	17	17	16	16	17	17
Parallel	29	27	28	22	28	29	21	22	25	25

	Time(milliseconds)									
n = 500	1	2	3	4	5	6	7	8	9	10
Sequential	234	281	218	219	249	265	219	218	219	220
Parallel	109	125	125	125	125	109	109	110	109	109

	Time(milliseconds)									
n = 1000	1	2	3	4	5	6	7	8	9	10
Sequential	2792	2822	2797	2858	2832	2828	2832	2832	2799	2813
Parallel	781	818	812	800	806	797	814	782	815	797

Average	Time(milliseconds)		Parallel speedup	
n = 100			n = 100	0.63671875
Sequential	16.3		n = 500	2.027705628
Parallel	25.6		n = 1000	3.515956121
n = 500				
Sequential	234.2			
Parallel	115.5			
n = 1000				
Sequential	2820.5			
Parallel	802.2			



From the parallel speedup graph, it shows the parallel version provided significant parallel speedup when n is 500 or over. In addition, the linear line demonstrates the higher the size, the better the speedup as a result of this the parallel version yield a good efficiency.

Conclusion

The overall unit provided an insight into parallelism and its application in modern world problems.

The main findings in the unit are workload must be divided evenly across each thread in order to achieve significant parallel speedup with good efficiency and the parallel speedup difference between a cluster of machines and a single processor machine. Furthermore, multithreaded applications can achieve good parallel speedups when the problem size exceeds a certain threshold.

The communications between each processor nodes are important and without thread-safe precautions will lead to a parallel slowdown. Some problems called embarrassingly parallel problems (first few weekly lab assignments) do not require such communication so they won’t affect from this matter.