4003-531/4005-735 **Parallel Computing I** **Assignment 2**
Term 20102 **January 2, 2011**

Assignment 2

Due: January 17, 2010

Before starting, be sure to review the instructions for running Parallel Java on the RIT CS Parallel Computers (`http://www.cs.rit.edu/~ark/runningpj.shtml`). Remember, to execute programs on `parasite`, submit it to the job queue running on `paragon`. **Directly logging into `parasite` is not allowed.**

1. (20pts)

   Download `MandelbrotSetMethodSeq.java` and `MandelbrotSetMethodSmp.java` from the course website.

   (a) (10pts) Consider the following command-line arguments:

   ```
   3200 9600 -0.75 4 1200 2500 0.4 ms.pjg
   ```

   Measure the running time T for `MandelbrotSetMethodSeq` with these command-line arguments on `parasite` and `MandelbrotSetMethodSmp` with these command-line arguments on `parasite` using 1, 2, 4, and 8 processors and using `fixed`, `dynamic`, `dynamic(300)`, `guided`, and `guided(300)` schedules. Calculate *Speedup* and *Eff* as a function of K (the number of processors). Submit your results in a table organized as follows:

	K	T (msec)	*Speedup*	*Eff*
	seq		xxx	xxx
`fixed`	1			
`fixed`	2			
`fixed`	4			
`fixed`	8			
`dynamic`	1			
`dynamic`	2			
`dynamic`	4			
`dynamic`	8			
`dynamic(300)`	1			
`dynamic(300)`	2			
`dynamic(300)`	4			
`dynamic(300)`	8			
`guided`	1			
`guided`	2			
`guided`	4			
`guided`	8			
`guided(300)`	1			
`guided(300)`	2			
`guided(300)`	4			
`guided(300)`	8			

   Explain your results.

(b) (10pts) Consider the following command-line arguments:

```
3200 9600 -0.75 -4 1200 2500 0.4 ms.pjg
```

Measure the running time T for `MandelbrotSetMethodSeq` with these command-line arguments on `parasite` and `MandelbrotSetMethodSmp` with these command-line arguments on `parasite` using 1, 2, 4, and 8 processors and using `fixed`, `dynamic`, `dynamic(300)`, `guided`, and `guided(300)` schedules. Calculate *Speedup* and *Eff* for the different schedules as a function of K (the number of processors). Submit your results in a table organized as follows:

	K	T (msec)	*Speedup*	*Eff*
	seq		xxx	xxx
`fixed`	1			
`fixed`	2			
`fixed`	4			
`fixed`	8			
`dynamic`	1			
`dynamic`	2			
`dynamic`	4			
`dynamic`	8			
`dynamic(300)`	1			
`dynamic(300)`	2			
`dynamic(300)`	4			
`dynamic(300)`	8			
`guided`	1			
`guided`	2			
`guided`	4			
`guided`	8			
`guided(300)`	1			
`guided(300)`	2			
`guided(300)`	4			
`guided(300)`	8			

Explain your results.

Submission Submit a plain text file named `hw2-1.txt` or a PDF file named `hw2-1.pdf`. The `hw2-1` file should contain the tabulated results and an explaination for part (a) and the tabulated results and an explaination for part (b).

2. (20pts)

Consider the following table that records the running time T_i for each of the 24 iterations i of a parallel `for`-loop. Note that the parallel `for`-loop's running time on one processor is $T_p(1) = 1777$.

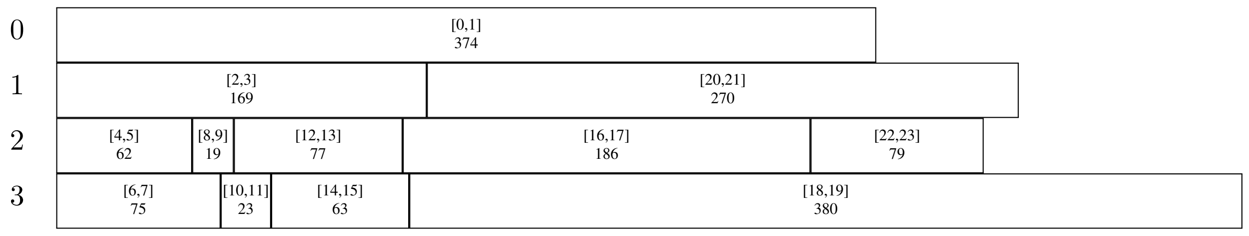
i	0	1	2	3	4	5	6	7	8	9	10	11
T_i	200	174	113	56	30	32	40	35	17	2	3	20

i	12	13	14	15	16	17	18	19	20	21	22	23
T_i	37	40	31	32	63	123	181	199	166	104	50	29

Calculate load balance B using `fixed`, `dynamic`, `dynamic(2)`, `guided`, and `guided(2)` schedules as a function of K (the number of processors).

To compute the parallel `for`-loop's running time on K processors ($T_p(K)$), simulate the behavior of the schedule, assuming that there is no parallel `for`-loop overhead. Specifically, assume that the K processors are assigned the labels 0 through $K - 1$. Assign the first chunk of the iterations to processor 0, the second chunk of the iterations to processor $1, \ldots$, and the K^{th} chunk of the iterations to processor $K - 1$. The instant a processor completes its chunk of the iterations, it is assigned the next available chunk of the iterations. (If multiple processors complete their chunks of the iterations at the same time, then assign the next available chunk of the iterations to the processor with the smallest label.)

For example, the behavior of the `dynamic(2)` schedule on 4 processors can be visualized as:



Therefore, $T_p(4) = 541$ and $B = 1.22$.

Submit your results in a table organized as follows:

	K	$T_p(K)$	B
`fixed`	2		
`fixed`	4		
`fixed`	8		
`dynamic`	2		
`dynamic`	4		
`dynamic`	8		
`dynamic(2)`	2		
`dynamic(2)`	4	541	1.22
`dynamic(2)`	8		
`guided`	2		
`guided`	4		
`guided`	8		
`guided(2)`	2		
`guided(2)`	4		
`guided(2)`	8		

Submission Submit a plain text file named `hw2-2.txt` or a PDF file named `hw2-2.pdf`. The `hw2-2` file should contain the tabulated results.

3. (20pts)

   It can be proven that all points of the Mandelbrot Set lie within a square of side-length 4 that is centered at the origin and aligned with the real and imaginary axes. Or, to put it another way, any complex point $c = x + y \cdot i$ that belongs to the Mandelbrot must satisfy $-2 < x < 2$ and $-2 < y < 2$. Using this fact, write Java programs called `MSAreaSeq` (a sequential program) and `MSAreaSmp` (a parallel program) that estimate the area of the Mandelbrot Set using a Monte Carlo algorithm. [1] Both the `MSAreaSeq` and `MSAreaSmp` programs must take the following command-line arguments:

   - *seed*: random seed (a `long`)
   - *maxiter*: maximum number of iterations (an `int`)
   - *N*: the number of random points (a `long`)

   For instance, here is a sample execution of `MSAreaSeq`:

```
[mtf@fenrir code]\$ java MSAreaSeq 54321 5000 20000000
MS area = 1.5060232
48065 msec
```

   Note that a correct `MSAreaSmp` program will produce exactly the same area when run with the same command-line arguments.

   Considering `MSAreaSmp`, explain which parallel `for`-loop schedule should be used. Be sure to use this parallel `for`-loop schedule when measuring the running time of `MSAreaSmp`.

   Measure the running time T for `MSAreaSeq` with command-line arguments `54321 8191 20000000` on `parasite` and `MSAreaSmp` with the same command-line arguments on `parasite` using 1, 2, 4, and 8 processors. Calculate *Speedup*, *Eff*, and *EDSF* as a function of K (the number of processors). Submit your results in a table organized as follows:

K	T (msec)	*Speedup*	*Eff*	*EDSF*
seq		xxx	xxx	xxx
1				xxx
2				
4				
8				

   Submission Submit `MSAreaSeq.java`, `MSAreaSmp.java`, and a plain text file named `hw2-3.txt` or a PDF file named `hw2-3.pdf`. The `hw2-3` file should contain an explaination of the parallel `for`-loop schedule used and the tabulated results.

[1]The true area of the Mandelbrot Set is not known, but it is estimated to be 1.50659177 ± 0.00000008.[1]

4. (20pts)

   Write a Java program called `EncryptFileSeq` that encrypts a file using the Advanced Encryption Standard (AES). The `EncryptFileSeq` program must take the following command-line arguments:

   - *key*: key (256-bit hexadecimal number)
   - *inFileName*: input file name
   - *outFileName*: output file name

   Use `edu.rit.crypto.blockcipher.AES256Cipher` to perform the AES encryption. Note that `edu.rit.crypto.blockcipher.AES256Cipher` uses a 128-bit (16-byte) block length and a 256-bit (32-byte) key length. If the size of input file is not a multiple of 16-bytes, then the last block should be padded with zeros. Note that the size of the output file will be a multiple of 16-bytes.

   The nature of a block cipher is such that each block of the input file may be encrypted independently. Hence, a natural way to parallelize the above program is to perform multiple encryptions in parallel. However, it would be unreasonable to read the entire input file into memory, encrypt all of the blocks in parallel, and write the entire encrypted output file from memory. Instead, it is more reasonable to loop, reading a chunk of the input file into memory, encrypting all of the blocks in the chunk in parallel, and writing the encrypted chunk of the output file from memory. Write a Java program called `EncryptFileSmpNoOverlap` that follows the above description. The `EncryptFileSmpNoOverlap` program must take the following command-line arguments:

   - *key*: key (256-bit hexadecimal number)
   - *inFileName*: input file name
   - *outFileName*: output file name
   - *chunkSize*: number of 16-byte blocks in a chunk

   A limitation of the above parallel program is that reading a chunk of the input file into memory and writing the encrypted chunk of the output file from memory might contribute to a large sequential fraction. Hence, a way to further parallelize the above parallel program is to, in parallel, read one chunk of the input file into memory, encrypt all of the blocks in another chunk (in parallel), and write yet another encrypted chunk of the output file from memory. Write a Java program called `EncryptFileSmpOverlap` the follows the above description. The `EncryptFileSmpOverlap` program must take the following command-line arguments:

   - *key*: key (256-bit hexadecimal number)
   - *inFileName*: input file name
   - *outFileName*: output file name
   - *chunkSize*: number of 16-byte blocks in a chunk

   Measure the running time T for `EncryptFileSeq` with the file `/usr/local/pub/mtf/pc1-20102/assignment2/pg100x20.txt`[2] on `parasite` and `EncryptFileSmpNoOverlap` and `EncryptFileSmpNoOverlap` with the same file and using a chunk size of 65536 on `parasite` using 1, 2, 4, and 8 processors. Calculate *Speedup*, *Eff*, and *EDSF* as

[2]Twenty copies of *The Complete Works of William Shakespeare* from Project Gutenberg.

a function of K (the number of processors). Submit your results in a table organized as follows:

K	T (msec)	*Speedup*	*Eff*	*EDSF*
seq		xxx	xxx	xxx
no overlap				
1				xxx
2				
4				
8				
overlap				
1				xxx
2				
4				
8				

Notes

- Include *all* input and output operations (e.g., opening and closing files) in the running time of the programs.
- Use `java.io.BufferedInputStream` and `java.io.BufferedOutputStream`; unbuffered I/O leads to an unacceptably large sequential fraction.
- Avoid unnecessary allocations by reusing arrays allocated for blocks and chunks.
- Although the description of `EncryptFileSmpOverlap` suggests three parallel sections, is acceptable to use two parallel sections, one that performs both input and output and another that peforms (parallel) encryptions. In fact, this is likely to be more efficient, since the operating system may not be able to effectively perform both input and output in parallel.
- When running the programs, you will most likely observe very different behaviors depending upon the file system on which the input and output files reside. For example, on `parasite`, `/usr/local/pub/` and `/home/` are Network File System (NFS) file systems, with significant I/O overhead. In contrast, a personal machine would likely use a local disk file system, with only moderate I/O overhead.
- When measuring the running time of the programs, use the path `/usr/local/pub/mtf/pc1-20102/assignment2/pg100x20.txt` as the input file. Using the NFS file system will demonstrate the effects of overlapping.
- Note that a program submitted to the job queue running on `paragon` with an output file in `/tmp/` will result in a file being created in `parasite`'s `/tmp/` directory, not in `paragon`'s `/tmp/` directory. Since directly logging into `parasite` is not allowed, such a file will not be accesible. Therefore, when measuring the running time of the programs, do not use a file in `/tmp/` as the output file.
- The reference solution for the `EncryptFileSeq` program is 84 lines, for the `EncryptFileSmpNoOverlap` program is 145 lines, and for the `EncryptFileSmpOverlap` is 189 lines.
- The file `/usr/local/pub/mtf/pc1-20102/assignment2/key.txt` is a 256-bit hexadecimal number, suitable for use as an AES encryption key. The files `/usr/local/pub/mtf/pc1-20102/assignment2/pg100.enc`, `/usr/local/pub/mtf/pc1-20102/assignment2/pg100x5.enc`, `/usr/local/pub/mtf/pc1-20102/assignment2/pg100x10.enc`, and `/usr/local/pub/mtf/pc1-20102/assignment2/pg100x20.enc` are the encryptions of the corresponding `.txt` files using the above key.

Submission Submit `EncryptFileSeq.java`, `EncryptFileSmpNoOverlap.java`, `EncryptFileSmpOverlap.java`, and a plain text file named `hw2-4.txt` or a PDF file named `hw2-4.pdf`. The `hw2-4` file should contain the tabulated results.

Submission

Submit a single ZIP file named `hw2.zip` to the `Homework 2` Dropbox on MyCourses by the due date. The `hw2.zip` file should contain:

- `hw2-1.txt` or `hw2-1.pdf`
- `hw2-2.txt` or `hw2-2.pdf`
- `MSAreaSeq.java`
- `MSAreaSmp.java`
- `hw2-3.txt` or `hw2-3.pdf`
- `EncryptFileSeq.java`
- `EncryptFileSmpNoOverlap.java`
- `EncryptFileSmpNoOverlap.java`
- `hw2-4.txt` or `hw2-4.pdf`

The `hw2.zip` file should contain no additional files.

References

[1] Robert Munafo. Area of the Mandelbrot Set. `http://www.mrob.com/pub/muency/areaofthemandelbrotset.html`.

Document History

January 2, 2011
Original version