

Michael Xavier Canonizado
BSCS 3A - Operating Systems
Machine Exercise 4 - 5

Machine Exercise 4:

```
$ java MachineExercise4
Generating matrix 1 of size 50
Generating matrix 2 of size 50

Serial multiplication took 2.206 ms
Parallel multiplication with 1 threads took 4.419 ms (Speedup: 0.50x)
Parallel multiplication with 2 threads took 2.071 ms (Speedup: 1.07x)
Parallel multiplication with 4 threads took 1.318 ms (Speedup: 1.67x)
Parallel multiplication with 8 threads took 2.940 ms (Speedup: 0.75x)

michaelcanonizado@LAPTOP-USIQ9ECR MINGW64 /c/Michael Root/Code/Repositories/bid
systems/machine-exercise-4-5/machine-exercise-4 (main)
```

```
$ java MachineExercise4
Generating matrix 1 of size 1000
Generating matrix 2 of size 1000

Serial multiplication took 1218.492 ms
Parallel multiplication with 1 threads took 1209.616 ms (Speedup: 1.01x)
Parallel multiplication with 2 threads took 599.969 ms (Speedup: 2.03x)
Parallel multiplication with 4 threads took 289.972 ms (Speedup: 4.20x)
Parallel multiplication with 8 threads took 167.310 ms (Speedup: 7.28x)

michaelcanonizado@LAPTOP-USIQ9ECR MINGW64 /c/Michael Root/Code/Repositories/bid
systems/machine-exercise-4-5/machine-exercise-4 (main)
$ |
```

```
michaelcanonizado@LAPTOP-USIQ9ECR MINGW64 /c/Michael Root/Code/Repositories/bid
systems/machine-exercise-4-5/machine-exercise-4 (main)
$ java MachineExercise4
Generating matrix 1 of size 5000
Generating matrix 2 of size 5000

Serial multiplication took 1012132.706 ms
Parallel multiplication with 1 threads took 1088048.134 ms (Speedup: 0.93x)
Parallel multiplication with 2 threads took 597916.128 ms (Speedup: 1.69x)
Parallel multiplication with 4 threads took 440461.787 ms (Speedup: 2.30x)
Parallel multiplication with 8 threads took 233365.406 ms (Speedup: 4.34x)

michaelcanonizado@LAPTOP-USIQ9ECR MINGW64 /c/Michael Root/Code/Repositories/bid
systems/machine-exercise-4-5/machine-exercise-4 (main)
$ |
```

Observation: Multithreading really does increase the efficiency of the algorithm, but on small matrix sizes, the overhead of thread creation hinders the benefit of multithreading. The benefit of multithreading is seen on larger matrix sizes, as seen on the calculated speedup.


Machine Exercise 5:

```
ems/machine-exercise-4-5/machine-exercise-5$ ./me5

=====| Asynchronous Cancellation |=====
[Main] Sending cancel request to asynchronous thread...
[Cleanup] Closing file
[Main] Asynchronous thread has been canceled.

=====| Demonstrating Deferred Cancellation |=====
[Main] Sending cancel request to deferred thread...
[Cleanup] Closing file
[Main] Deferred thread has been canceled.

Check 'async_output.txt' and 'deferred_output.txt' to see the difference.
michaelcanonizado@LAPTOP-USIQ9ECR:/mnt/c/Michael Root/Code/Repositories/bi
ems/machine-exercise-4-5/machine-exercise-5$ |
```

machine-exercise-5 >  async_output.txt

```
1 Writing line 0
2 Start critical section 0
3 End critical section 0
4 Writing line 1
5 Start critical section 1
6 End critical section 1
7 Writing line 2
8 Start critical section 2
9 End critical section 2
10 Writing line 3
11 Start critical section 3
12 End critical section 3
13 Writing line 4
14 Start critical section 4
15 |
```

machine-exercise-5 >  deferred_output.txt

```
1 Writing line 0
2 Start critical section 0
3 End critical section 0
4 Writing line 1
5 Start critical section 1
6 End critical section 1
7 Writing line 2
8 Start critical section 2
9 End critical section 2
10 Writing line 3
11 Start critical section 3
12 End critical section 3
13 Writing line 4
14 Start critical section 4
15 End critical section 4
16 |
```

Observation: In the example above, I programmed the proof to simulate start and ends of "critical" operations. We can clearly see that we can add a safe-point to where deferred cancellation exits safely. Compared to async that just terminates at any point, even in the middle of a critical operation.