EECE5640 Homework 2

Question 1

Overview:

Each philosopher invited to this program is allowed to eat only if they are holding two forks; otherwise, the philosopher must think before eating. It is a threaded program that uses forks and mutex locks to represent threads as for philosophers. Through semaphores, the output is managed. When all of philosophers have eatten, we exit the thread and the program.

Detailed description:

I made the decision to use mutex locks as forks in my solution and semaphores to secure the output. Each philosopher has a unique semaphore and is represented as a single thread. Additionally, every philosopher has a corresponding lock (5 philosophers = 5 locks). I have a while loop in my algorithm that will continue to run until all of the philosophers have finished (they have each thought and eaten at least once). I use trylock in the while loop to lock the mutexes. I try the left fork first, and if that works, I try the right fork. The philosopher then eats, rests for one second, and then wakes up. The philosopher will think in the event that either of the trylocks fails. They wait one second after eating or sleeping before beginning the cycle all over again. To ensure that the first or second philosophers do not always end up being able to eat (as I found issue with that while testing).

a.)

My solution is unaffected by the number of philosophers, and it shouldn't matter whether there are odd or even numbers of threads because the code doesn't depend on the number of threads being even or odd.

The test results that only 3 forks are placed in the center of the table, but each philosopher still needs to acquire 2 forks to eat are as follow

```
-bash-4.2$ g++ -o philosophersA philosophersA.cpp -lpthread -std=c++11
-bash-4.2$ ./philosophersA 3


-----TABLE STATE-----

Philosopher 0 is eating
- has left fork: No.2
- has right fork: No.0

Philosopher 1 is thinking
- waiting for left fork: No.0
- waiting for right fork: No.1

Philosopher 2 is thinking
- waiting for left fork: No.1
- waiting for right fork: No.2

```

```


-----TABLE STATE-----

Philosopher 0 is thinking
- waiting for left fork: No.2
- waiting for right fork: No.0

Philosopher 1 is eating
- has left fork: No.0
- has right fork: No.1

Philosopher 2 is thinking
- waiting for left fork: No.1
- waiting for right fork: No.2



-----TABLE STATE-----

Philosopher 0 is thinking
- waiting for left fork: No.2
- waiting for right fork: No.0

Philosopher 1 is thinking
- waiting for left fork: No.0
- waiting for right fork: No.1

Philosopher 2 is eating
- has left fork: No.1
- has right fork: No.2



-----TABLE STATE-----

Philosopher 0 is thinking
- waiting for left fork: No.2
- waiting for right fork: No.0

```

Please see the detail of codes in the folder q1/philosophersA.cpp.

b.)

Each philosopher invited to this program is allowed to eat only if they are holding two forks; otherwise, the philosopher must think before eating. It is a threaded program that uses forks and mutex locks to represent threads as for philosophers. Through semaphores, the output is managed. The mutexes are initialized with the first philosopher's scheduling priority in mind.

The philosopher who receives priority over the others will either get priority over the mutex or simply eat more frequently. If a thread is assigned scheduling priority and mutexes are initialized with an attribute instructing them to follow thread priority, the mutex will permit that thread to take the mutex if it becomes available before any other thread. The prioritized thread (thread 1) consumes food more frequently than the other threads, according to this program. The last philosopher would never eat if it didn't occasionally have mutexes.

```
-bash-4.2$ g++ -o philosophersB philosophersB.cpp -lpthread -std=c++11
-bash-4.2$ ./philosophersB 5


-----TABLE STATE-----

Philosopher 0 is eating
- has left fork: No.4
- has right fork: No.0

Philosopher 1 is thinking
- waiting for left fork: No.0
- waiting for right fork: No.1

Philosopher 2 is eating
- has left fork: No.1
- has right fork: No.2

Philosopher 3 is thinking
- waiting for left fork: No.2
- waiting for right fork: No.3

Philosopher 4 is thinking
- waiting for left fork: No.3
- waiting for right fork: No.4



-----TABLE STATE-----

Philosopher 0 is thinking
- waiting for left fork: No.4
- waiting for right fork: No.0

Philosopher 1 is thinking
- waiting for left fork: No.0
- waiting for right fork: No.1

Philosopher 2 is thinking
- waiting for left fork: No.1
- waiting for right fork: No.2

Philosopher 3 is eating
- has left fork: No.2
- has right fork: No.3

Philosopher 4 is thinking
```

```
- waiting for left fork: No.3
- waiting for right fork: No.4



-----TABLE STATE-----

Philosopher 0 is eating
- has left fork: No.4
- has right fork: No.0

Philosopher 1 is thinking
- waiting for left fork: No.0
- waiting for right fork: No.1

Philosopher 2 is thinking
- waiting for left fork: No.1
- waiting for right fork: No.2

Philosopher 3 is thinking
- waiting for left fork: No.2
- waiting for right fork: No.3

Philosopher 4 is thinking
- waiting for left fork: No.3
- waiting for right fork: No.4



-----TABLE STATE-----

Philosopher 0 is thinking
- waiting for left fork: No.4
- waiting for right fork: No.0

Philosopher 1 is eating
- has left fork: No.0
- has right fork: No.1

Philosopher 2 is thinking
- waiting for left fork: No.1
- waiting for right fork: No.2

Philosopher 3 is eating
- has left fork: No.2
- has right fork: No.3

Philosopher 4 is thinking
- waiting for left fork: No.3
- waiting for right fork: No.4



-----TABLE STATE-----
```

```
Philosopher 0 is thinking
- waiting for left fork: No.4
- waiting for right fork: No.0

Philosopher 1 is thinking
- waiting for left fork: No.0
- waiting for right fork: No.1

Philosopher 2 is thinking
- waiting for left fork: No.1
- waiting for right fork: No.2

Philosopher 3 is thinking
- waiting for left fork: No.2
- waiting for right fork: No.3

Philosopher 4 is eating
- has left fork: No.3
- has right fork: No.4



-----TABLE STATE-----

Philosopher 0 is thinking
- waiting for left fork: No.4
- waiting for right fork: No.0
```

Please see the detail of codes in the folder q1/philosophersB.cpp.

c.)

What happens to your solution if the philosophers change which fork is acquired first (i.e., the fork on the left or the right) on each pair of requests?

The outcome won't change if the forks are picked in a different order because every time we check whether a philosopher can eat or not, a philosopher is attempting to pick a fork. The outcome is identical to that of question 1 as long as we ensure that philosopher receives the forks in the same order (for example, left first, right second or right first, left second).

Question 2

The information of the system I am running on is as follow. And the number of hardware threads available is 16.

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
```

```
CPU(s):                16
On-line CPU(s) list:   0-15
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Model name:            Intel(R) Xeon(R) CPU           E5620  @ 2.40GHz
Stepping:              2
CPU MHz:               1600.000
CPU max MHz:           2400.0000
CPU min MHz:           1600.0000
BogoMIPS:              4799.73
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15
```

a.) Pthreads

```
-bash-4.2$ g++ -o MonteCarloPi MonteCarloPi.cpp -lpthread -std=c++11
-bash-4.2$ ./MonteCarloPi
Number of threads: 1
number of “darts” thrown: 5000000
Create pthread success!
Thread 0 joined
pi_estimate: 3.14288
Time taken by function: 0.129031seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 2
number of “darts” thrown: 5000000
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
pi_estimate: 3.14218
Time taken by function: 0.064874seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 4
number of “darts” thrown: 5000000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
pi_estimate: 3.14191
Time taken by function: 0.034198seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 8
number of “darts” thrown: 5000000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
Thread 4 joined
Thread 5 joined
Thread 6 joined
Thread 7 joined
pi_estimate: 3.13978
Time taken by function: 0.017672seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 16
number of “darts” thrown: 5000000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
Thread 4 joined
Thread 5 joined
Thread 6 joined
Thread 7 joined
Thread 8 joined
Thread 9 joined
Thread 10 joined
Thread 11 joined
Thread 12 joined
Thread 13 joined
Thread 14 joined
Thread 15 joined
pi_estimate: 3.14231
Time taken by function: 0.012849seconds
```

Time

1 thread	2 thread	4 thread	8 thread	16 thread
0.129031s	0.064874s	0.034198s	0.017672s	0.012849s

Speedup

1 thread	2 thread	4 thread	8 thread	16 thread
1	1.9889	3.7731	7.3014	10.0421

Pi Estimate

1 thread	2 thread	4 thread	8 thread	16 thread
3.14288	3.14218	3.14191	3.13978	3.14231

Trends that I am seeing and its explanation:

When my input, darts, is large enough, the calculated value of pi will be relatively accurate. In my test, I set the darts as 5000000, and when the number of my threads increases, the speedup also increases.

b.) OpenMP

```
-bash-4.2$ g++ -o MonteCarloPiOmp MonteCarloPiOmp.cpp -fopenmp
-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 1
number of “darts” thrown: 5000000
Final Estimation of Pi = 3.14164
Time taken by function: 0.135973seconds

-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 2
number of “darts” thrown: 5000000
Final Estimation of Pi = 3.142
Time taken by function: 0.0698787seconds
```

```
-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 4
number of “darts” thrown: 5000000
Final Estimation of Pi = 3.14109
Time taken by function: 0.0347366seconds

-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 8
number of “darts” thrown: 5000000
Final Estimation of Pi = 3.14131
Time taken by function: 0.0181777seconds

-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 16
number of “darts” thrown: 5000000
Final Estimation of Pi = 3.14096
Time taken by function: 0.016216seconds
```

Time

1 thread	2 thread	4 thread	8 thread	16 thread
0.135973s	0.0698787s	0.0347366s	0.0181777s	0.016216s

Speedup

1 thread	2 thread	4 thread	8 thread	16 thread
1	1.9458	3.9144	7.4805	8.3851

Pi Estimate

1 thread	2 thread	4 thread	8 thread	16 thread
3.14164	3.142	3.14109	3.14131	3.14096

When I implement my thought through openMP, the trends that I observed are same with the pthread approach. But in my test, openMP's speedup is a little bit lower than pthread.

c.) Weak and Strong Scalability

Evaluate scalability of Pthread Approach

Evaluate strong scalability:

According to the Amdahl’s Law, we are going to give a fixed input to evaluate strong scaling. So, I will give a numbem 20000 as our input. The result of pthread approach is as follow.

```
-bash-4.2$ ./MonteCarloPi
Number of threads: 1
```

```
number of “darts” thrown: 20000
Create pthread success!
Thread 0 joined
pi_estimate: 3.1448
Time taken by function: 0.000931seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 2
number of “darts” thrown: 20000
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
pi_estimate: 3.1484
Time taken by function: 0.000688seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 4
number of “darts” thrown: 20000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
pi_estimate: 3.1728
Time taken by function: 0.000566seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 8
number of “darts” thrown: 20000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
Thread 4 joined
Thread 5 joined
Thread 6 joined
Thread 7 joined
pi_estimate: 3.1424
Time taken by function: 0.000662seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 16
```

```
number of “darts” thrown: 20000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
Thread 4 joined
Thread 5 joined
Thread 6 joined
Thread 7 joined
Thread 8 joined
Thread 9 joined
Thread 10 joined
Thread 11 joined
Thread 12 joined
Thread 13 joined
Thread 14 joined
Thread 15 joined
pi_estimate: 3.1136
Time taken by function: 0.001955seconds
```

Time

1 thread	2 thread	4 thread	8 thread	16 thread
0.000931s	0.000688s	0.000566s	0.000662s	0.001955s

Speedup

1 thread	2 thread	4 thread	8 thread	16 thread
1	1.3532	1.6449	1.4063	0.4762

As we can see, when we increase the number of thread until 4, my algorithm's strong scalability reaches its maximum.

Evaluate weak scalability:

According to the Gustafson's Law, we are going to give a varying input and threads, then get the efficiency to evaluate weak scaling.

```
-bash-4.2$ ./MonteCarloPi
Number of threads: 1
number of “darts” thrown: 50000
Create pthread success!
Thread 0 joined
pi_estimate: 3.13352
Time taken by function: 0.001761seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 2
number of “darts” thrown: 100000
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
pi_estimate: 3.13848
Time taken by function: 0.001793seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 4
number of “darts” thrown: 200000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
pi_estimate: 3.14032
Time taken by function: 0.001857seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 8
number of “darts” thrown: 400000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
Thread 4 joined
Thread 5 joined
Thread 6 joined
Thread 7 joined
pi_estimate: 3.14336
Time taken by function: 0.002201seconds

-bash-4.2$ ./MonteCarloPi
Number of threads: 16
number of “darts” thrown: 800000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
Thread 4 joined
Thread 5 joined
Thread 6 joined
Thread 7 joined
Thread 8 joined
Thread 9 joined
Thread 10 joined
Thread 11 joined
Thread 12 joined
Thread 13 joined
Thread 14 joined
Thread 15 joined
pi_estimate: 3.14616
Time taken by function: 0.003391seconds
```

Time

1 thread, 50000	2 thread, 100000	4 thread, 200000	8 thread, 400000	16 thread, 800000
0.001761s	0.001793s	0.001857s	0.002201s	0.003391s

Efficiency

1 thread	2 thread	4 thread	8 thread	16 thread
1	0.9821	0.9483	0.80000	0.5193

I noticed the implementation's weak scaling property by expanding the darts number in accordance with the growth in the number of threads, where each doubling of the input size results in a doubling of the number of threads. Before using 4 threads, I saw a comparable speed, indicating that the technique has decent weak scalability. I did see a speed decline after 4 threads, which may indicate that 4 threads is also the upper limit of weak scaling for this approach.

Evaluate scalability of OpenMP Approach

Evaluate strong scalability:

The steps are same with above. The result of openMP approach is as follow.

```
-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 1
number of “darts” thrown: 20000
Final Estimation of Pi = 3.1372
Time taken by function: 0.000657447seconds

-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 2
number of “darts” thrown: 20000
Final Estimation of Pi = 3.1472
Time taken by function: 0.000498902seconds

-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 4
number of “darts” thrown: 20000
Final Estimation of Pi = 3.1498
Time taken by function: 0.000363699seconds

-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 8
number of “darts” thrown: 20000
Final Estimation of Pi = 3.1494
Time taken by function: 0.000376795seconds

-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 16
number of “darts” thrown: 20000
Final Estimation of Pi = 0.1986
Time taken by function: 0.00111385seconds
```

Time

1 thread	2 thread	4 thread	8 thread	16 thread

1 thread	2 thread	4 thread	8 thread	16 thread
0.000657447s	0.000498902s	0.000363699s	0.000376795s	0.00111385s

Speedup

1 thread	2 thread	4 thread	8 thread	16 thread
1	1.3178	1.8077	1.7448	0.5902

Like in the experiments I did above with constant input data size that equals to 20000, I found that performance clearly decreased when I reached 16 threads. I can therefore say that the strong scalability of my algorithm peaks at about 8 threads.

Evaluate weak scalability:

```
-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 1
number of “darts” thrown: 50000
Final Estimation of Pi = 3.14416
Time taken by function: 0.00150691seconds

-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 2
number of “darts” thrown: 100000
Final Estimation of Pi = 3.14484
Time taken by function: 0.00160127seconds

-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 4
number of “darts” thrown: 200000
Final Estimation of Pi = 3.14238
Time taken by function: 0.00165135seconds

-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 8
number of “darts” thrown: 400000
Final Estimation of Pi = 3.14168
Time taken by function: 0.00194536seconds

-bash-4.2$ ./MonteCarloPiOmp
Number of threads: 16
number of “darts” thrown: 800000
Final Estimation of Pi = 3.14184
Time taken by function: 0.00292733seconds
```

Time

1 thread, 50000	2 thread, 100000	4 thread, 200000	8 thread, 400000	16 thread, 800000

1 thread, 50000	2 thread, 100000	4 thread, 200000	8 thread, 400000	16 thread, 800000
0.00150691s	0.00160127s	0.00165135s	0.00194536s	0.00292733s

Efficiency

1 thread	2 thread	4 thread	8 thread	16 thread
1	0.9411	0.9125	0.7746	0.5148

By increasing the darts number in accordance with the growth in the number of threads, where each doubling of the input size results in a doubling of the number of threads, I was able to identify the implementation's weak scaling property. I observed a similar speed prior to using 4 threads, demonstrating the technique's passable weak scalability. I did notice a speed drop at 4 threads, which might mean that this approach's upper limit for weak scaling is also 4.

Question 3

a.)

The fundamental difference between using OpenMP by itself and utilizing lightweight threading libraries is efficiency in extremely parallel situations. It is challenging to take use of huge parallelism while using OpenMP's pricy context switching and synchronization techniques. User-level threads, for example, allow context switching and synchronization activities that are more effective thanks to their lightweight nature.

b.)

I chose the Go threading library from the list of lightweight threading libraries presented and gave an example of how I would use it to parallelize a straightforward vector addition kernel (adding two vectors of single-precision floating point numbers together). The function must have time allotted to it in order to use goroutines in threading; otherwise, it will be returned right away. My strategy, where I utilize goroutines to generate the LWTs and Go syntax, would be as follows:

```
package main

import{
    "fmt"
    "time"
}

var N int = 100
var vector0 [N]float32
var vector1 [N]float32
var output [N]float32

func add(n int) {
    time.Sleep(time.Second)
    output[i] = vector0[i] + vector1[i]
}
```

```
func main() {
    for i: = 0, i < N; i++ {
        go add(i)
    }
}
```

c.)

Performance of BLAS-1 functions is assessed in this paper. The Sscal function is one of the scalar and vector operations available in the BLAS-1 subprograms. The elements in the vectors are split among threads in order to provide task-level parallelism in these subprograms.

Question 4

Please see the detail of codes in the folder q4.

Test results are as follow also in line with our expectations.

```
-bash-4.2$ ./semaphores
Number of threads: 1
number of “darts” thrown: 40000000
Create pthread success!
Thread 0 joined
pi_estimate: 3.14171
Time taken by function: 1.03743seconds

-bash-4.2$ ./semaphores
Number of threads: 2
number of “darts” thrown: 40000000
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
pi_estimate: 3.14139
Time taken by function: 0.52256seconds

-bash-4.2$ ./semaphores
Number of threads: 4
number of “darts” thrown: 40000000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
pi_estimate: 3.14202
Time taken by function: 0.267583seconds
```

```
-bash-4.2$ ./semaphores
Number of threads: 8
number of “darts” thrown: 40000000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
Thread 4 joined
Thread 5 joined
Thread 6 joined
Thread 7 joined
pi_estimate: 3.14241
Time taken by function: 0.136818seconds

-bash-4.2$ ./semaphores
Number of threads: 16
number of “darts” thrown: 40000000
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Create pthread success!
Thread 0 joined
Thread 1 joined
Thread 2 joined
Thread 3 joined
Thread 4 joined
Thread 5 joined
Thread 6 joined
Thread 7 joined
Thread 8 joined
Thread 9 joined
Thread 10 joined
Thread 11 joined
Thread 12 joined
Thread 13 joined
Thread 14 joined
Thread 15 joined
pi_estimate: 3.14324
Time taken by function: 0.09021seconds
```