# Lab report 2

Nhan Dao

September 30, 2020

## 1. MONTE CARLO'S ESTIMATION OF PI USING POSIX'S PTHREADS

### a. Briefly explain the difference between a process and a thread

A process can have multiple threads, these threads are considered within the process by the kernel. Being a part of the same process, threads shares instruction, global, and heap region of memory.

A multi-process program has memory space for each process, hence inter-process communication is slow because memory is not shared between them unlike threads. Context switching between process is more expensive compared to threads - which means when the OS/Kernel switch to executing a different processes, it has to reload the registers to regain the context for that execution, this does not happen with threads.

### b. Creating & Joining threads using POSIX

Listing 1 and listing 2 shows the creation and joining of POSIX threads.

```
1  for(i = 0; i < thread_count; i++) {
2    pthread_create(&thread_handles[i], NULL, &Thread_work, (void*) i);
3  }
```

Listing 1: Creating Pthreads (line 45-48 in lab2part1.c)

```
1  for(i = 0; i < thread_count; i++) {
2    pthread_join(thread_handles[i], NULL);
3  }
```

Listing 2: Joining Pthreads (line 49-51 in lab2part1.c)

### c. Generating random uniformly distributed dart tosses

Listing 3 shows the subroutine that generates uniformly distributed dart tosses between [-1, 1]. Each worker thread runs this subroutine, and counts the number of tosses that falls within the unit circle.

```
1  srand(seed);
2  for(toss = start; toss < finish; toss++) {
3    x = (double) rand() / RAND_MAX * 2 - 1;
4    y = (double) rand() / RAND_MAX * 2 - 1;
5    distance_squared = x*x + y*y;
6    if (distance_squared <= 1)
7      local_number_in_circle += 1;
8  }
```

Listing 3: Generating uniformly distributed dart tosses (line 82-89 in lab2part1.c)

### d. Initializing and destroying Mutex

```
1  pthread_mutex_init(&mutex, NULL);
```

Listing 4: Initializing Mutex (line 42 in lab2part1.c)

```
1  pthread_mutex_destroy(&mutex);
```

Listing 5: Destroying Mutex (line 56 in lab2part1.c)

### e. Computing global sum of the number of dart tosses within the unit circle

Listing 6 shows the code that compute the global sum of all 'successful' samples whilst avoiding any race condition using mutex.

```
1  pthread_mutex_lock(&mutex);
2  number_in_circle += local_number_in_circle;
3  pthread_mutex_unlock(&mutex);
```

Listing 6: Computing global sum (line 95-98 in lab2part1.c)

### f. Results

Estimating $\pi$ with Monte Carlo's method using $10^9$ tosses with 4 threads vs 1 thread. The linux Time command was used to time the execution.

The estimated values for $\pi$ was accurate to 3 decimal places. It took 2:13.82s to compute $\pi$ using $10^9$ samples with 4 threads, and only 25.790s to compute with 1 thread per fig. 1.1 and fig. 1.2 respectively.

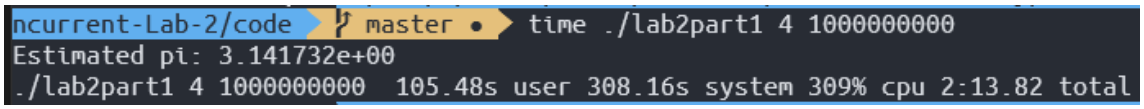


Figure 1.1: Terminal output of lab2part1.c program using $10^9$ tosses with 4 threads

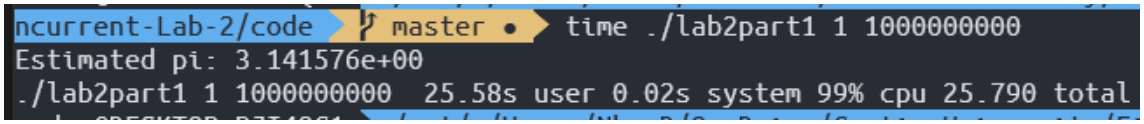


Figure 1.2: Terminal output of lab2part1.c program using $10^9$ tosses with 1 threads

## 2. Trapezoid rule with POSIX Pthreads

### a. Creating threads using POSIX

```
1     /* Start the threads. */
2     for (i = 0; i < thread_count; i++) {
3         pthread_create(&thread_handles[i], NULL, &Thread_work, (void*)
              i);
4     }
```

Listing 7: Creating POSIX Pthreads (line 72-75 in lab2part2.c)

### b. Joining threads using POSIX

```
1     /* Wait for threads to complete. */
2     for (i = 0; i < thread_count; i++) {
3         pthread_join(thread_handles[i], NULL);
4     }
```

Listing 8: Joining POSIX Pthreads (line 77-80 in lab2part2.c)

### c. If each thread is allowed to increment the total sum on the fly, does a race condition exist? If so, how does this affect the final result?

With syncronisation, a race condition will exist when multiple thread increment the total sum at the same time. This affects teh final result by making the global total sum to be less than the actual sum due increments that were unaccounted for because they were done at the same time.

### d. Mutex based critical section

```
1          total += my_int;
2
3        break;
```

Listing 9: Mutex based threads synchronisation (line 120-122 in lab2part2.c)

### e. Semaphore based critical section

```
1          sem_wait(&sem);
2          total += my_int;
3          sem_post(&sem);
```

Listing 10: Semaphores based threads synchronisation (line 120-122 in lab2part2.c)

### f. Busy-wait based critical section

```
1          while (flag != my_rank);
2          total += my_int;
3          flag = (flag + 1) % thread_count;
```

Listing 11: Busy-wait based threads synchronisation (line 115-117 in lab2part2.c)

### g. Results

The code was compile to compute the trapezoid rule for $a = 0$, $b = 1$, $n = 2^{30}$ for the function $f(x)$. 128 threads were used for each of the three methods (mutex, semaphores, busy-wait) to determine whic method is the fastest. The function $f(x)$ is hardwired to compute $x^2$, see listing 12.

```
1
2      return_val = x*x;
3      return return_val;
4  } /* f */
```

Listing 12: Hardwired f(x) for trapezoid rule (line 154-159 in lab2part2.c)

Linux echo was use to pipe in the user input so that it does not vary execution the execution time. Linux's time command was used to time the different synchronisation methods. The results are shown in figs. 2.1 to 2.3.

The result shows that semaphores is the fastest, then mutex and then busy-wait in terms of execution time. Busy wait is the least efficient because it consumes CPU cycles as the threads continously wait. Busy-wait enfores the order of access to the critical section, and waits for the system to schedule on the thread rank matching the flag variable to allow be allowed through - if it schedules on the wrong thread (a thread that is busy-waiting), it wastes CPU cycles in the

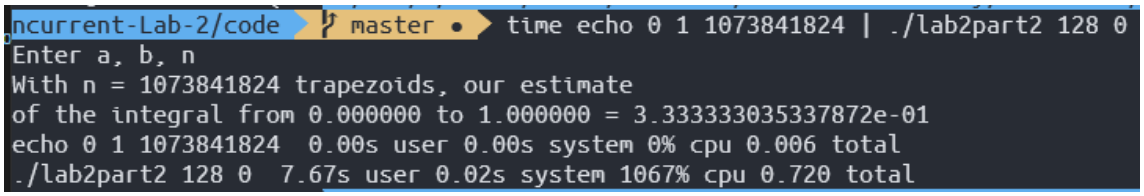


Figure 2.1: Terminal output of lab2part2.c program using 128 threads and mutex

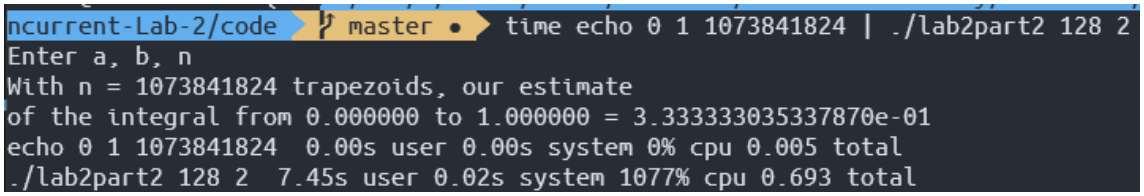


Figure 2.2: Terminal output of lab2part2.c program using 128 threads and semaphores

checking of the while condition until it deschedules and eventually until it picks the correct thread.

Mutex is middle between the three, only slower than semaphores by 40ms. The order the threads execute the critical section is random, which does not guarentee that locks are given in the order they are called.

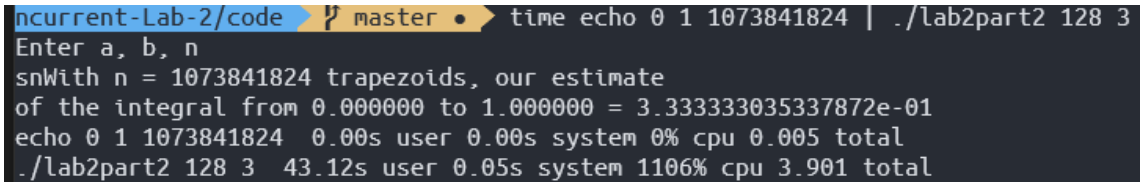


Figure 2.3: Terminal output of lab2part2.c program using 128 threads and busy-wait