LEBANESE AMERICAN UNIVERSITY
SCHOOL OF ARTS AND SCIENCES
DEPARTMENT OF COMPUTER SCIENCE & MATHEMATICS
BYBLOS

CSC 447: Parallel Programming for Multicore and Cluster Systems

Instructor: Haidar M. Harmanani
Spring 2017

Lab 2

π Computation
Due: February 22, 2017 (In Lab)

Part I [25 Points] – PI using Integration

Implement using `Pthreads` a C program that calculates π based on integration. The solution should use a block data distribution. Implement the problem while experimenting with various number of runs. Compute the error in every iteration and plot the error versus time using Excel. Start with 10 iterations and increase them in intervals of 1000 until 10,000,000.

Part II [50 Points] – PI using Monte Carlo Simulation

Redo the above Lab using Monte Carlo Simulation. Compute the error in every iteration and plot the error versus time using Excel. Start with 10 iterations and increase them in intervals of 1000 until 10,000,000. Please note that you cannot just simply use the same random number generator in various threads as this may not necessarily result with uniform random numbers due to overlap.

Part III [25 Points] – Hello World with a Controlled Order

Write a Hello World program that has threads say goodbye in a specific order: we want those with an even id to print first, and those with an odd id to print after all the evens have printed. Since we are not guaranteed that the threads will start in any given order, we must have the odd threads *wait* until all the even threads have printed. To make this happen, we will use condition variables.

To achieve the evens-print-first functionality, you will need to use the condition variable routines in the PThread library:

```
pthread_cond_wait(pthread_cond_t *condition, pthread_mutex_t *lock);
pthread_cond_broadcast(pthread_cond_t *condition);
```

The pthread_cond_wait routine will put the caller thread to sleep on the condition variable condition and release the mutex lock, guaranteeing that when the subsequent line is executed after the caller has woken up, the caller will hold lock. The pthread_cond_broadcast routine wakes up all threads that are sleeping on condition variable condition.

IMPORTANT: To call any of the above routines, the caller **must** first hold the lock that is associated with that condition variable. Failure to do this can lead to several problems. Check these global variables in your HelloWorld.c file:

```
pthread_mutex_t print_order_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t evens_done_CV = PTHREAD_COND_INITIALIZER;
int number_evens_finished = 0;
```

Add the logic necessary to have all the evens print before the odds. You will need to edit the sayHello function to contain calls to pthread_mutex_lock, pthread_mutex_unlock, pthread_cond_wait, and pthread_cond_broadcast. Do not use pthread_cond_signal.

Requirement

1. Your program should have different random numbers in each run.
2. The code should be signed-off with the TA to show it works.

Appendix

Listing I – Serial Pi Computation by Integration

```
static long num_steps=100000;
double step, pi;
void main()
{  int i;
   double x;
   step = 1.0/(double) num_steps;
   for (i=0; i< num_steps; i++){
      x = (i+0.5)*step;
      pi += 4.0/(1.0 + x*x);
   }
   pi *= step;
   printf(“Pi = %f\n”,pi);
}
```

Listing II – Serial Monte Carlo PI

```
int inTheCircle=0, numtrials=1000000;
double x, y;
for (i=0; i<numTrials; i++){
     x = rand();
     y = rand();
     if((x*x + y*y) < 1.0)
       inTheCircle++;
     }
pi = 4*inTheCircle/numTrials;
```

Listing III – Threaded Hello World

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Listing III – Using Condition Variables

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i<TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
    Check the value of count and signal waiting thread when condition is
    reached.  Note that this occurs while mutex is locked.
    */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      printf("inc_count(): thread %ld, count = %d  Threshold reached.\n",
             my_id, count);
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
        my_id, count);
    pthread_mutex_unlock(&count_mutex);

    /* Do some "work" so threads can alternate on mutex lock */
    sleep(1);
    }
  pthread_exit(NULL);
}

void *watch_count(void *t)
{
  long my_id = (long)t;
  printf("Starting watch_count(): thread %ld\n", my_id);

  /*
  Lock mutex and wait for signal.  Note that the pthread_cond_wait
  routine will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);
  while (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %ld Condition signal received.\n",
my_id);
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);

}
```