Kennesaw State University

Department of Computer Science

CS4491 Programming for HPC

Assignment 4 : Reinforce pthreads Concepts

Michael Hug

hmichae4@students.kennesaw.edu

October 1, 2014

Intial problem statement

4.2 : Suppose we toss darts randomly at a square dartboard, whose bullseye is at the origin, and whose sides are 2 feet in length. Suppose also that theres a circle inscribed in the square dartboard. The radius of the circle is 1 foot, and its area is π square feet. If the points that are hit by the darts are uniformly distributed (and we always hit the square), then the number of darts that hit inside the circle should approximately satisfy the equation

$$\frac{\text{number in circle}}{\text{total number of tosses}} = \frac{\pi}{4},$$

since the ratio of the area of the circle to the area of the square is $\pi/4$. We can use this formula to estimate the value of with a random number generator:

```
number_in_circle = 0;
for (toss = 0; toss < number_of_tosses; toss++) {
        x = random double between     1 and 1;
        y = random double between     1 and 1;
        distance_squared = x  x + y  y;
        if (distance_squared <= 1) number_in_circle++;
}
pi_estimate = 4 number_in_circle/((double) number_of_tosses);
```

This is called a Monte Carlo method, since it uses randomness (the dart tosses).

Write a Pthreads program that uses a Monte Carlo method to estimate . The main thread should read in the total number of tosses and print the estimate. You may want to use long long int s for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π.

*4.3 :Write a Pthreads program that implements the trapezoidal rule. Use a shared variable for the sum of all the threads computations. Use busy-waiting, mutexes, and semaphores to enforce mutual exclusion in the critical section. What advantages and disadvantages do you see with each approach? **note : Dr Gayler said to ignore the three different methods*

Response to problem 4.2

The Monte Carlo method for pi estimation was one that I had not seen before. It involved theoretically tossing darts at a dart board. The c source code to estimate pi using the Monte Carlo method follows.

```c
#include <time.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <limits.h>

long long int number_in_circle = 0;
long long int number_of_tosses_per_thread = -1;
int thread_count = -1;
pthread_mutex_t mut;

int get_max_threads()
{
  int max_string_size = 10; //including null char
  FILE *fp;
  char* ret;
  int max = -1;
  char str[max_string_size];
  fp = fopen("/proc/sys/kernel/threads-max","r");
  if(NULL == fp)
  {
    printf("error opening file thread max\n");
  }
  else
  {
    ret = fgets(str, max_string_size, fp);
    if (NULL == ret)
    {
      printf("file read error\n");
    }
    else
    {
      max = atoi(str);
    }
  }
  int retu = fclose(fp);
  if (0!=retu)
  {
    printf("file close error\n");
  }
  return max;
}
```

```
double grabRand()
{
  double div = RAND_MAX / 2;
  return -1 + (rand() / div);
}
void *monty(void* _)
{
  double distance_squared,x,y;
  long long int toss;
  long long int mycircleCount =0;
  for (toss = 0; toss < number_of_tosses_per_thread; toss++)
  {
    x = grabRand();
    y = grabRand();
    distance_squared = x*x + y*y;
    if (distance_squared <= 1)
    {
      mycircleCount++;
    }
  }
  pthread_mutex_lock(&mut);
  number_in_circle+=mycircleCount;
  pthread_mutex_unlock(&mut);
  return NULL;
}
int main(int argc, char *argv[])
{
int i, ret;
  int max_threads;
  int thread_count;
  double number_of_tosses;
  double pi_estimate;
  pthread_t* threads;
  srand(time(NULL)); //void return;

  if (3 != argc)
  {
    printf("I want 2 positional arguments : thread count &
          tosses per thread\n");
    return 1;
  }
  max_threads = get_max_threads();
  if(1 > max_threads)
  {
    printf("Are you on a posix complicant system?\n");
    return 2;
```

```
  }
  thread_count = atoi(argv[1]);
  if(1 > thread_count || thread_count>max_threads)
  {
    printf("supply an integer thread count
      between 1 and your system's max inclusivly\n");
    return 3;
  }
  number_of_tosses_per_thread = atoll(argv[2]);
  if(1 > number_of_tosses_per_thread ||
    number_of_tosses_per_thread>pow(10,52))
  {
    printf("Supply a toss_per_thread
      between 1 and a Sexdecillion inclusivly\n");
    return 4;
  }
  number_of_tosses_per_thread = number_of_tosses_per_thread/thread_count;
  if(0!=pthread_mutex_init(&mut, NULL))
  {
    printf("mutex creation fail\n");
    return 5;
  }
  threads = malloc(thread_count * sizeof(pthread_t));
  if (NULL == threads)
  {
    printf("pthread malloc failure\n");
    return 6;
  }
  for(i=0;i<thread_count;i++)
  {
    ret = pthread_create(&threads[i], NULL, monty, NULL);
    if(0!=ret)
    {
      printf("thread creation fail\n");
      return 7;
    }
  }
  for(i=0;i<thread_count;i++)
  {
    ret = pthread_join(threads[i],NULL);
    if(0!=ret)
    {
      printf("thread join fail\n");
      return 8;
    }
  }
```

```
  ret = pthread_mutex_destroy(&mut);
  if(0!=ret)
  {
    printf("mutex destroy fail\n");
    return 9;
  }
  free(threads); //void return
  //printf("%llu\n",number_in_circle);
  printf("Total tosses : %llu\n",thread_count*number_of_tosses_per_thread);
  number_of_tosses = thread_count*number_of_tosses_per_thread;
  pi_estimate = 4*number_in_circle/number_of_tosses;
  printf("Your estimation of PI : \t%.15f \n",pi_estimate);
  printf("math.h's macro for  PI: \t%.15f\n", M_PI);
  return 0;
}
```

Response to problem 4.3

I again decided to estimate pi, but this time using the trapezoidal area method. I found this method very quick and suprisingly accurate. The following contains the c source code to use the trapezoidal method.

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

double delta=1;
int thread_count = -1;
double area=0;
pthread_mutex_t mut;

struct my_thread_rank
{
  int rank;
};
int get_max_threads()
{
  int max_string_size = 10; //including null char
  FILE *fp;
  char* ret;
  int max = -1;
  char str[max_string_size];
  fp = fopen("/proc/sys/kernel/threads-max","r");
  if(NULL == fp)
```

```
  {
    printf("error opening file thread max\n");
  }
  else
  {
    ret = fgets(str, max_string_size, fp);
    if (NULL == ret)
    {
      printf("file read error\n");
    }
    else
    {
      max = atoi(str);
    }
  }
  int retur = fclose(fp);
  if (0!=retur)
  {
    printf("file close error\n");
  }
  return max;
}
double trap_area(double y1,double y2,double delta)
{
  return .5*(y1+y2)*delta;
}
double eval_function(double x)
{
  return sqrt(1-x*x);
}
void* my_thread(void * data)
{
  struct my_thread_rank *info = data;
  int my_rank = info ->rank;
  double lower_bound=my_rank * 1/(double)thread_count;
  double upper_bound=(1+my_rank) * 1/(double)thread_count;
  double y1;
  double y2;
  double my_area = 0.0;
  double i;
  y2 = eval_function(lower_bound);
  for(i=lower_bound+delta;i<=upper_bound;i+=delta)
  {
    y1=y2;
    y2=eval_function(i);
    my_area += trap_area(y1,y2,delta);
```

```
  }
  pthread_mutex_lock(&mut);
  area+=my_area;
  pthread_mutex_unlock(&mut);
  free(info);
  return 0;

}
int main(int argc, char *argv[])
{
  int i, ret;
  int max_threads;
  pthread_t* threads;

  if (3 != argc)
  {
    printf("I want 2 positional arguments :
    thread count & the delta of the trap(ex .00001)\n");
    return 1;
  }
  max_threads = get_max_threads();
  if(1 > max_threads)
  {
    printf("Are you on a posix complicant system?\n");
    return 2;
  }
  thread_count = atoi(argv[1]);
  if(1 > thread_count || thread_count>max_threads)
  {
    printf("Must supply an integer thread count
    between 1 and your system's max inclusivly\n");
    return 3;
  }
  delta = atof(argv[2]);
  if(0 > delta || 1 < delta)
  {
    printf("Supply a delta between 1 and zero\n");
    return 4;
  }
  if(0!=pthread_mutex_init(&mut, NULL))
  {
    printf("mutex creation fail\n");
    return 5;
  }
  threads = malloc(thread_count * sizeof(pthread_t));
  if (NULL == threads)
```

```
  {
    printf("pthread malloc failure\n");
    return 6;
  }
  for(i=0;i<thread_count;i++)
  {
    struct my_thread_rank *info = malloc(sizeof(struct my_thread_rank));
    if(NULL==info)
    {
      printf("strut malloc failure");
      return 7;
    }
    info->rank = i;
    ret = pthread_create(&threads[i], NULL, my_thread,info);
    if(0!=ret)
    {
      printf("thread creation fail\n");
      return 8;
    }
  }
  for(i=0;i<thread_count;i++)
  {
    ret = pthread_join(threads[i],NULL);
    if(0!=ret)
    {
      printf("thread join fail\n");
      return 9;
    }
  }
  ret = pthread_mutex_destroy(&mut);
  if(0!=ret)
  {
    printf("mutex destroy fail\n");
    return 10;
  }
  free(threads); //void return
  printf("Your trapezoidal estimation of PI: \t%.15f\n",area*4);
  printf("math.h's macro for PI: \t\t\t%.15f\n", M_PI);

  return 0;
}
```