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

In class we discussed parrelization of matrix operations. We were able to efficiently optimize matrix operations by using our knowledge of how C stores arrays. C is row major and will store rows contiguiously. We spawned threads who each were assigned their own row. The thread completed the mathematical operations on it's own row without examining any other rows. The thread then wrote to the last column(still in assigned row) the result of it's calculation. Each thread read it's data and never other thread's data. Each thread wrote to a location where no other thread would write to a location. There were no race conditions because we cleverly assigned read and write locations to each thread.

If we consider that the number of rows and columns is not evenly divisible by the number of threads, we will have to make some changes to our code. The ideal situation would be that the number of threads is exactly a multiple of the number of rows. The book has been asuming that all mathematical operations are treated equally. Treating all operations the same leaves us with the knowledge that when two threads start to do math on two rows, they will end at the same time. I think it would be best to think about this recursivly. The base case is when there are no rows left: exit the recursion. If there are rows left and more rows than thread, each thread will get it's own row. If there are rows left, but less rows than threads, we would chunk the rows and give each threads it's own portion of the remaining rows. When we do this, we have to be careful about the race condition that exists when the threads are finished. They will all finish at the same time, so we will have a small botleneck on writing to the last column of the row(the result). The threads have to take turns adding their result to the final column.

```
#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;
  }
  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

The man page for gcc did not contain information about the -O arguemnt. The info page had sufficient information.

Running the unoptomized unaltered code :
Both single and multithreaded estimations were identical except for the least significant digit. The threaded execution was an only marginally faster.

Running the optomized unaltered code:
This overheated my box!!! I let it run and went to grab a soda to find my

machine shutting down. Busy-wait and deadlock is not a good combination

altering lines 40 and 41 from :

```
int flag;
double sum;
```

to :

```
int volatile flag;
double volatile sum;
```

Running the unoptomized altered code :
Same results as the unoptomized unaltered code - no suprise
Running the optomized altered code :
Now we get the same results as the unoptomized unaltered code.The grand total of all optomizations in the code were likely only the flag and sum variable modification orderings.

In conclusion, we accomplished the same goal several different ways. We told the compiler not to optomize our code from within the source code and also as an agrument to the compiler. We also found that busy-waits should have a threshold or something to prevent your system from crashing. And also, I should use the advice that I give grandma, and not run code I find on the internet. I like the idea of the volatile, that the profesional programer can cherry pick the variables that should not be optomized, rather than having either a yes or no on compiler optomization.

Response to problem 4.14

```
see code in .c file
```

The unexpected part was that I was able to get done quicker, even with the overhead of translating the 2d matrix to 1d!