# CPEN 333: System Software Engineering

Lectures and course material for CPEN 333: System Sofware Engineering, Department of Electrical and Computer Engineering, University of British Columbia.

« Lab 1 - Introduction to C++ Lab 3 - Testing and Mutexes »

# Lab 2 – Multi-Threading

In this lab, we will get some practice with creating threads and applying concurrency effectively in *modern C++* (i.e. C++11 or higher). We will look at two cases: speeding up the `quicksort` algorithm by using multiple threads; and monte carlo simulation, a very powerful scientific technique for estimating quantities when they are difficult to compute analytically.

To help get you started, some of the code is posted on GitHub here (https://github.com/cpen333/lab2).

Feel free to discuss approaches and solutions with your classmates, but labs are to be completed individually. Each student is expected to be able to answer questions about the content, describe their work, and reproduce their code (or parts thereof).

**Contents:**



## Part 1: QuickSort

The QuickSort algorithm is one of the most efficient single-threaded sorting algorithms out there, with an expected computational complexity of $O(n \log n)$. We will try to speed this algorithm up slightly by introducing concurrency.

Either create a new file called `quicksort.cpp`, or download the template provided on GitHub. We will create two versions of the QuickSort algorithm for sorting a set of random integers:

- `quicksort` : regular single-threaded version
- `parallel_quicksort` : a version which sorts the left and right halves (after partitioning) in separate threads to be executed concurrently

We will then measure the computation time of both, and compute the *speed-up factor* to see how much we have gained.

The basic layout of your code should look as follows:

```
#include <iostream>
#include <thread>
#include <vector>

// partitions elements low through high (inclusive)
// around a pivot and returns the pivot index
int partition(std::vector<int>& data, int low, int high) {
  // your code here
}

// sorts elements low through high (inclusive) using a single thread
void quicksort(std::vector<int>& data, int low, int high) {
  // your code here
}

// sorts elements low through high (inclusive) using multiple threads
void parallel_quicksort(std::vector<int>& data, int low, int high) {
  // your code here
}

int main() {

  // create two copies of random data
  const int VECTOR_SIZE = 1000000;
  std::vector<int> v1(VECTOR_SIZE, 0);
  // fill with random integers
  for (int i=0; i<VECTOR_SIZE; ++i) {
    v1[i] = rand();
  }
  std::vector<int> v2 = v1;  // copy all contents

  // sort v1 using sequential algorithm
  quicksort(v1, 0, v1.size()-1);
```

```
  // sort v2 using parallel algorithm
  parallel_quicksort(v2, 0, v2.size()-1);

  return 0;
}
```

In the above, rather than working with an array, we are using an `std::vector<int>`. Vectors are very much like arrays, except they are *objects*, they store the number of elements, and they can be resized. Just like with an array, we can access individual elements of a vector using the `[]` operator, e.g. `int x = v[0]`. For more details on the vector class, see the documentation. If you need a reminder of the implementation details of the QuickSort algorithm, feel free to have a look at the page on Wikipedia.

## Testing your algorithm

Write a method to check that the outputs of both of your implementations are indeed sorted. It's good to get into the habit of writing tests as you go along.

## Measuring computation time

To measure computation time, we will use more modern features of C++, specifically the `<chrono>` library. The `<chrono>` library contains a set of classes and functions for measuring times in different units, and for computing and converting durations. The classes are heavily templated, which can make it a little confusing to use for beginners, but for the most part you can just copy and paste a select few lines of code.

To get the current time at the highest possible clock resolution, use

```
auto time = std::chrono::high_resolution_clock::now();
```

There are three possible clocks which you could choose from: `steady_clock`, `system_clock`, and `high_resolution_clock`. We are using the high resolution one in the hope of measuring our computation time with the most accuracy (though technically we are dealing with precision, not accuracy). We use the `auto` keyword mainly for convenience. The actual type of the returned value is `std::chrono::time_point<std::chrono::high_resolution_clock>`, which is a mouthful. The `auto` keyword auto-detects this type so we don't have to know, care, or write it out.

The difference between two time points creates a `std::chrono::duration<Rep,Period>`, where `Rep` and `Period` are more templated types. When taking the difference between two high resolution clock timepoints, this expands to `std::chrono::duration<long long int, std::ratio<1, 1000000000>>`. But again, we shouldn't really care since we will be using the `auto` keyword:

```
auto t1 = std::chrono::high_resolution_clock::now();
// do some stuff...
auto t2 = std::chrono::high_resolution_clock::now();
auto duration = t2-t1;
```

We can force the duration to be in some known quantity, like milliseconds or microseconds, by applying a duration cast:

```
auto duration_ms = std::chrono::duration_cast<std::chrono::milliseconds>(duration);
long ms = duration_ms.count();
std::cout << "The operation took " << ms << " ms" << std::endl;
```

We extracted the actual number of milliseconds above using the `.count()` method. See the documentation for durations for more details. And that's it! With these few lines of code, and a duration cast to a desired time resolution, you should be able to measure computation times down to the nanosecond.

### Speed-up Factor

Recall that the speed-up factor is computed by

$$S_p = T_1/T_p$$

where $p$ is the number of processors or cores, and $T_p$ is the time taken for the code to run on $p$ processors.

### Questions

1. How much speed-up were you expecting based on the number of processors/cores on your machine?
2. Did you achieve the speed-up you expected? If not, what do you think might be interfering with this?
3. In your parallel implementation, how many threads were created (approximately) when sorting one million elements? Did you create two new threads in each call to `parallel_quicksort` ? Is there a way to reduce the number of new threads by about half without sacrificing parallelism?

## Part 2: Monte Carlo Methods

The Monte-Carlo method is a general purpose technique for estimating quantities traditionally difficult to compute analytically. It involves estimating a quantity using random sampling over and over and over and over again, and averaging the results to come up with a `best guess' of

the value. In general, each sample is independent of all other samples, making this ideal for concurrent implementations.

For example, consider a game of monopoly. What is the expected number of times of being sent to jail in a game with four players and 100 turns? Not only do you have to consider the random rolls of the die, but also the community chest and chance cards. Rather than try to go through all possible games with every possible dice roll for every possible turn, we can *simulate* a whole bunch of games – say 100,000 – count how many times each player ended up in jail, and average them. This should give us a decent estimate of the true value.

In this part of the lab, we are going to use the Monte-Carlo method to estimate the value of $\pi$, and to integrate some functions.

## Generating random numbers

In Part 1, we generated a list of random integers using the C `rand()` function. This is *usually* fine, but in C++ there are new methods in the `<random>` library, so we are going to try them out.

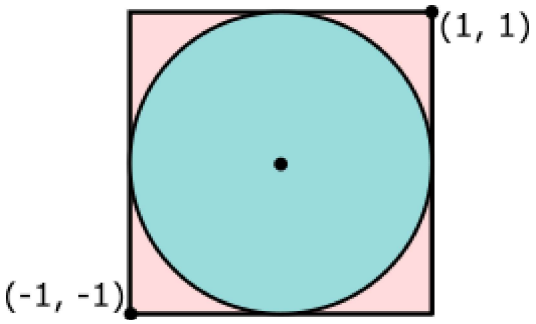
```
#include <random>

int main() {
  // specify the engine and distribution.
  std::default_random_engine rnd;
  std::uniform_real_distribution<double> dist(-1.0, 1.0);

  // generate 100 random doubles
  for (int i=0; i<100; ++i) {
    std::cout << dist(rnd) << std::endl;
  }

  return 0;
}
```

We need to create two objects: a random number *engine*, and a *distribution* from which to draw the random numbers. The output values are obtained by evaluating the distribution at the random engine. In the above, we use the default random engine, and create a uniform distribution of real numbers in the range of [-1, 1]. The template parameter in the distribution tells it to give us values of type `double`. Every time this code runs, we will get the exact same random sequence, which is great for debugging, but bad for business. To generate different numbers each time, the random engine constructor can accept a seed value (similar to the C `srand(int)` function). It's quite common to use the current system time:

```
// seed the random engine with the current time
std::default_random_engine rnd(
   std::chrono::system_clock::now().time_since_epoch().count());
```

## Estimating PI

In order to estimate the value of PI using random sampling, we need to define a random function where we expect the answer to be PI. The most common approach is to consider a circle with radius 1 inside a square with side-lengths 2.




The area of the unit circle is simply $\pi$. Surround this with a bounding square. The square has side-lenghts 2 and an area of 4. If we were to randomly generate samples inside the square region [-1,1]x[-1,1], we would expect that the points would fall inside or on the unit circle with a probability of $p = \pi/4$, the ratio of the two areas. If a random sample falls within the circle, we call that a *hit*. If it falls outside the circle, we call that a *miss*. All we need to do now is generate samples within the square region, count the fraction of hits, and multiply by four to recover the estimate of $\pi$.

Create a new source file called `pi.cpp`, or download the template provided on GitHub. First, create a single-threaded method that estimates the value of PI. The layout of the file should look something like the following:

```
#include <thread>
#include <iostream>
#include <random>

double estimate_pi(int nsamples) {
  // your code here
}

int main() {

  double pi = estimate_pi(1000);
  std::cout << "My estimate of PI is: " << pi << std::endl;

  return 0;
}
```

How accurate is your PI estimate with only 1000 random samples? How many samples do you need to be accurate to 3 decimal places?

## Speeding up the computation

One way to try to speed things up is to generate and check each random sample in a separate thread. Each sample is independent of each other, so we should be able to perform our sampling in parallel. Because we haven't talked about synchronization yet, we have to be a bit careful with counting the number of *hits* between threads. The easiest way to handle this at the moment is to create a vector or array of booleans, and have each thread responsible for filling in one value.

Create a naive concurrent version of `estimate_pi` based on the following:

```
// generates a random sample and sets the value of `inside`
// to true if within the unit circle
void pi_sampler(std::vector<bool>& hits, int idx) {

  // single instance of random engine and distribution
  static std::default_random_engine rnd;
  static std::uniform_real_distribution<double> dist(-1.0, 1.0);

  // YOUR CODE HERE

}

// naively uses multithreading to try to speed up computations
double estimate_pi_multithread_naive(int nsamples) {
  // stores whether each sample falls within circle
  std::vector<bool> hits(nsamples, false);

  // create and store all threads
  std::vector<std::thread> threads;
  for (int i=0; i<nsamples; ++i) {
    threads.push_back(std::thread(pi_sampler, std::ref(hits), i));
  }

  // wait for all threads to complete
  for (int i=0; i<nsamples; ++i) {
    threads[i].join();
  }

  // estimate our value of PI
  double pi = 0;
  for (int i=0; i<nsamples; ++i) {
    if (hits[i]) {
      pi = pi + 1;
    }
  }
  pi = pi / nsamples*4;

  return pi;
}
```

Fill in the function that does the random sampling and circle test. **WARNING:** *use a very small number of samples, such as 1000. If you create too many threads at once you will get a system error.* Is this method any faster than the single-threaded version? Why not?

We want to limit the number of threads created in order to reduce overhead, but also try to maximize concurrency. How many threads should we allow? You can determine the number of cores on your machine using the static function

```
std::thread::hardware_concurrency()
```

A better way of splitting up the work is to create a small number of threads, and let each thread handle a certain number of samples. Create a new function that performs the random sampling using the same number of threads as there are cores on your machine. To avoid data sharing between threads, keep $N$ separate hit counts and add them together once all threads are done.

```
// count number of hits using nsamples, populates hits[idx]
void pi_hits(std::vector<int>& hits, int idx, int nsamples) {

  // single instance of random engine and distribution
  static std::default_random_engine rnd;
  static std::uniform_real_distribution<double> dist(-1.0, 1.0);

  // YOUR CODE HERE
}

// divides work among threads intelligently
double estimate_pi_multithread(int nsamples) {

  // number of available cores
  int nthreads = std::thread::hardware_concurrency();

  // hit counts
  std::vector<int> hits(nthreads, 0);

  // create and store threads
  std::vector<std::thread> threads;
  int msamples = 0; // samples accounted for
  for (int i=0; i<nthreads-1; ++i) {
    threads.push_back(
      std::thread(pi_hits, std::ref(hits), i, nsamples/nthreads));
    msamples += nsamples/nthreads;
  }
  // remaining samples
  threads.push_back(
    std::thread(pi_hits, std::ref(hits), nthreads-1, nsamples-msamples));

  // wait for threads to finish
  for (int i=0; i<nthreads; ++i) {
    threads[i].join();
```

```
  }

  // estimate pi
  double pi = 0;
  for (int i=0; i<nthreads; ++i) {
    pi += hits[i];
  }
  pi = pi/nsamples*4;

  return pi;
}
```

Now are you seeing a speed-up? With so few threads, you can increase the number of samples again until you get an accuracy of 3 decimal places. For how many samples does it become worth it to use multiple threads?

### Questions

1. What have you learned in terms of splitting up work between threads?
2. What implications does this have when designing concurrent code?
3. How many samples do you *think* you will need for an accuracy of 7 decimal places?

## Monte-Carlo integration

One very common application of Monte-Carlo methods is numerical integration. Consider some function $f(x)$ defined over some volume $V$. How can we pose the integral of this function in a random sampling framework?

The most basic assumption we can make when approximating an integral is that the function is constant over the entire volume. In such a case, the integral becomes

$$\int_V f(x)\,dx \approx \text{vol}(V) f(x)$$

Therefore, if we randomly select an $\hat{x}$ from somewhere within the volume, we have a naive estimate of the integral: $\text{vol}(V) f(\hat{x})$. If we take a different sample, we would get a different rough estimate of the integral. Thus, to converge to the true solution, all we need to do is take a ton of samples and average all our integral estimates together:

$$\int_V f(x)\,dx \approx \frac{1}{N} \sum_{i=1}^{N} \text{vol}(V) f(\hat{x}_i)$$

### Centre of mass computation

In this final problem, we are going to use Monte-Carlo integration to estimate the centre of mass of a unit sphere (radius 1, centred at the origin) with a spatially varying density, $\rho$. Recall that the centre of mass is defined as $\mathbf{c} = (c_x, c_y, c_z)$ with

$$c_x = \int_V x\,\rho(\mathbf{x})d\mathbf{x} \;/\; \int_V \rho(\mathbf{x})d\mathbf{x},$$

$$c_y = \int_V y\,\rho(\mathbf{x})d\mathbf{x} \;/\; \int_V \rho(\mathbf{x})d\mathbf{x},$$

$$c_z = \int_V z\,\rho(\mathbf{x})d\mathbf{x} \;/\; \int_V \rho(\mathbf{x})d\mathbf{x}.$$

To apply Monte-Carlo integration within the sphere, we have three options:

1. generate random samples directly within the sphere using spherical coordinates
2. generate random samples within a bounding cube, and if they happen to fall outside the sphere, reject them and try again
3. extend our density function to a bounding cube such that

$$\rho_{\text{cube}} = \begin{cases} \rho & \text{if } \mathbf{x} \in V \\ 0 & \text{otherwise} \end{cases}$$

   We could then integrate the new density, $\rho_{\text{cube}}$ over the bounding cube, which is a bit easier.

Write a program to compute the centre of mass of a unit sphere with the following densities:

- Density 1: $\rho(\mathbf{x}) = \exp^{-\|\mathbf{x}\|^2}$
- Density 2: $\rho(\mathbf{x}) = \text{abs}(x + y + z)$
- Density 3: $\rho(\mathbf{x}) = (x-1)^2 + (y-2)^2 + (z-3)^2$

You may find some methods defined in `<cmath>` useful.

## Questions

1. How did you divide the problem to take advantage of concurrency?
2. How did you implement the density function? Did you make use of code re-use?
3. Did you test your program to make sure it was computing the integrals correctly? If not, how might you do that?

« Lab 1 - Introduction to C++ Lab 3 - Testing and Mutexes »

**CPEN 333: System Software Engineering is maintained by C. Antonio Sánchez.**