159.735 Assignment 1 Notes

Sequential implementation

The following linear congruential generator is a suitable random number generator:

$$n_{i+1} = (an_i + c) \mod m$$

where $a = 1664525$, $m = 2^{32}$, and $c = 1013904223$.

Generating a sequence of random numbers on just one processor is very straightforward, as in the following pseudo-code:

```
n_prev = 12345;  // Seed can be anything you like
for i=0; i < N; ++i
    n_next = (a * n_prev + c) % m     // Next random integer in the sequence

    // ...
    // do something with the random number
    // ...

    n_prev = n_next   // Set up for the next random number
```

- N is the number of random samples. This should not exceed m.
- In a C/C++ implementation, it is suggested to use 8 byte `long` type integers and 8 byte floating point values, ie `double` type

As a side note: you only have to use one call of the random number generator to sample a single set of (x, y) coordinates. Consider how you would do this. You do NOT have to use the generator twice, i.e. once for x and once again for y— in fact, you do not want to do this!

How could we parallelize this procedure? Obvious way is to partition the range of N according to the number of processes, p, that will be used. Each process will work on N/p steps in the computation

How do we parallelize the random number generation?

Possible methods

Implement the generator separately on each process. No good because:

- if the same seed is used, each process will get an identical sequence

- if a different seed is used, each process will get its random numbers from a different "universe". Each process must sample the sequence generated by the same seed

Employ a master/slave approach. The master will generate the sequence and send each one to its appropriate slave process. This is the "naive" method referred to in the assignment handout

- way too excessive communication overhead!

Leapfrog method

A linear congruential generator of the form, $x_{i+1} = (ax_i + c) \mod m$ has the following property

$$x_{i+k} = (Ax_i + C) \mod m$$

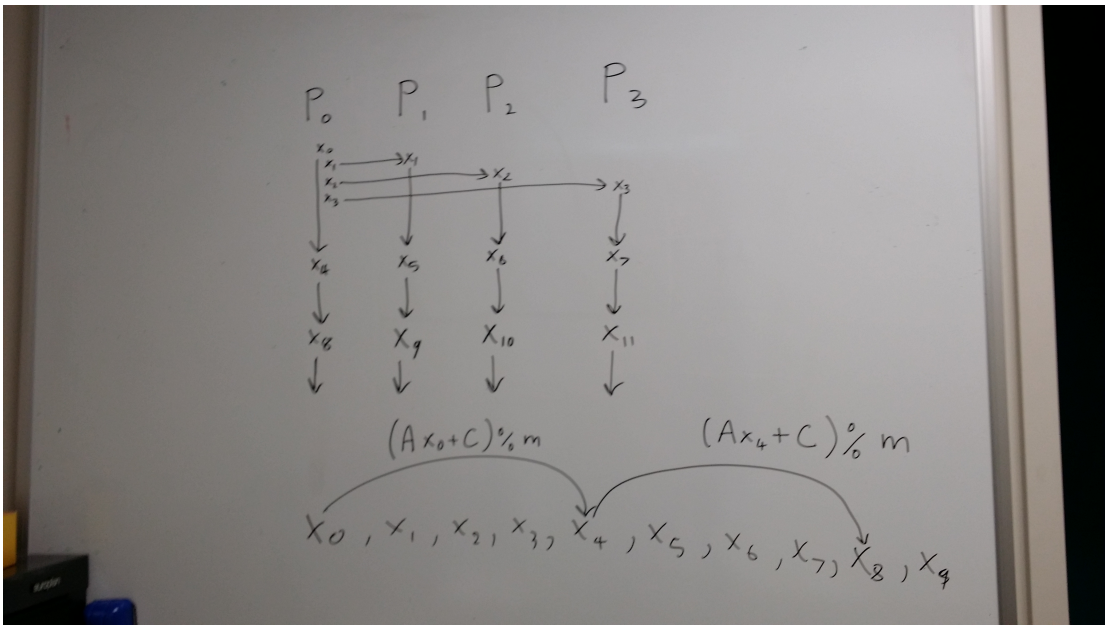
where k is the "jump constant" and

$$A = a^k \mod m$$
$$C = c(a^{k-1} + a^{k-2} + \cdots + a^1 + a^0) \mod m$$

This means that for a given random number x_i, one can generate the number x_{i+k} in the sequence by "leaping" over the numbers in between without having to compute them.

If the jump constant is set to the number of processors, then we can use this property to share the same universe of random numbers amongst the processors with minimal communication overhead. The master process generates the first k random numbers and sends each one to its corresponding slave processes. Each process can then generate its own set of random numbers by leaping over the ones used in the other processes.

For example, consider how it would be done for four processors as in the following whiteboard snapshot:



The constants a, c, and m are fixed throughout. The constants A and C depend upon the number of parallel processes you will use. A table of jump constants is provided for you on Stream. However, if you would like to calculate them yourself, take note of the following:

Care must be taken in calculating the constants A and C. We are working with integers and we do not want intermediate values to overflow. Also, the numbers must be computed exactly.

In particular, for the modulo exponentiation operation, using the C library `pow()` function is absolutely the wrong way to do it, i.e. do **NOT** do anything like this:

```
int a = 1664525;
int m = pow(2, 32);
int k = 4;
int A = pow(a, k) % m;
```

Care should be taken in calculating A and C. To compute $y = x^n \mod p$, one can use a simple algorithm as follows:

```
initialize:
    y = 1
for (e=0;e<n;++e):
    y = (y * x) mod p
output y
```

Computation of C is a little trickier, but you can make use of the following useful properties of modulo arithmetic.

Distributivity:

$$(a + b) \mod m = [(a \mod m) + (b \mod m)] \mod m$$

Associativity:

$$(ab) \mod m = [a(b \mod m)] \mod m$$