Parallel Additive Lagged Modular Fibonacci Random Number Generators (ALFG's)

(Or why bad code documentation is worse than none at all)

Lewis Hall
Jason Main

Misleading documentation of glibc random():

- From random(3) man page “it uses a non-linear additive feedback random number generator”
- Documentation in actual code discusses “special state info interface”
- What glibc actually uses:
- Additive Lagged Modular Fibonacci Random Number Generator (RNG)

Fibonacci Generators

Normal Fibonacci Sequence

$$f_n = \begin{cases} n & (\mathrm{n} < 2) \\ f_{n-2} + f_{n-1} & (\mathrm{otherwise}) \end{cases}$$

$$0 \;\; 1 \;\; 1 \;\; 2 \;\; 3 \;\; 5 \;\; 8 \;\; 13 \;\; 21 \;\; 34 \;\; 55 \;\; 89 \;\; 144 \;\; 233 \;\; 377 \ldots$$

Lagged Modular Fibonacci Sequence

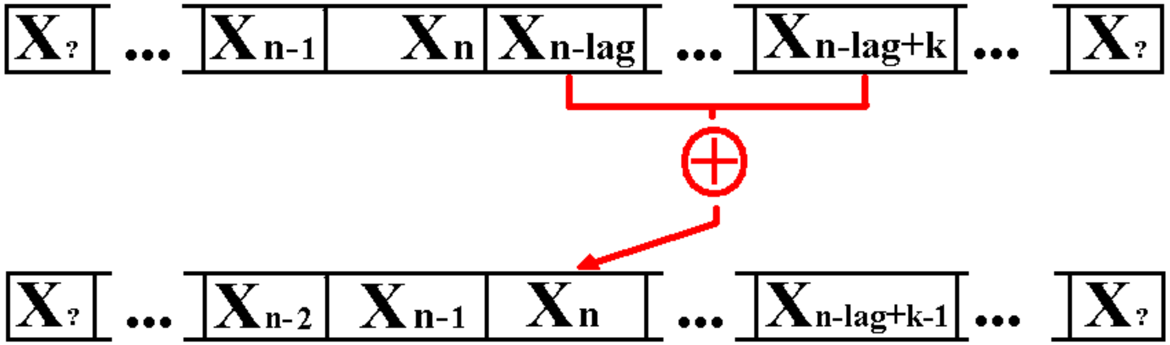
$$f_n = \left(f_{n-lag} + f_{n-(lag-k)} \right) \bmod \mathrm{P}$$

P is typically the max machine word

lag is 31 as default for glibc and is limited to 63

Notes about glibc implementation

Kept in a circular buffer with the last lag



Essential ideas for parallelization

- Each processor calculates a contiguous block of random numbers from the original sequence
- Processor i calculates the state as if it had just been generated $X_{\lfloor n/p \rfloor \cdot i}$
- Formulate a group of algorithm iterations into a matrix. Then use linear algebra to find the initial state quickly. *(lag many iterations are used)*

How to parallelize glibc’s implementation?

- Each process needs to be updated with constantly changing state information from other processes.
- This leads to excessive communication time.

- Disjoint subsequence paradigm.

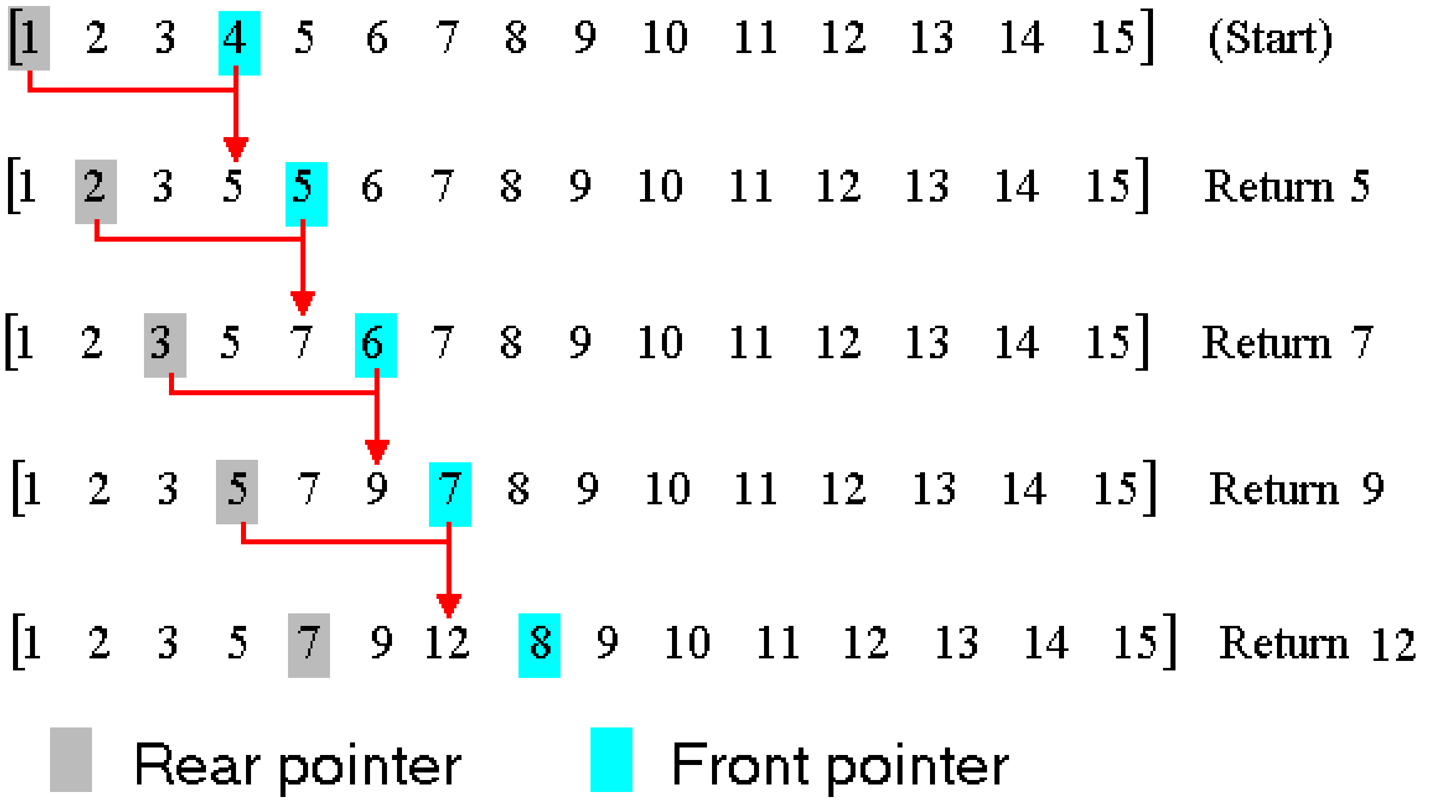
Independent State Calculation

- Each process knows how many random numbers total need generated and how many processes will participate, and thus can calculate the proper state to begin generating random numbers.

- This can be accomplished using a little clever linear algebra, polynomial calculation knowledge, and recursion.

- Eliminates the need for communicating state information, so each process can quickly compute it’s own subset of random numbers.

A few iterations of the random()...



[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15] (Start)

[1 2 3 5 5 6 7 8 9 10 11 12 13 14 15] Return 5

[1 2 3 5 7 6 7 8 9 10 11 12 13 14 15] Return 7

[1 2 3 5 7 9 7 8 9 10 11 12 13 14 15] Return 9

[1 2 3 5 7 9 12 8 9 10 11 12 13 14 15] Return 12

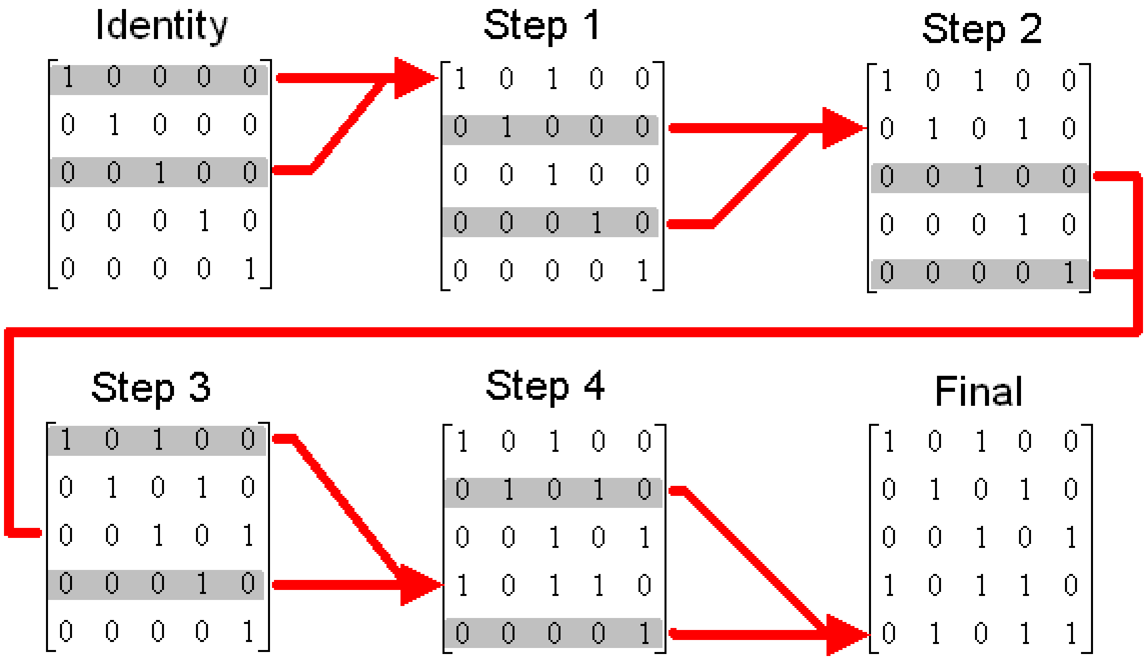
▨ Rear pointer ▨ Front pointer

Core Idea: Formulate matrix **A** so that **A**x = x* where x is the state vector and x* is the vector after lag many iterations. We will then manipulate **A**.

Formulate matrix **A** so that **A**x = x* where x is the state vector and x* is the vector after lag many iterations.

Construction of the Matrix

Algorithm forms ***A*** *as a square matrix of size lag*



Identity

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 1

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 2

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 3

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 4

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Final

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

For this example:

lag is 5

k is 2

The oldest member of the state is at index 0

After this two fast linear algebra algorithms exist for calculating $\mathbf{A}^n$ for large n

Successive doubling: Recursively square **A** and use the binary expansion of n to choose which powers of **A** to combine *(order* $\boldsymbol{O(lag^3 \log_2 n)}$ *)*

Diagonalization: Find decomposition $\mathbf{A} = \mathbf{X}^{-1}\mathbf{D}\mathbf{X}$. Then $A^n = \mathbf{X}^{-1}\mathbf{D}\mathbf{X}$. *(order* $\boldsymbol{O(lag^2)}$ *as the decomposition can be precomputed. Probably wont work here.)*

‘Srinivas Aluru’ Style Matrices

(Combine a permutation of the state vector with a single iteration of the LF algorithm)

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_n \\ X_{n-1} \\ X_{n-2} \\ X_{n-3} \\ X_{n-4} \\ X_{n-5} \\ X_{n-6} \end{bmatrix} = \begin{bmatrix} X_{n+1} \\ X_n \\ X_{n-1} \\ X_{n-2} \\ X_{n-3} \\ X_{n-4} \\ X_{n-5} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_n \\ X_{n-1} \\ X_{n-2} \\ X_{n-3} \\ X_{n-4} \\ X_{n-5} \\ X_{n-6} \end{bmatrix} = \begin{bmatrix} X_{n+7} \\ X_{n+6} \\ X_{n+5} \\ X_{n+4} \\ X_{n+3} \\ X_{n+2} \\ X_{n+1} \end{bmatrix}$$

‘Our’ Style Matrices

(Iterate lag many iterations so that no permutation has to occur)

Matrix Powers

Say we have the binary expansion of B:

$$B = b_{31} b_{30} b_{29} b_{28} \cdots b_2 b_1 b_0$$

For illustration say that $\mathrm{b_x} = 0$ for odd x

$$B = b_{31} 2^{31} + b_{30} 2^{30} + b_2 2^2 \cdots b_1 2^1 + b_0 2^0$$

$$A^B = A^{b_{31} 2^{31} + b_{30} 2^{30} + b_2 2^2 \cdots b_1 2^1 + b_0 2^0}$$

$$A^B = A^{b_{31} 2^{31}} A^{b_{30} 2^{30}} A^{b_{31} 2^{29}} \cdots A^{b_1 2^1} A^{b_0 2^0}$$

Now since $\mathrm{b_x}$ is 0 for even x we have:

$$A^B = A^{2^{31}} \cdots A^{2^9} A^{2^7} A^{2^5} A^{2^3} A^{2^1}$$

$$A^B = A^{2147483648} \cdots A^{512} A^{128} A^{32} A^{8} A^{2}$$

$$A$$

$$A^2 = AA$$

$$A^4 = A^2 A^2$$

$$A^8 = A^4 A^4$$

$$A^{16} = A^8 A^8$$

$$A^{32} = A^{16} A^{16}$$

$$A^{64} = A^{32} A^{32}$$

$$A^{128} = A^{64} A^{64}$$

$$A^{256} = A^{128} A^{128}$$

$$\vdots$$

Basic Algorithm for ALFG

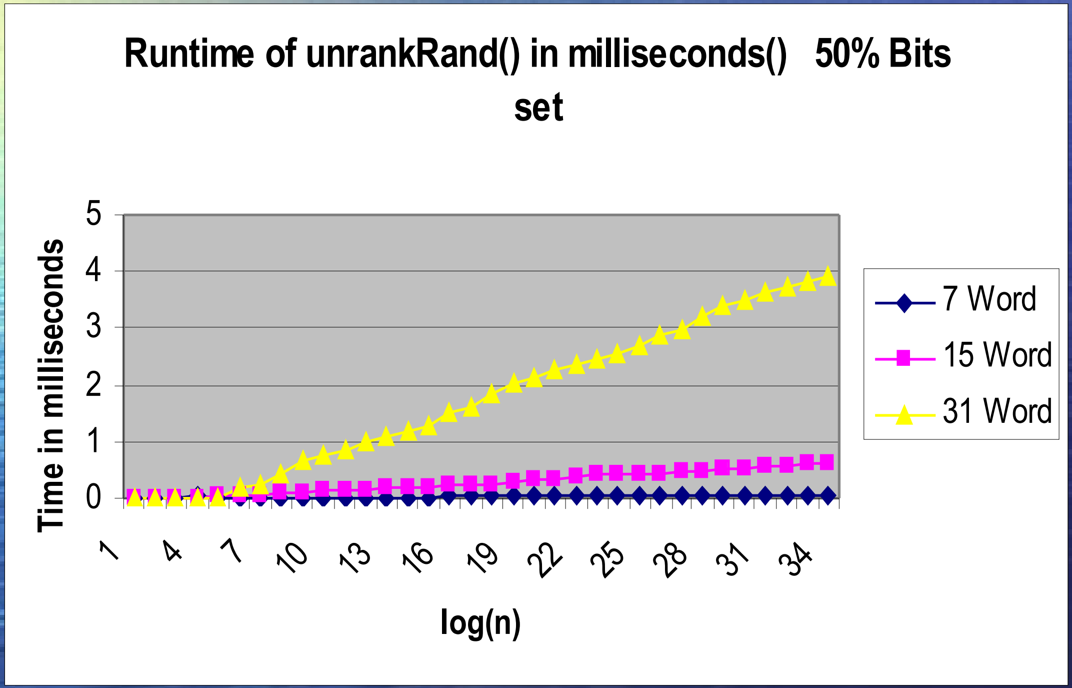
```
UnrankRandom ( stride )
    while ( stride % polySize )
      random()
      stride--

    B = stride / polySize
    A = createPermutationMatrix()
    P = fastpower(A,B)
    x = Px
END
```

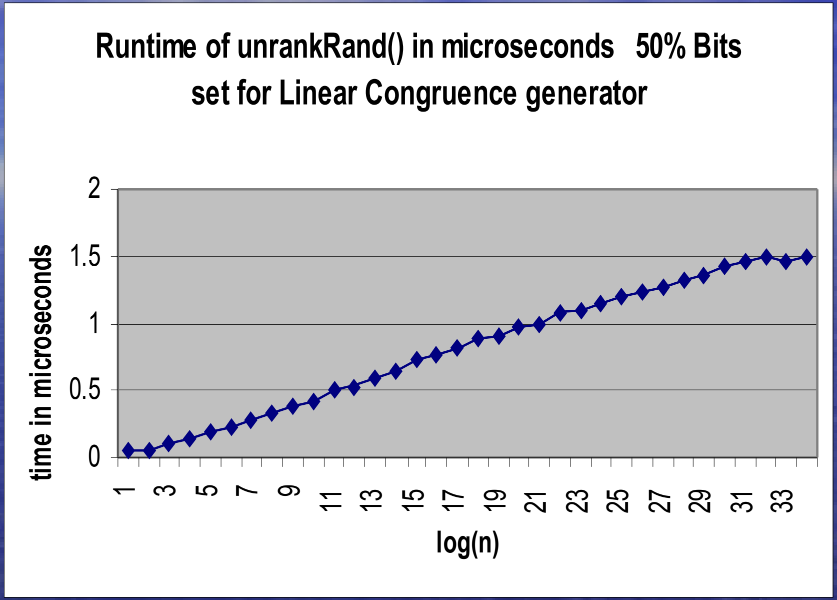
Takes how far you want to move ahead

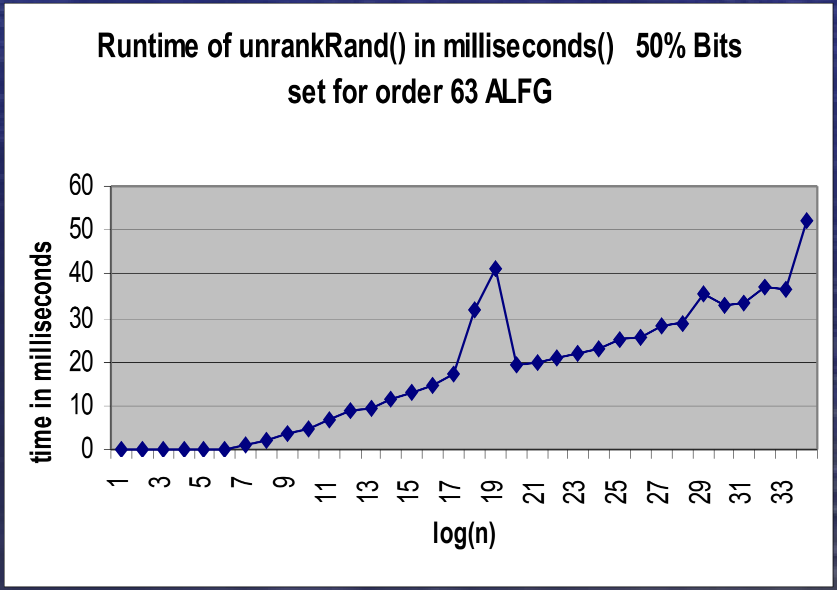
There is really some code here to handle
(stride < polySize)

B is how many permutations from our
matrix we need given that each multiple
takes us ahead by polySize states

Timing Results







Linear congruential state calculation

- In order to calculate the state after k random numbers have been generated, the equations becomes:

$$X_{n+k} = a^k X_n + c(a^{k-1} + a^{k-2} + \cdots + a + 1) \mod m$$

Works great for leap frog but what do you do for block parallelization where k can be a billion?

$$(a^k + a^{k-1} \ldots + 1) = \begin{cases} 1 & k = 0 \\ (a+1) & k = 1 \\ (a^{k/2} + a^{k/2-1} \ldots + 1)(a^{k/2+1} + 1) & k \text{ is odd} \\ (a^{k/2} + a^{k/2-1} \ldots + 1)(a^{k/2} + 1) - a^{k/2} & k \text{ is even} \end{cases}$$

Additionally we allow for negative numbers to be passed to unrankRand() which requires modular division. That’s it really

Where do we go next...

- Matrix Multiplies can be slow (especially the 63rd degree polynomial) Can we do better? If we can then we can add larger supported polynomial sizes to random().
- If you can call unrankRand() and get a state from a number can you call rankRand() and get a number from a state.
- What about optimizing the algorithm as it stands (For the large ALFG's). The LC is too fast as it is.
- We have evidence that srandom() really sucks…. What function can be added to improve the situation.

(Basically I want enough so that I can publish…)

- Code must be linked into a library for all to use.

Diagonalization

Goal: Find U and D such that

$$A = \left(UDU^{-1}\right) \qquad \text{(D is diagonal)}$$

$$A^B = \left(UDU^{-1}\right)\left(UDU^{-1}\right) \underset{\ldots}{\text{n times}} \left(UDU^{-1}\right)$$

$$A^B = UD\left(U^{-1}U\right)D\left(U^{-1}U\right)\cdots\left(U^{-1}U\right)DU^{-1}$$

$$A^B = UDD \underset{\ldots\ldots\ldots}{\text{n multiples of D}} DU^{-1}$$

$$A^B = UD^BU^{-1} \qquad \text{(Since D is diagonal computing } D^B \text{ is O(n logB))}$$

Through a little linear algebra you can show that D is a diaginal matrix of eigenvalues and U's columns are the eigenvectors in modular arithmatic.

$$\text{Ax} \equiv \lambda\text{x mod m}$$

Diagonalization Progress

Using a 'Aluru' style permutation matrix we find fow the default order of 31....

$$\left(A-\lambda I\right)=0 \Rightarrow \lambda^{31} - \lambda^{3} - 1 = 0 \quad \text{(Characteristic equation)}$$

This polynomial shows up elsewhere in the analysis of ALFG's. The generators parameters were chosen so that this polynomial was irreducable over $M = 2^{32}$.

In regular algebra we get over this by inventing a root...

$$\left(x^2+1=0\right) \Rightarrow \left(x = \pm\ i\right)$$

Obvious choice has problems.....

$$\alpha^{31} = \alpha^{3} + 1$$

You must track all powers of α from α^{30} to α^{0}.....

Must find a way for searching for roots over arbitrary fields to succed with this approach

Optimizing the existing algorithm

- For the large polynomial matrix multiplication is roughly 2000 times slower than calling random()...
- Small cases get optimized to calling random() iterativly (anywhere from about 100 to 10000)
- Make the permutation matrix equivalent to more than polySize iterations to reduce matrix multiplies
- Create a function that attempts to find the best compromise between the 3 stages of the algorithm

Binary random() calls generated by LC

```
msb                           lsb
01001111110010000111010101111110
11101011000011011001101011011111
11101001111101000001001100101100
00000000101100111100000111110101
01101111110011100110101110001010
10111100110010000000010111111011
00001011000101100011011000011000
01101010101000110001000001110001
10000111010111111011011001010110
01101111110001100000011011010111
10110010000100110011011111000100
00101000010010100111010101011101
01110101000100011100000111100010
01101111110011100011100101110011
11000101110011010101100000110000
```

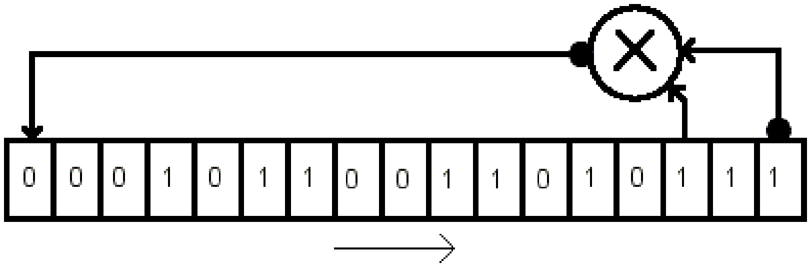
Basic idea for rankRand() with LC

Since the ith bit can only be 2^i periodic, we exploit this….

```
$RankRand\_LC(x)$
    $\mathrm{n} = 0, \mathrm{i} = 1$
    $\mathrm{while\ x} \neq \mathrm{x}_0$
        $\mathrm{tmp} = \mathrm{unrank}(x_0, 2^{i-1})$
        $\mathrm{if\ (x} \equiv \mathrm{tmp\ \ mod\ 2^i\ )}$
            $\mathrm{x} = \mathrm{unrank}(\mathrm{x}, 2^{i-1})$
            $\mathrm{n} = \mathrm{n} + 2^{i}$
    $\mathrm{Return\ \text{-}\ n\ mod\ M}$
$\mathrm{END}$
```

Progress ranking a ALFG

- The least significant bits of the state vector act like a Linear Feedback Shift Register (LFSR).
- Provided they aren’t all 0 the order is 2^{lag} -1



- The upper bits behave in a way similar to what we saw for LC generators, and can be ranked()….
- The period of our ALFG’s is $(2^{m-1})(2^{lag}-1)$ for this reason.
- If I can a fast rank for a LFSR then I’ve got one for ALFG’s
- The details are to involved to discuss here….

Note: I stole this diagram off the Internet

Monte Carlo calculation of pi

$$\int_0^1 \sqrt{1-x^2}\,dx = \frac{\Pi}{4}$$

$$Area = \int_{x_1}^{x_2} f(x)dx = \lim_{N\to\infty}\frac{1}{N}\sum_{r-1}^{N} f(x)(x_2 - x_1)$$

- Generate pairs of random numbers.
- Plug them into the equation to calculate ¼ of pi.
- Multiply final result by 4 to get actual approximation of pi.

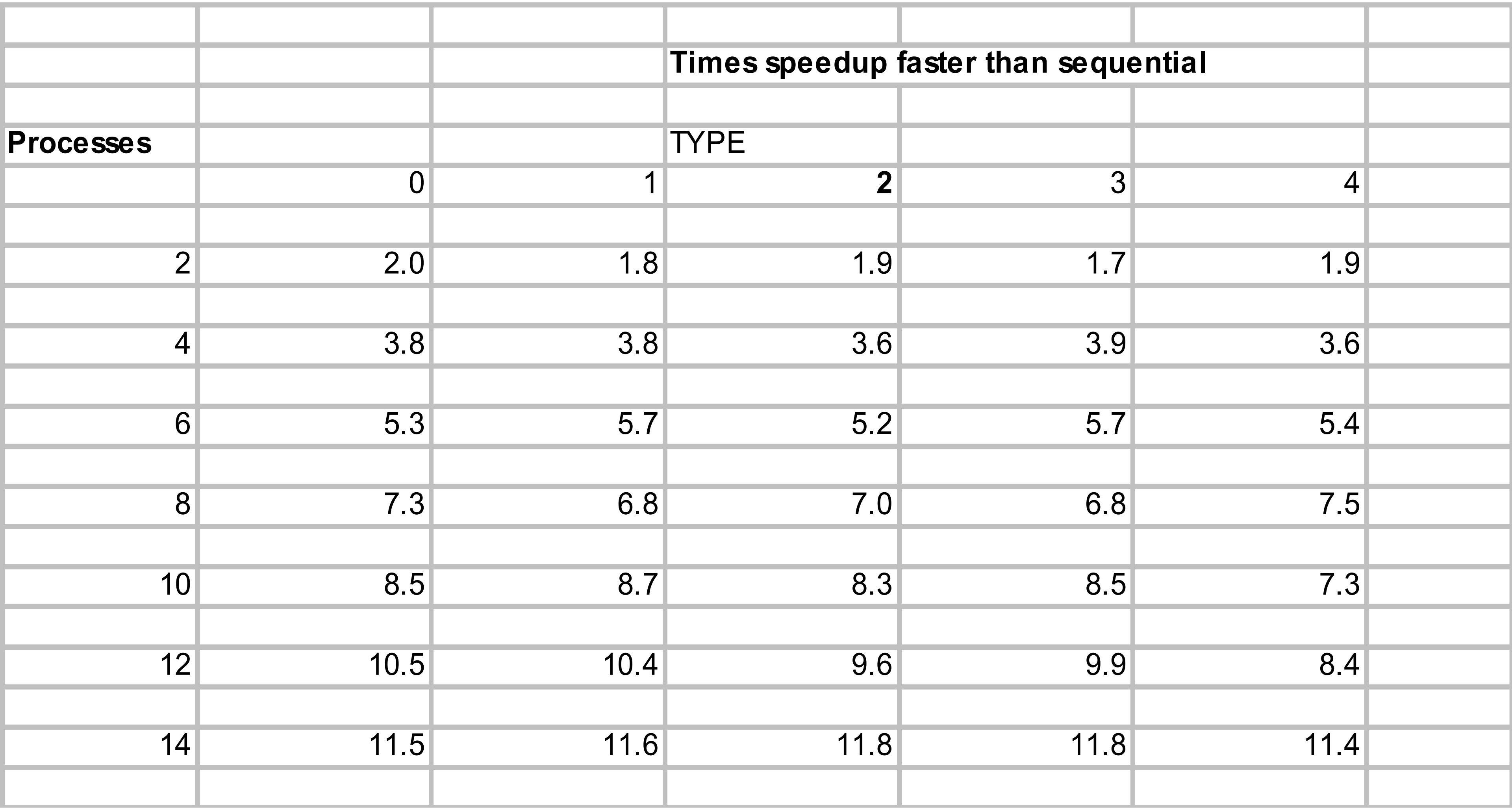
Parallel Implementation of Monte Carlo

- Each process knows how many random number to use.
- Each process calculates its portion of pi.
- Each process send partial pi to master.
- Master calculates approximation of pi.

PRNG = SRNG

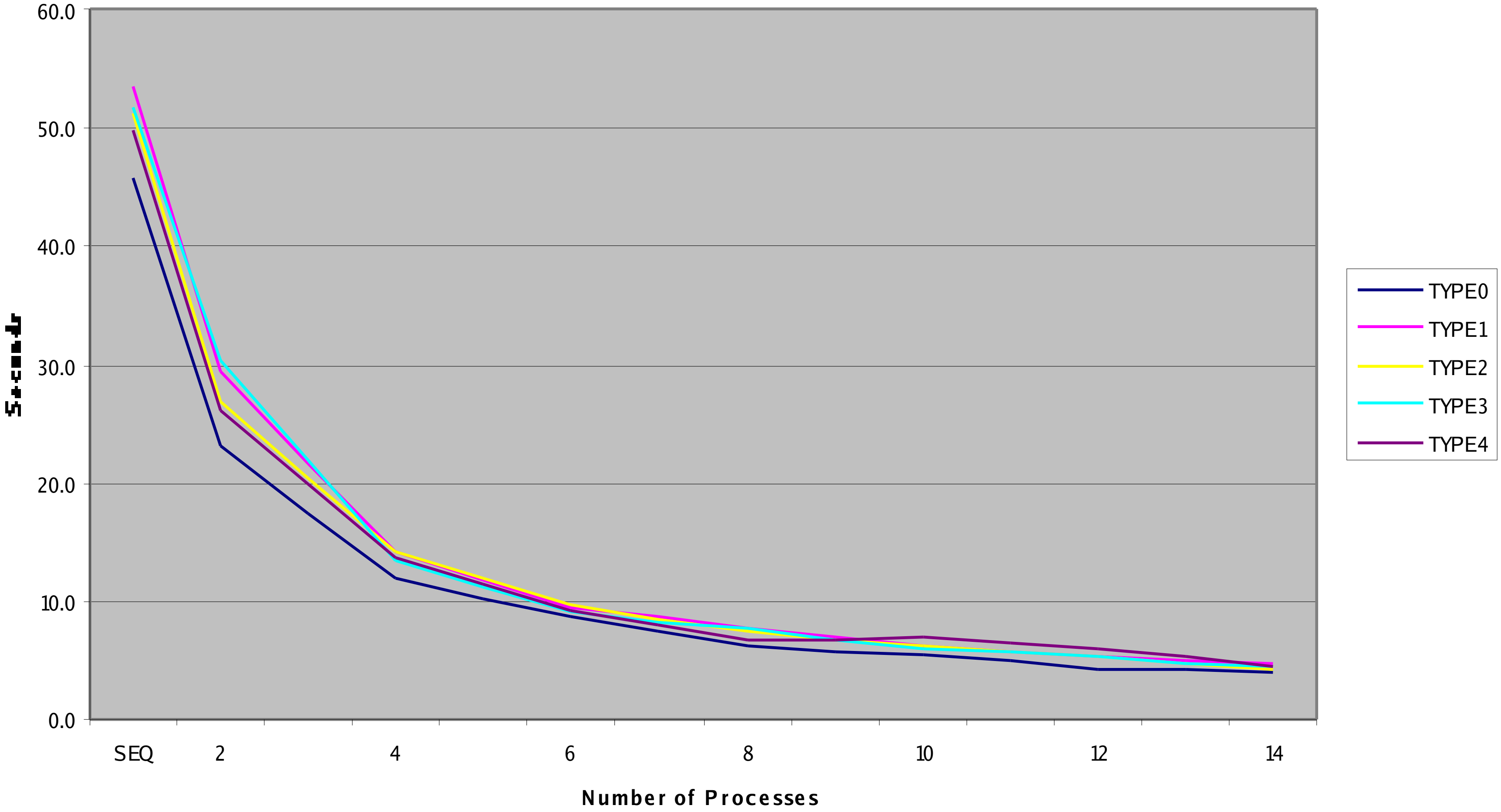
- Sequential and parallel generated random numbers are exactly the same.
- Proved this by generating 1 billion random numbers sequentially, and then generated 1 billion numbers using state calculation (unrankRand) and the two lists were identical.
- Reproducing same sequence of random numbers is important.

Times Speedup

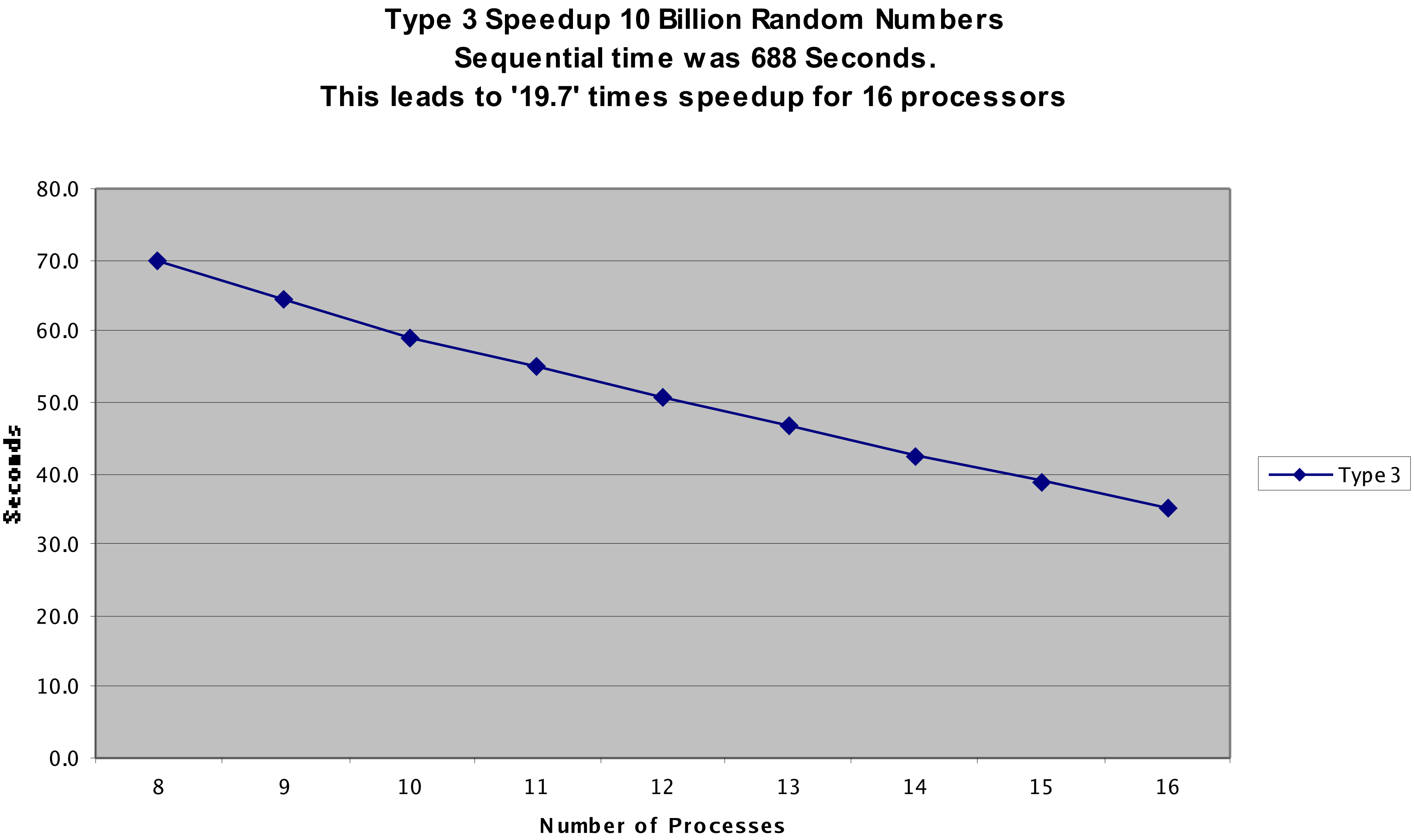


			Times speedup faster than sequential			
Processes			TYPE			
	0	1	**2**	3	4	
2	2.0	1.8	1.9	1.7	1.9	
4	3.8	3.8	3.6	3.9	3.6	
6	5.3	5.7	5.2	5.7	5.4	
8	7.3	6.8	7.0	6.8	7.5	
10	8.5	8.7	8.3	8.5	7.3	
12	10.5	10.4	9.6	9.9	8.4	
14	11.5	11.6	11.8	11.8	11.4	

Speedup Graph



Larger problem sizes



User Interface
Goal 1: Provide a simpler way to initialize parallel bucket sort

```
int* createProcessRandomTable(int me, int numProc, unsigned long n,
                              unsigned long* myRange);
```

me - a name I call myself
n - the total number of randoms to be distributed over the cluster
numProc - number of processors that will be participating

RETURNS:

An array of integers initialized so that the union of all number on the cluster is the same as would be for a serial call

On error the errno value is set and 0 is returned

myRange is populated with the number of integers assigned to this slave.
It is not assumed that n is divisible by numProc

Example Results: (serial and parallel match exactly)

SERIAL createProcessRandomTable(0, 1, 123,0)

```
0x1c2a96f3 0x5210418e 0x2f3e66fd 0x4124a372 0x23417bb3 0x1630e4f3 0x56165e02 0x396b0f5f 0x446fe8f1
0x7215ab58 0x68d36f75 0x440a2931 0x47078833 0x54390614 0x56b21e49 0x295e1c93 0x3c7dab41 0x3251b478
0x44935bb2 0x4f58ea54 0x12450236 0x9997771 0x3a10bd76 0x48f92344 0x3b041923 0x292fa451 0x7a5c608f
0x3d953e5 0x5b976632 0x292f7967 0x646e55ea 0x77c1fd25 0x7b3fbaf5 0x13acbce8 0x38e6a098 0x1e8136a9
0x29dda1db 0xefcfe9a 0x57ec4608 0x6e4d8acc 0x112a9f2 0x40bfb57d 0x3257b3fd 0x481a3225 0x14f8bb91
0x909d246 0x71784eb9 0x517666d3 0x3b5b86be 0x360baa6b 0x20cf5127 0x4da088f4 0x3fa521dc 0x5ae00e9e
0x1699ac38 0x7aa93aff 0x40fb2ef 0x10f60cc8 0x7e828ee5 0x5fa71921 0x3a25862f 0x62f0e4cf 0x57691647
0x35654125 0x769da1b7 0x104fb6df 0x53e677ce 0x207b4392 0x1f4cb579 0x2bd2bdd6 0xec8ce5f 0x205f5f6c
0x6c927354 0x4120825c 0x68799191 0x18b2ee5 0x4a2a54a3 0x59f1e04a 0x530195b8 0x585db61 0xffd8ab5
0x73d0e6e0 0x53266456 0x4fa2ac92 0x4eb0f57e 0x69c0108e 0x4a4be791 0x52c0a86d 0x7ab61d56 0x48ce7676
0x3267c18f 0x34dba386 0x2bbf5b46 0x9d0d7d6 0x6a40e4ab 0x225cfcfd 0x1a208eb5 0x3e275c79 0x42d84090
0x396d442e 0x69fa1a4f 0x51a10eef 0x59cca39a 0x568c8da3 0x12c1914b 0x4246352c 0x5817bc89 0x5cebe5ee
0x1c381576 0x2b195241 0x6271c150 0x2c35a02c 0x1eea3921 0x359825a6 0x7bd84cbe 0x6d9b2e9f 0x1f583634
0x4624344f 0x405bd70d 0x1a0e538b 0xef2aac6 0x72c3989c 0x4ee9f711
```

PARALLEL 15 calls createProcessRandomTable(0 through 15 , 1, 123, &range)

```
0: 0x1c2a96f3 0x5210418e 0x2f3e66fd 0x4124a372 0x23417bb3 0x1630e4f3 0x56165e02 0x396b0f5f 0x446fe8f1
1: 0x7215ab58 0x68d36f75 0x440a2931 0x47078833 0x54390614 0x56b21e49 0x295e1c93 0x3c7dab41 0x3251b478
2: 0x44935bb2 0x4f58ea54 0x12450236 0x9997771 0x3a10bd76 0x48f92344 0x3b041923 0x292fa451 0x7a5c608f
3: 0x3d953e5 0x5b976632 0x292f7967 0x646e55ea 0x77c1fd25 0x7b3fbaf5 0x13acbce8 0x38e6a098
4: 0x1e8136a9 0x29dda1db 0xefcfe9a 0x57ec4608 0x6e4d8acc 0x112a9f2 0x40bfb57d 0x3257b3fd
5: 0x481a3225 0x14f8bb91 0x909d246 0x71784eb9 0x517666d3 0x3b5b86be 0x360baa6b 0x20cf5127
6: 0x4da088f4 0x3fa521dc 0x5ae00e9e 0x1699ac38 0x7aa93aff 0x40fb2ef 0x10f60cc8 0x7e828ee5
7: 0x5fa71921 0x3a25862f 0x62f0e4cf 0x57691647 0x35654125 0x769da1b7 0x104fb6df 0x53e677ce
8: 0x207b4392 0x1f4cb579 0x2bd2bdd6 0xec8ce5f 0x205f5f6c 0x6c927354 0x4120825c 0x68799191
9: 0x18b2ee5 0x4a2a54a3 0x59f1e04a 0x530195b8 0x585db61 0xffd8ab5 0x73d0e6e0 0x53266456
10: 0x4fa2ac92 0x4eb0f57e 0x69c0108e 0x4a4be791 0x52c0a86d 0x7ab61d56 0x48ce7676 0x3267c18f
11: 0x34dba386 0x2bbf5b46 0x9d0d7d6 0x6a40e4ab 0x225cfcfd 0x1a208eb5 0x3e275c79 0x42d84090
12: 0x396d442e 0x69fa1a4f 0x51a10eef 0x59cca39a 0x568c8da3 0x12c1914b 0x4246352c 0x5817bc89
13: 0x5cebe5ee 0x1c381576 0x2b195241 0x6271c150 0x2c35a02c 0x1eea3921 0x359825a6 0x7bd84cbe
14: 0x6d9b2e9f 0x1f583634 0x4624344f 0x405bd70d 0x1a0e538b 0xef2aac6 0x72c3989c 0x4ee9f711
```

Other functions

int seedParallel(unsigned long seed, int me, unsigned long long int
numsPerProc)

Basically this places random()s state for each process so that if each process calls random() numsPerProc times the result is the same as the serial code starting with srandom()

unsigned long long seedParallelTotal(unsigned long seed,
int me,
int p,
unsigned long long int itr,
unsigned int grain);

Similar to seedParallel() but it divides the range for you and tells you where each split is. Also takes ‘grain’ for users whose loops consume more than 1 random number per iteration (like ours)….

int unrankRand(long long int stride)

All other functions are based off of this one. Duplicate the action of stride calls to random(). Erase the effect of -stride many random() calls if stride is passes as a negative number

Threaded Interface (currently half re-implemented….)

- Make multiple sources of random numbers available so that a user does not have to worry about what thread gets what random number
- Fully interlock the random objects so that nothing funny happens even if the user tries silly things
- Allow seamless converting of state from random()s internal state to a randomObject to a wrapped up array that works with glibs setstate
- Do so without slowing down random() itself

Threaded functions

par_initialize() - Initialize the thread safe version. There is the small possibility of a memory
leak if you don't call this function but the thread interface will work
without it. There is also the chance of a mis-synchronization.
This function is not needed if you don't use the par_* interface

par_generateRandomObject() - create a random number stream. All of the other par_
functions use this parameter to alter their values. If NULL is
passed into any of the par_ function for the stream field then
they operate on glibc's random state.

par_setsate()   - Thread safe version of the corresponding glibc function

par_initstate()   - Thread safe version of the corresponding glibc function

par_srandom()   - Thread safe version of the corresponding glibc function

par_unrankRand() - Thread safe version of unrank()

par_random()   - Not recommended. Thread safe, but slow. If you're not calling
random() while your calling any of the setstate(), etc functions
this is not needed. See wait_random() instead.

wait_random()   - This function will block until par_unrank() is called. After this
has occurred there is no time penalty for using wait_random() vs.
random().

random_rollback() - Function to help figure out what numbers were received from random
if you did not use the thread safe interface and you called
random() from one thread at the same time that unrankRand() was
being called from another

par_getSem()   - These function get and take the semaphore that protects the par_ interface
when a user has this semaphore he/she knows that any of the par_ functions
will block. This does not prevent the none par_ functions from executing.
NOTE: The wait_random()

par_releaseSem()   - This function will release the semaphore received by par_getSem()

Current Status

- Released code under the GNU lesser license
- Student interface is working and is very polished
- Threaded code is still under development. An Onyx crash made it impossible to finish this before submit. Will do so ASAP post semester recovery
- Library interface has changed to modify random()'s state instead of our own internal copy called t_random().
- Cluster would not fire up from a telnet session so have no way to verify that this change did not break the parallel Monte Carlo code
- I (Jason Main) have made myself available for supporting the library

Research Findings Summary

- The general idea was known since the early 90’s.
- Most people focus on Mascagni’s distinct subsequence formulation. Each processor gets its own independent source of random numbers. This is implemented in the complex ANSI funded library SPRNG.
- No implementations of this or similar algorithms was found.
- Future plans: Rank()
  Optimize
  Publish?
- QUESTION: Does srandom() work well for initializing ALFG’s? Further inquiry is needed.
- Leapfrog may be impossible. Aluru ‘proved’ that a communication-less leapfrog implementation was. What he may have actually proven is that you cant use the same sized ALFG
- Timing woes were caused by using timeofday() in the lab environment. Our 10 Billion number case got ‘super linear’ speedup by 30 % and our 1 billion number case got 70% of linear speedup.