Deterministic Parallel Random-Number Generation for Task-Parallel* Platforms (2012)

Charles E. Leiserson, Tao B. Schardl, Jim Sukha

Presented by Maximo Machado

*Paper refers to Task-Parallel as Dynamic Threading which is no longer in popular use
Why do we want determinism in programs?

- Debuggability
  - Determinism makes it easy to replicate situations in which the bug occurs
  - Allows experiments to be performed to determine what mismatches the expected result

- What does determinism mean in practice?
  - Typically, that the program’s execution is equivalent in behavior to the serial execution of a program
Why use a Task-Parallel Platform like Cilk?

- Parallelism is a desired feature which introduces non-determinism.
- Task-Parallel Platforms are able to control and manage this non-determinism through their schedulers.
- Contrasts with Static/POSIX Threads which are common but require the programmer to explicitly manage scheduling and load balancing.
Mitigating RNG Non-determinism

- RNG is made deterministic through a seed which is its initial state 0
- RNG on each request to generate a new number enters a new state, i.e. state 0 goes to state 1
- However, parallelism complicates things, suddenly execution ordering can impact the ordering of RNG requests, thereby making a simple seed approach ineffective
Global RNG – Potential Solution #1

- Share a single RNG between all threads through a lock
- Lots of contention on lock and so is not very performant
- The state the RNG is in when a thread issues a request is dependent on the execution order of all threads
Worker-local RNG - Potential Solution #2

● No lock needed since each worker has their own RNG
  ○ This solves the contention issue

● However, it cannot guarantee same RNG call goes to the same worker every time the program is run because of the non-deterministic scheduler

● Problem of these two solutions is that the RNG seed is based on the previous state, which is dependent on execution order

● How to create a solution based on some globally fixed ID?
Outline

1. Pedigrees
   ● The Solution

2. Dot-Mix
   ● Pedigree Based DPRNG*

3. Results

4. Conclusion

*Deterministic Parallel Random Number Generator
What are Pedigrees and How Can They Help?

- Uniquely identifies function on 2 key factors
  - What function spawned it? (Its parent)
  - How many functions did its parent spawn before it? (Its rank)
- This is scheduler independent
int fib(int n) {
    if (n < 2) return n;
    else {
        int x, y;
        x = spawn fib(n-1);
        y = fib(n-2);
        sync;
        return (x+y);
    }
}
DOT-Mix

- Pedigrees are variable length and the random numbers must fit into a fixed sized machine word
- We can’t directly use pedigrees to generate random numbers
- Use compression function that takes the pedigree and takes a dot product with random numbers to generate a word sized number
RC6 Block Cipher – The Mix in DOT-Mix

- The compression is not enough, highly correlated pedigrees will result in highly correlated numbers generated.
- We need a mixing function, the one in DOT-Mix is as such:
  - First, swap top and bottom half of bits of compressed value.
  - Then, apply function $f$ to compressed value $z$ for $r$ rounds.
    - Higher rounds creates greater overhead but better RNG.

$$f(z) = \phi(2z^2 + z) \mod m.$$
Results of RNG Quality

- Comparable to Mersenne Twister in Dieharder RNG benchmark
- Optimal choice of $r = 4$

<table>
<thead>
<tr>
<th>Test</th>
<th>$r$</th>
<th>Passed</th>
<th>Weak</th>
<th>Poor</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mersenne twister</td>
<td>16</td>
<td>83</td>
<td>6</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>84</td>
<td>6</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>81</td>
<td>5</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>DOTMIX (tree)</td>
<td>16</td>
<td>82</td>
<td>2</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>79</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>79</td>
<td>5</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>DOTMIX (loop)</td>
<td>16</td>
<td>84</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>79</td>
<td>4</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>55</td>
<td>2</td>
<td>8</td>
<td>42</td>
</tr>
<tr>
<td>LCGMIX (tree)</td>
<td>4</td>
<td>84</td>
<td>4</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>24</td>
<td>6</td>
<td>21</td>
<td>56</td>
</tr>
</tbody>
</table>
Results of Performance Overhead

- Adding Pedigrees to Cilk less than 1% overhead on real world applications
- DOT-Mix within reasonable range of overhead for debugging programs

Figure 8: Overhead of various RNG’s on the CBT benchmark when generating $n = 2^{20}$ random numbers. Each data point represents the minimum of 20 runs. The global Mersenne twister RNG from the GSL library [21] only works for serial code, while the worker-local Mersenne twister is a nondeterministic parallel implementation.
Conclusion + Future Work

- Extending 4-independent hash functions to Pedigrees
  - While unlikely, if a collision occurs, it is much more likely for DOT-Mix to produce many subsequent collisions
- Applications where pedigree memoization with incremental hash functions are performant