Thoughts on pseudorandom number generation in massively parallel platforms I

A. Augusto Alves Jr

Presented at Coffee-breaking meeting - KIT, Karlshuhe
May 7, 2020

Karlsruhe Institute of Technology

- Overview
- Standard sequential design
- Naive parallel designs and their pitfalls
- Safe design

Overview

A pseudo-random number generators (PRNG) are algorithms for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers.

- The PRNG-generated sequence is completely determined by the PRNG's seed.
- The PRNG-generated sequence has a finite length, called period of the PRNG.
- PRNG keeps an internal state, which advances every time an output is generated.
- PRNG user interface usually provides a `void discart(size_t n)` or `void jump_ahead(size_t n)` method. Calling this method will advance the PRNG state n times. Equivalent to calling the generator n times and discarding the result.

Standard sequential approach

This is the simplest and safest use case:

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

int main()
{
    unsigned int seed = 1234; //seed defined in the general scope
    std::mt19937 gen(seed); //prng instantiated in the general scope
    std::uniform_real_distribution<> dis(1.0, 2.0);
    for (int n = 0; n < 10000; ++n) {
        std::cout << dis(gen) << std::endl;
    }
    ...
}
```

The prng's state is stored in the main scope and will advance sequentially inside the loop. No race-conditions or any other problem.

What about this?

Running in parallel using `omp`:

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

int main()
{
    unsigned int seed = 1234; //seed defined in the general scope
    std::mt19937 gen(seed); //prng instantiated in the general scope
    std::uniform_real_distribution<> dis(1.0, 2.0);
    #pragma omp parallel for //run in parallel
    for (int n = 0; n < 10000; ++n) {
        std::cout << dis(gen) << std::endl;
    }
    ...
}
```

- Which thread will update the state first?
- What will happen if two threads try to update the PRNG's state at same time?

Comments

- If the PRNG implementation is thread safe, will be no crash, but numbers will be generated out-order in relation to the thread index.
- If it is not thread safe:
  - The PRNG's state could be updated to an inconsistent state.
  - Some members of the sequence could be repeated or miss completely.
  - If no hard crash happens, problem could pass unnoticed for a while.
  - The output pattern will depend from the system load etc...
  - If threads as managed by the user (`std::threads` , pthreads, gpus,...) this design does not work.

A safer design I

Generate sequentially, store and use in parallel:

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

int main()
{
    unsigned int seed = 1234; //seed defined in the general scope
    std::mt19937 gen(seed); //prng instantiated in the general scope
    std::uniform_real_distribution<> dis(1.0, 2.0);
    std::vector<double> rnumbers(10000);//allocate memory at once!
    for (int n = 0; n < 10000; ++n) {
        rnumbers.push_back(dis(gen));
    }
    #pragma omp parallel//run in parallel
    ...
}
```

- Fine, but can be slow if is not possible to pre-allocate the container.
- Puts pressure on the memory and CPU-caches and can defeat the efficiency gains of concurrency itself.
- Sooner or later a huge reallocation will be necessary.

A safer design II

Run an PRNG instance of per iteration, setting a different seed:

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

int main()
{
    unsigned int seed = 1234; //seed defined in the general scope
    #pragma omp parallel//run in parallel
    for (int n = 0; n < 10000; ++n) {
     std::mt19937 gen(seed+n); //prng instantiated in the general scope
     std::uniform_real_distribution<> dis(1.0, 2.0);
     std::cout << dis(gen) << std::endl;
    }
    ...
}
```

- Now the iterations are completely independent and no race condition should happen, but then potential problems with seeding show up:
- Different seeds does not guarantees independent streams.
- Certain PRNGs need to recover from “bad” seeds. Even worst, the recovery period depends on seed value.

The correct design

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

int main()
{
    unsigned int seed = 1234; //seed defined in the general scope

    #pragma omp parallel//run in parallel
    for (int n = 0; n < 10000; ++n) {
        std::mt19937 gen(seed); //prng instantiated in the general scope
        gen.discard(n); //advance the state
        std::uniform_real_distribution<> dis(1.0, 2.0);
        std::cout << dis(gen) << std::endl;
    }
    ...
}
```

- Now the iterations are completely independent and no race condition should happen.
- No problems with seeding.
- What could wrong this? Hint: it is starts with ".d"...

For the next meeting

- Discuss multithread backends explicitly managed by the user: `std::thread`, GPUs, TBB, etc.
- Present some measures of efficiency.
- Compare different PRNGs.