Adding a subsection for concurrent random number generation in C++17

Document #: P0010

Date: 2015-11-25

Reply to: Pattabhi < pattabhi@vcp.mobi >

Contents

Abstract

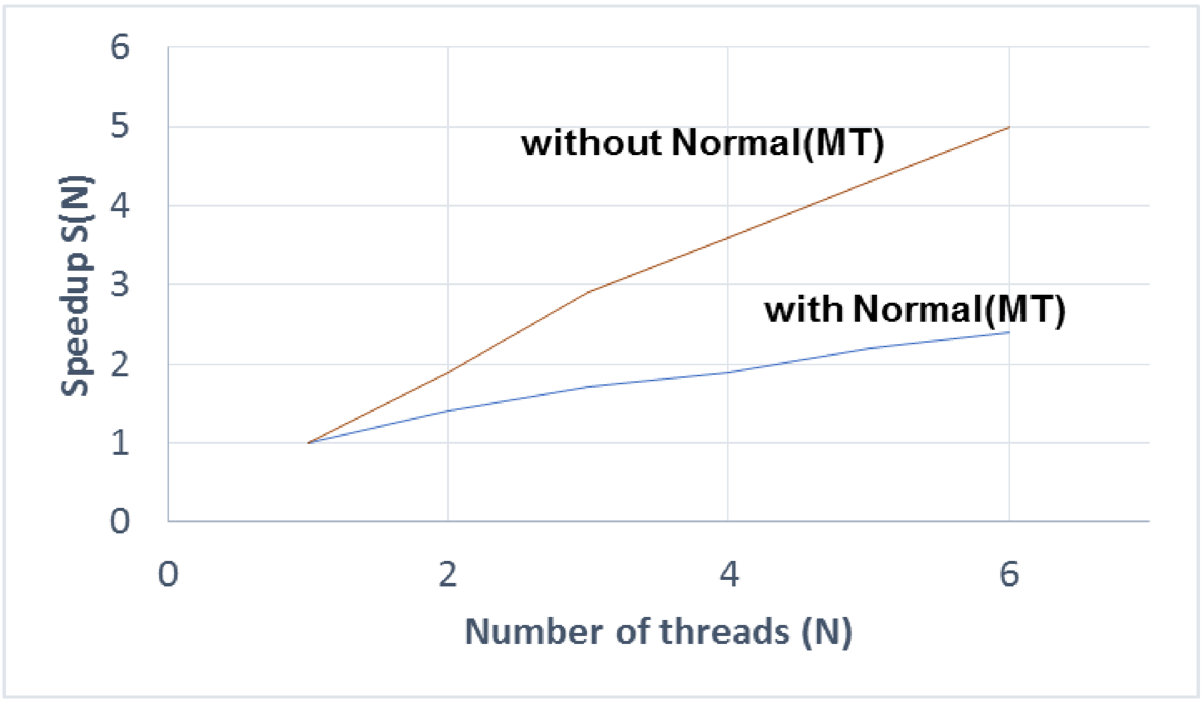
There is a performance and quality issue in running <random> generators concurrently. This paper discusses the scenarios and provides wording to implement a safe and effective generation of random numbers in concurrent mode.

Background and Proposal

The C++ provides some major new features such as lambda, auto and unordered containers [N4296]. Among these features, the thread library [Clause 30 Thread support library] and the <random> library [Sub clause 26.5 Random number generation] do provide helpful tools for high performance numerical simulations.

However, there is a performance and quality issue in running <random> components concurrently. This issue has been an important topic of discussion for the past three decades [Maclaren]. This section presents the various scenarios of concurrent random number generation and provides a recommendation / solution.

For example, the following figure depicts the concurrent speedup of a code with and without employment of normal_distribution (engine mt19937).



The figure shows fall of speedup due to the random number generation and the reasons are:

1. Random number generating member functions of engine classes are not constant, for example (**26.5.3.2 Class template mersenne_twister_engine):**

```
template<class UIntType, size_t w, ...>
class mersenne_twister_engine
{
public:
        ....
        ....
        ....
        explicit mersenne_twister_engine(result_type value = default_seed);
        template<class Sseq> explicit mersenne_twister_engine(Sseq& q);
        void seed(result_type value = default_seed);
        template<class Sseq> void seed(Sseq& q);

        // generating functions
        result_type operator()(); //it is not a const member function
        void discard(unsigned long long z);
};
```

As this generating function is not a constant, it is not safe for concurrency.

2. In concurrency, multiple threads reading data from a single source is safe, but writing data on a single source is not safe (unless enforcing a mechanism such as lock_guard).

   Receiving random number from a generator involves both reading and writing activities. That is once the generator delivers a random number then it begins to generate next number in the series. In other words the generator writes new number at each call. Thus random number generation is not naturally safe, i.e., data race is inevitable when multiple threads access same generator.

Thus the data race has reduced the speedup and potential duplication of same random numbers in more than one thread, which does not add up statistics.

The possible different remedies for this data race are:

a) Application of lock_guard mechanism. However, it slows down the process such that the speed is worse than that of running the code in serial.
b) Composition of individual variables of same engine for each thread, for example if number of threads for parallel action is four then declaration of individual engines is like:

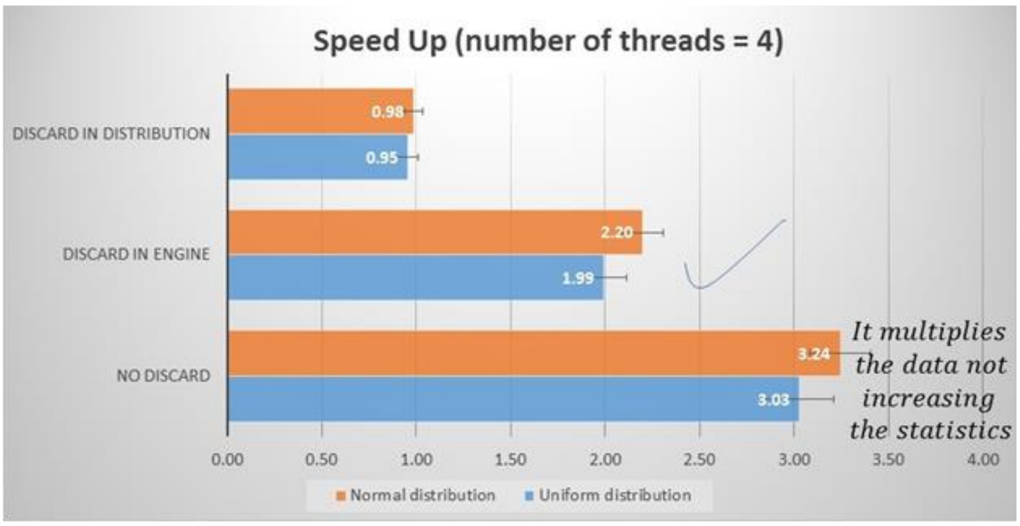
```
using given_engine = mt19937;
std::vector<Normal_Rnd<given_engine>>N(4);
where, object Normal_Rnd is like:
// A Normal distribution based on a
// given random number generator engine
template <typename given_engine>
class Normal_Rnd {
public:
	double operator()() { return normal(engine); }
	Normal_Rnd() :normal{ 0.0, 1.0 } {}

	void seed(unsigned int s) { engine.seed(s); }
	void seed() { engine.seed(); }
	void discard(unsigned long long z) { engine.discard(z); }
	void discard_distribution(unsigned long long z)
	{
		for (auto i = z; i != 0; --i)
			normal(engine);
	}
	Normal_Rnd(double mean, double std_dev)
		: normal(mean, std_dev) {}
private:
	given_engine engine;
	normal_distribution<double> normal;
};
```

i. If all engines begin with zeroth element in the random number series, then all threads would get same set of numbers, hence that would not add up the statistics. Unless, in a case, where each thread works on different task, skip ahead or saving state to avoid overlap is essential to gain meaningful statistics.

ii. Therefore, if x1, x2, x3 and x4 be the number of iterations for the first thread, second thread, third thread and fourth thread, respectively, then before beginning of iterative loop, engine object variable one would discard(0), variable two would discard(x1), variable three would discard(x2) and variable four would discard(x3). This would ensure good collection of random numbers with no data race. However, it has overhead of running many variables of engine. Also, the function discard has certain complexities.

iii. In some scenarios, users may prefer the function discard_distribution, however it is not much better than running in serial. The following picture depicts this scenario.



c) Composition of individual variables of different engines for each thread, for example:

Normal_ mersenne_twister N1;

Normal_ linear_congruential N2;

Normal_ subtract_with_carry_engine N3; etc.

This has an advantage of avoiding discard function, however it has a disadvantage of having more than one type of engine and it has statistical complexity.

The option b(ii) uses discard function. The effect of option b(ii) can be achieved faster and the speed-up can be as close to that of b(i) by use of **different seed** for each thread's engine. However, to achieve perfect skip-ahead with the efficiency, the use of different seeds demands better knowledge of given random number sequence.

Proposed wording

Proposed to add a subsection at the end of sub clause 26.5 Random number generation such as:

"26.5.9 Concurrent random number generation [concurrentrand]

The function call operator of the standard random number engine types is not const, so potentially concurrent calls on a single object conflict and may result in a data race. Data races may be avoided, for example, by using different objects (variables) of a random number engine in each thread. If a case requires to avoid same sequence of random numbers in each thread, then each random number engine should be initialized with different seeds such that each thread gets a different sequence of results."

This is supposed to be a note, but it could be preferred as a subsection just like the mini subsection 26.4.11 in sub clause 26.4 Complex numbers:

"26.4.11 Header <ccomplex> [ccmplx]

1 The header behaves as if it simply includes the header <complex>."

The advantages of having it in a subsection is that it provides an unambiguous position for this message in the vast sub clause for random number generation and it will be easy to refer in future development.

Acknowledgments

Many thanks, for their thoughtful comments and constructive discussions, to Hans Boehm, Paul A. Bristow, Lawrence Crowl, Peter Jäckel, Stephan T. Lavavej, Nick Maclaren, Alisdair Meredith, Roger Orr, Sam Saariste, Andy Sawyer, I.M. Sobol`, Herb Sutter, Anthony Williams, Michael Wong, members of BSI C++ panel, and more.

Special thanks to Jonathan Wakely for helping to construct the wordings, providing constructive suggestions and making necessary corrections in the draft.

Bibliography

Maclaren, N. (n.d.). Cryptographic pseudo-random numbers in simulation. *Lecture Notes in Computer Science , 809*, 185-190. Retrieved 1994, from http://link.springer.com/chapter/10.1007%2F3-540-58108-1_23

N4296: Working Draft, Standard for Programming Language C++. (2014). Retrieved from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf