J. Plasma Fusion Res. Vol.96, No.7 (2020) 367-371

講座 シミュレーションのための乱数入門

3．乱数発生の高速化と並列化技法

3. Techniques of Acceleration and Parallelizing the Random Number Generation

佐竹真介
SATAKE Shinsuke
自然科学研究機構 核融合科学研究所

（原稿受付：2020年3月17日）

現代のスーパーコンピュータを用いた大規模シミュレーションは，多数の計算ノードを用いた並列計算が標準的になっています．本章では，MPI分散並列プログラムにおける疑似乱数の並列発生方法について，数値計算ライブラリKMATH_RANDOMを例に解説します．また，乱数発生の高速化チューニングの実例や，物理乱数発生器を使う場合の高速化，並列化についてのアイデアについても紹介します．

Keywords:
pseudo random number, physical random number, parallel computing

・はじめに

本章では，大規模シミュレーションでの乱数の利用を想定し，乱数生成を効率的に行うためのプログラミング上の技法について解説します．現代の大規模シミュレーションでは特に，MPIやOpenMPといった分散並列，スレッド並列を駆使したプログラムを利用することが一般的になっていますから，そのような並列化プログラムの中で乱数を利用する上での留意点や工夫について紹介したいと思います．なお，本章では疑似乱数発生法として第2章で紹介したMersenne-Twister（メルセンヌ-ツイスター）法（MT法）を 対象として説明しますが，物理乱数器の利用を想定した場合の並列化や高速化に関するアイデアも紹介したいと思います．

まず乱数の並列化の説明に先立って，MPIによる分散並列と，OpenMP（あるいはコンパイラの自動並列化機能による）スレッド並列の違いを明確にしておきましょう．分散並列は，計算を実行するCPUがそれぞれ物理的，あるいは論理的に分割された異なるメモリ領域に配列や変数を記憶し，シミュレーションに必要な計算を各CPUが分担して行い，必要に応じて他のCPUと明示的な通信命令によってデータをやり取りする並列計算です．一方，スレッド並列は，CPUの中にある複数のコアが，同一のメモリ上のデータを使った演算を分担するやり方で，基本的にはFortranのdo loopやC言語のfor loopなどの繰り返し計算を分割してそれぞれのコアが並列計算する方法です．この両者を組み合わせたハイブリッド並列も大規模シミュレーションでは一般的に行われています．以下，分散並列における並列化の区切りを「プロセス」，スレッド並列におけるそれを「スレッド」と呼んで区別します．

さて，並列シミュレーションにおいて乱数を利用する上で留意しなければならないことは，プログラムの詳細にも依存しますが，基本的に分散並列の各プロセスは異なる乱数列を利用しなければならないということです．例えば，分子動力学（MD）シミュレーションでタンパク質の折り畳み構造を調べる計算を，10プロセスの分散並列で行うとしましょう．異なる初期配置を持ったタンパク質分子を，乱数を利用して多数用意する際に，それぞれのプロセスで同じ疑似乱数発生法，例えばMT法で，第2章で説明したseedの値も同じに取ってしまったら，10個のプロセスで全く同じ計算を開始してしまうことになり，並列化の意味がありません．そんなことはわかりきったことであり，seedの値をプロセス毎に変えればよい，と考える人もいるかも知れませんが，それはお勧めできません．疑似乱数のseedの値は，利用する乱数列の開始点を一周期のどこに取るかを決めるものだと2.1節で説明しましたが，例えばseedの値を10個のプロセスででたらめに（人手で決める，時計を使う，他の疑似乱数を使うなど）与えても，それらの部分列の開始点が互いに十分離れているという保証は全くありません．seedの選び方がたまたま悪くて2つのプロセスの乱数列の開始点が数個しかずれてなかった，ということも考えづらいですが，起こりえない話ではありません．各プロセスが参照する乱数の部分列が確実に重複しないように選ぶことは，プロセス間で擬似乱数によるおかしな相関関係が生じる懸念を排除するために必要なことです．それでは，具体的にどのように乱数発生を並列していけばよいのか見ていきましょう．

・Mersenne-Twister法による乱数生成の並列化

分散並列計算におけるMT法の乱数発生の並列化については，以前にもプラズマ・核融合学会誌の講座で紹介させていただいたことがあります[1]．その時に紹介したのは，

National Institute for Fusion Science, Toki, GIFU 509-5292, Japan
author's e-mail:satake@nifs.ac.jp

Dynamic Creator と呼ばれる，MT 法の考案者の松本眞氏ら自身が開発した技法[2]で，ソースコードも公開されています[3]．Dynamic Creator では，第 2 章で説明した，MT 法の漸化式の形を決めるパラメータのうち，ベクトル $(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})$ を変えることで，特性多項式が互いに素で，最大周期長がある Mersenne 素数 p に対し 2^p-1 となり，かつ高次元に均等分布する疑似乱数の漸化式の型を多数用意する，という方法を取ります．数学的に，特性多項式が互いに素な漸化式同士は相関を持たないとされているため，Dynamic Creator は言わば互いに独立な乱数表を複数用意することに相当します．ただし，特性多項式が互いに素なパラメータ $(\boldsymbol{a},\boldsymbol{b},\boldsymbol{c})$ の探索はトライ＆エラー的になされるため，数千並列の並列計算のための準備には非常に長い時間がかかります（筆者が十数年前に1024個用意するのに，ワークステーション 1 台を 1 か月以上稼働し続ける必要がありました）．

そこで，超多並列計算のための実用的な MT 法の並列化として今回紹介したいのは，Jump Method と呼ばれるものです[4]．MT 法の特徴である長大な周期長を活かし，1 つの疑似乱数列の中の，遠く離れた複数の点を並列計算で乱数を利用する開始点として明示的に指定する方法です．これは，先に述べたような seed の値を単に変化させるのとは異なり，開始点間の距離を $N \gg 1$ 個ずつ飛ばしに指定することができます．N 個飛ばしした後の漸化式の内部状態を並列プロセス数 P だけ事前に用意するには，原理的には N 個乱数を発生させては内部状態を記録する（第 2 章のサンプルプログラムにおける"mt_rand.txt"のデータに相当）という操作を P 回繰り返せば可能です．ですが，P 個の乱数列の部分区間が重ならないように N を十分大きく（100兆など）取り，かつ P が数千となると，実用的な方法とは言えません．しかし，Jump Method に使われる SIMD-oriented Fast Mersenne Twister（SFMT）という新しいタイプの MT 法[5]では，[6]に示される原理に基づいて N 個飛ばし後の MT 法の漸化式の内部状態に一気にジャンプできる計算ルーチンが用意されているので，それを利用することで乱数列の遠く離れた先の開始点から始めるための内部状態を高速に取得できます．SFMT は名前から推察できるように，高速化チューニングされた乱数発生ルーチンです．従って Jump Method の利用に限らず，逐次的に MT 法で疑似乱数を発生させる用途としても，従来のバージョンよりも SFMT を用いた方が高速実行が期待できます．

さて，実際の利用法についてですが，ここでは[5]のオリジナルのソースではなく，C，C++，Fortran90での実装が用意されている，理化学研究所の大規模並列数値計算技術研究チームが整備・公開しているライブラリ"KMATH_RANDOM"[7]をベースに紹介したいと思います．このライブラリの利用許諾についてはマニュアルの記載事項を参照してください．まず前準備として，NTL という数値計算ライブラリを別途ダウンロード，インストールする必要がありますが，[8]に説明があるので詳細は省きます．次に，KMATH_RANDOM を展開して，random/Makefile.machine というファイルを編集します．必要なのは，利用するコンパイラとコンパイルオプションの選択，NTL をインストールしたディレクトリの指定です．random/arch/以下にいくつかのコンピュータにおけるサンプルがあるので，それを参考にご自身の環境に合わせて設定します．なお，コンパイルするには MPI 環境も必須です．また，MT 法の周期長 2^p-1 はここでコンパイルオプション〝-DDSFMT_MEXP=p〟で指定します．make コマンドを実行し，コンパイルが完了すると，random ディレクトリ下の c/，c++/，f90/ 以下に libkm_random.a というそれぞれの言語用のライブラリができるので，自分のプログラムで利用するにはこれをリンクしてコンパイルすることになります．

次に，ジャンプファイルという N 個飛ばしの疑似乱数列の開始点の状態を記録したファイルを生成します．まず，make ptool を実行すると，random/ptool/に km_rand_gen_jump ができるので，これを実行します．

（例）km_rand_gen_jump -seed S1 S2 -max_ranks R -rand_range M

ここで，-max_ranks の値 R は想定される並列計算の最大プロセス数，-rand_range の値 M はジャンプの間隔 $N=2^M$ の指数部分を指定します．-seed の値 S1，S2 は，いわゆる通常の意味での seed の値を S1，S1＋1，…，S2 と変えたものを（S2－S1＋1）個生成するということです．これは N 個飛ばしで R 個用意する開始点全体の起点が異なるジャンプファイルを複数作成することを意味します．通常，1 つのシミュレーションに使うジャンプファイルはどれか 1 つを選べばよいですが，異なるジャンプファイルは，乱数を使ったシミュレーションを複数回行って統計平均を取りたい場合などに利用します．試しに S1＝1，S2＝2，R＝200，M＝100で実行すると，jump/ディレクトリに file_00001，00002 の 2 つのジャンプファイルのセットができます．これで，MT 法の疑似乱数に対し，2^{100} 飛ばしの開始点の情報が200個× 2 系列作られました．この計算は例えば最近の Xeon CPU 上では数秒で終わってしまいます．これは逐次計算で $2^{100}\times 200$ 個× 2 回，通常の MT 法で乱数を発生させるのとは比較にならないほど速いです．

これで前準備は完了です．次にこのジャンプファイルを使って実際に並列計算で SFMT を利用する方法を見ていきましょう．第 2 章と同様に，サンプルプログラム rand_test.f90 と km_rand_wrapper.f90 を GitHub に挙げておきます[8]．ifort コンパイラを利用する場合のコンパイル用 makefile のサンプルが用意してあるので，ご自分の環境に合わせて修正して make し，実行プログラム kmrand_time を作成してください．実行方法ですが，このサンプルプログラムは MPI とスレッド並列のハイブリッドコードになっているので，環境変数 OMP_NUM_THREADS にスレッド並列数を設定し，また KMATH_RAND_JUMP_FILE_PATH に先ほど作成した file_00001 等のジャンプファイルが格納されているディレクトリを絶対 path で設定します．そして，〝mpirun -np N kmrand_time〟のような MPI 実行コマンドで計算を開始してください．ここで N は試したい MPI 並列数（N はジャンプファイル作成時の-max_ranks の値 R 以下）です．

このテストプログラムでは，rand_test.f90 の冒頭に指定された，1 プロセス当たり ntest 個の乱数を，ntime 回発生させてその平均時間を計測します．そして，そのテストを nseed 個の異なるジャンプファイルに対して繰り返します．計測結果は"log.txt"に書き出されます．rand_test.f90 の主要部分を抜粋すると以下のようになっています．

```
----------------------------------------
call MPI_INIT(IERR)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nmpi,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
call KMATH_Random_Init(handle1,MPI_COMM_WORLD, ierr)

do k=1,nseed
   call KMATH_Random_Seed(handle1, k , ierr)
  do j=1,ntime
      call grnd_dsfmt(rnd1, ntest)
  end do
end do

call KMATH_Random_Finalize(handle1, ierr)
call MPI_Finalize(IERR)
--------------------------------
```

また，上のコードの流れは，以下のようになっています．

1．MPI_INIT で MPI 並列計算を起動
2．KMATH_Random_Init で KMATH_RANDOM の利用準備
3．KMATH_Random_Seed で k 番目のジャンプファイルを読み込む
4．grnd_dsfmt(rnd1, ntest)で配列 rnd1 に ntest 個の乱数を発生させる
5．KMATH_Random_Finalize で KMATH_RANDOM の利用終了
6．MPI_Finalize で MPI 並列計算の終了

このプログラムでは nseed 個のジャンプファイルについて，各プロセスで発生した乱数の最初の ntest 個を，"rand_s.txt"に全プロセス分をまとめて書き出します（全部を出力すると膨大になるので 1/100 に縮めてあります）．プロセス毎，seed 毎に異なる乱数列が生成されることが確認できると思います．

・KMATH_RANDOM の高速化チューニング

さて，このサンプルプログラムで発生速度が計測できるので，次に高速化について考えましょう．乱数の発生は grnd_dsfmt というサブルーチンで行っていますが，これは km_rand_wrapper.f90 の中に定義されています．実は，KMATH_RANDOM が用意する乱数発生ルーチン KMATH_Random_Vector は，高速化のために区間 (1, 2] の 8 バイト実数を，386個以上の偶数個発生するという少し変わった作りになっています．なので，1 回の必要な発生数が少ない場合や，奇数の場合の対処法が必要です．また，通常シミュレーションで使われる乱数は (0, 1] 区間に発生させた一様乱数を利用することが前提になっていることが多いですから，KMATH_Random_Vector で発生させた乱数を (0, 1] 区間に変換する操作が必要です．スレッド並列が有効になっている場合，ある程度の量の乱数列に対する変換をスレッド並列で一気に行った方が効率が良さそうです．更に，シミュレーションの中で不定長の乱数を発生させるために KMATH_Random_Vector を高頻度に呼ぶより，ある程度の大きさの，固定長 nbuff の乱数列を一気に生成し，必要な分だけ切り崩して使っていき，足りなくなったらまた一気に作る，というやり方にします．井戸水を使う人は，使うたびに必要な量を井戸まで汲みに行くのではなく，一定量水瓶にため込んでおいてそこから必要な量だけ柄杓ですくって使うのと同じように，ある一定量の乱数を貯めこんでから使う方法を，私は「水瓶方式」と呼んでいます．そのような一連の作業をしているのが grnd_dsfmt になります．

ここで，最も効率が良くなる nbuff を選ぶことで高速化につながりますが，その値は計算機の環境によります．**図 1** は，IFERC-CSC のスーパーコンピュータ JFRS1 上で，8 MPI×5 スレッド並列で kmrand_time を様々な nbuff と ntest に対して実行した際の，乱数100万個当たりの平均発生時間を表しています．ntest が大きくても小さくても，nbuff=64000 の時に最も乱数発生時間が短くなりました．この例では，チューニングの効果はあまり大きくありませんが，他の計算機環境では，特に ntest が小さい時に nbuff の選び方で数十%の速度の差が出ることもありました．なおこの水瓶方式ですが，grnd_dsfmt は MPI の各プロセスで独立に動くので，それぞれのプロセスで乱数の発生数やタイミングがずれていても問題ありません（並列化乱数発生法とは当然そうあるべきものですが）．また，2.2節で紹介したような物理乱数を利用する際も，物理乱数のデバイスにプログラムからアクセスして乱数データを取得するオーバーヘッドタイムを考慮すると，必要に応じて乱数を取得するよりも，水瓶方式を用いてある程度の量の乱数を一気に取得する方が効率がよいと考えられます．

・物理乱数発生器を利用した並列化

さて物理乱数の話が出たので，次に分散並列計算を行い

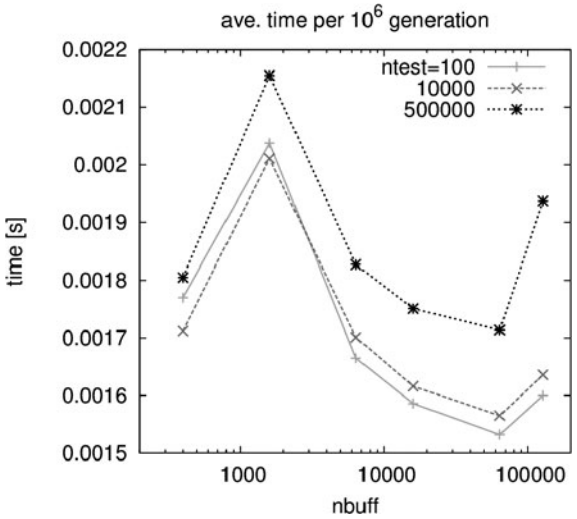

図 1　水瓶方式でバッファの大きさ nbuff を変えた場合の，乱数の発生時間の変化（百万個当たり）．nbuff = 64000 で極小値となります．

たいが物理乱数発生器が1つしかない場合，乱数の並列化をどうするか考えてみましょう．**図2**のように，計算の1ステップがCalc．A，B，C，…といったルーチンからなり，Calc.Bにおいて乱数が必要である場合を想定します．この時，例えば量子乱数発生器（QRNG）がrank0のMPIプロセスが走るノードにのみあるとします．考えられる並列化の方法としては，rank0は物理乱数の取得と，それを他のプロセスに分配することに専念し，他のプロセスが計算を担当するというやり方です．この際，「1ステップ当たり必要とされる乱数の最大個数（プロセス毎に異なる場合も含め）」×「計算に従事するプロセス数」の乱数をrank0のプロセスが乱数発生器から取得し，乱数が必要になるCalc.Bの手前のCalc.Aルーチンの間にMPI_ISCATTER（あるいはISEND/IRECVの組み合わせ）（＊）による非同期通信によって各プロセスに乱数を配信します．このやり方では，乱数を利用しないルーチンにかかる時間が，必要な乱数の発生にかかる時間に比べて長ければ，ほぼ待ち時間なしで並列計算が進められるため，計算の高速化にもつながります．各プロセスでは受け取った乱数を1ステップで全部利用しない場合もありますが，コードの実装を単純にするために，利用しなかった乱数は捨てて，次のステップで新しい乱数をrank0から受けとる形にします．この方法を私は「源泉かけ流し方式」と名付けました．温泉で源泉から汲んだお湯を，下流の各浴槽の使用量にかかわらず常に溢れるぐらい十分な量を流し込み続け，使わなかったお湯は再利用せずに捨ててしまう方式をイメージしたものです．Master-Slave形式の並列計算（パラメータ並列計算で，Masterプロセスが計算全体の管理を行い，手の空いたSlaveプロセスに次の計算パラメータを順次伝える形式）で初期条件に乱数が必要な場合にも，この源泉かけ流し方式でSlaveプロセスのどれかが計算を終えるまでに，Masterプロセスが次に使う乱数をため込んで準備しておく，という形での利用が可能です．また，乱数発生法が物理乱数に限定されるわけでもなく，疑似乱数を使った並列計算をしたいが，先に説明したJump Methodを準備するのが面倒という場合にも，一つのMT法で発生させた疑似乱数列を源泉かけ流し方式で各プロセスに分配して使うことも可能です．

次にMPI×OpenMPのハイブリッド並列の場合を考えます．もし**図2**のCalc.Aの計算時間が，QRNGで全MPIプロセスが利用する乱数を発生する時間より十分長い場合，rank0を乱数生成だけに専念させるのはCPUコアを無駄に遊ばせていることになりますので，rank0には計算にも従事してもらいたいです．そこで，Calc.Aの並列実行の仕方を**図3**のように変更してみましょう．この計算のプログラムのイメージは以下のようになります（GitHub[8]にもgensen_hybrid_img.f90として上げてあります）．

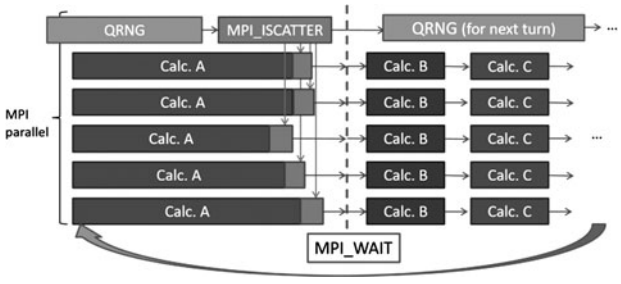

図2　MPI並列プログラムで量子乱数発生器（QRNG）が1つしかない場合の「源泉かけ流し方式」におけるプログラムの流れの模式図．rank0のプロセスは乱数生成に専念し，Calc.Aが終わった他のMPIプロセスは乱数を受信し，乱数を利用するCalc.Bに備えます．

```
   integer :: n,nmpi
   real(kind=8) :: rand(n*nmpi), rand_local(n)
!$OMP PARALLEL
!$OMP MASTER ! only master thread does the follow-
ing process
   if (myrank==0) then !assume MPI rank 0 has the
   QRNG board
      call QRNG_GEN(rand,n*nmpi) !generate random
      numbers
   end if
 call MPI_SCATTER(rand,n,mpi_real8,rand_local,n, &
 &              mpi_real8,0,mpi_comm_world,ierr)
     !scatter random numbers ( n per each MPI rank)
!$ OMP END MASTER
!$ OMP DO schedule (dynamic)     !thread parallel
   do j=1,m
      …  ! (Calculation A)
   end do
!$ OMP END DO
!$ OMP END PARALLEL
!... Both communication and calculation A are com-
pleted here.
   call sub_calc_B (n, a, b, c, ●●●, rand_local)
     ! Calc B which uses the random numbers
   call sub_calc_C (.....)
  ●●●
```

各MPIプロセスには，0〜k-1のk個のスレッドがあります．このうちの0番スレッドがMasterスレッドであり，Masterスレッドだけが，$OMP MASTER 〜 END MASTERの区間を実行します．Rank0のMPIプロセスのMasterスレッドは乱数を発生器から取得し，MPI_SCATTERでn個ずつ各プロセスに分配します．一方，Rank0以外のMPIプロセスのMasterスレッドは，乱数の受信に携わります．その間，全てのMPIプロセスにおいて，Masterスレッド以外の

（＊）**図2**では簡単のためMPI_ISCATTERを使っていますが，これではrank0自身にも乱数を分配することになります．この無駄を省くには，総プロセス数をnmpi，1プロセス当たりに分配する乱数の数をnとした場合，まずn×(nmpi−1)個の物理乱数をrank0で生成し，それを配列rand(n*nmpi)のn+1番目以降に保存します．配列randの先頭のn個の値は与えなくてよいです．そして，この配列randをMPI_ISCATTERで配信すれば，rank0自身には無意味なデータが渡り，rank1以降に生成した乱数がn個ずつ分配されます．

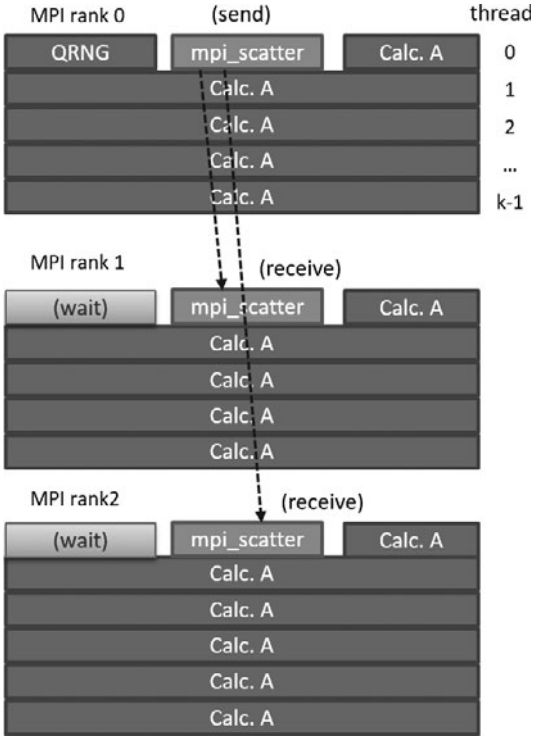

図 3 MPI-OpenMP ハイブリッドコードにおける源泉かけ流し方式の模式図. 各プロセスの Master スレッドは, 乱数の送受信が終わったところから Calc.A のスレッド並列計算に途中参加します.

1〜k-1 番スレッドは，Calc．A の Do loop を並列実行し始めています．$OMP DO に SCHEDULE (dynamic) が指定されているため，Master スレッドが乱数の生成と分配を終えて，まだ他のスレッドが Calc.A を終えていなければ，**図 3** に示すように Master スレッドは途中から Calc．A の並列計算に参加することになります．このようにすれば，源泉かけ流し方式でも乱数発生器を持つプロセスも計算に参加し，かつ乱数の分配の MPI 通信時間を Calc.A の演算に隠ぺいすることが可能です．もちろん，この方式が効率よく稼働するには MPI 並列数とスレッド並列数を適切に選び，Calc.A にかかる時間より乱数生成時間が短くなるようにしなければならないので，どんなプログラム・計算環境にも使える万能な方法ではないことに留意してください．

・KMATH_RANDOM における継続計算の方法

最後に，KMATH_RANDOM を使った計算を複数回継続する方法を説明します．第 2 章でもみたように，1 回のジョブの最後に MT 法の内部状態をファイルに書き出し，継続ジョブの最初にそれを読み直せばよいのですが，MPI 並列プログラムではそれを各プロセスが行う必要があります．更に，grnd_dsfmt のように水瓶方式を使っている場合，「水瓶」の中身も保存し，継続計算に引き継がなければなりません．そうでなければ，例えば100ステップを 3 回つないだ乱数を使った計算と，150ステップを 2 回つないだ計算とで途中から利用する乱数がずれてしまいます．

それでは，サンプルプログラムを使って，継続の仕方を具体的に説明していきましょう．km_rand_wrapper.f90 には，MT 法の内部状態と水瓶方式のバッファの中身を save，load するためのサブルーチン save(load)_grnd_dsfmt が用意してあります．その利用法のサンプル cont_test.f90 が GitHuB に上げてありますのでご参照ください [9]．はじめに，input_cont.txt 中の njob を 0 にして kmrand_cont を MPI 並列で実行すると，インデックスが iseed のジャンプファイルを使って新規に ntest 個の疑似乱数を各 MPI プロセスが生成し，"out_r**R_s**I_i000" に書き出します．ここで **R は MPI のランク，**I は選択したジャンプファイルの seed 番号です．また，乱数の継続用のファイル "rand_cont_s**I_i000"，"grnd_buff_ s**I_i000" も作られます．これらはそれぞれ，MT 法の内部状態と水瓶方式のバッファの状態を全 MPI ランク分まとめて書き出したものになっています．次に，njob=1 として再度 kmrand_cont を実行すると，上の継続用ファイル 2 つを読み込み，疑似乱数列の続きを ntest 個作ります．以降，njob=2,3,…とするごとに，"rand_cont_s**I_i**N"，"grnd_buff_ s**I_i**N" (N=njob-1) のデータを読んで疑似乱数列を一つ前のジョブの続きから生成します．ntest の数をいろいろ変えて実行し，各 MPI ランクが作る疑似乱数がそれぞれ異なる系列の乱数列で，かつ各々はジョブの区切りを跨いで連続した乱数列になることを確認してください．

駆け足になりましたが，これで KMATH_RANDOM ライブラリを利用した，並列化 Mersenne-Twister 法の利用と高速化と，物理乱数発生器を並列プログラムで利用する方法の解説はおしまいです．本章の内容とサンプルプログラムが，乱数を利用した多並列シミュレーションコードを書く方々の一助になれば幸いです．

参 考 文 献

［1］講座「核融合プラズマシミュレーションの技法　5．粒子シミュレーションのコーディング技法」プラズマ・核融合学会誌 **89**, 245 (2013).
［2］M. Matsumoto and T. Nishimura, "Dynamic Creation of Pseudorandom Number Generators", Monte Carlo and Quasi-Monte Carlo Methods 1998, Springer, 2000, pp. 56-69.
［3］http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html
［4］http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/JUMP/index-jp.html
［5］http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index-jp.html
［6］H. Haramoto *et al.*, "A Fast Jump Ahead Algorithm for Linear Recurrences in a Polynomial Space", Lecture Notes in Computer Science, vol. 5203, Springer, Berlin, Heidelberg (2008).
［7］https://www.r-ccs.riken.jp/labs/lpnctrt/projects/kmath-random/
［8］https://www.shoup.net/ntl/index.html
［9］https://github.com/satakeshinsuke/random_number/blob/master/Chapter3.tar