マルチスレッド環境からの
NAG 疑似乱数生成器及び準乱数生成器の呼び出し

1 概要

このテクニカルレポート及び関連するサンプルプログラムでは、マルチスレッド環境での NAG 乱数生成器の呼び出し方について説明します。サンプルは OpenMP（ http://openmp.org/） を使用して書かれていますが、NAG の呼び出しの基本構造は使用されるスレッドメカニズムに関係なく同じです。いくつかの OpenMP コマンドとプラグマはこのドキュメントで簡単に記述されていますが、詳細は OpenMP の Web サイトでご覧いただけます。また NAG は OpenMP のトレーニングコースを提供しています。詳細については nagmarketing@nag.co.uk 宛にメールでお問い合わせください。

まず初めに、疑似乱数及び準乱数について（非常に）簡単な概要を説明します。疑似乱数の数列は系統だった方法で生成された数列であり、独立していて真の乱数とは統計的に区別がつきません。準乱数（あるいは低食い違い量列）は多次元空間で均等分布をもたらすよう設計されています。したがって独立しておらず、簡単に真の乱数とは区別がつきます。疑似乱数は数列の独立性と「ランダム性」が重要な場合に使用される傾向があります。多くの場合、使用される手法の基礎となっている理論にこれらの特性が不可欠であるためです。一方、多次元のモンテカルロ法を使用する際には、準乱数の構造により疑似乱数よりも効率性が上がることがたびたびあります。NAG ライブラリには３つ目の種類の数列、スクランブル準乱数が含まれます。それは２種類の乱数を融合したもので、疑似乱数のランダム性をいくらか加えつつ準乱数の構造を保とうとしています（詳細はg05 チャプター・イントロダクションとその中の参考文献を参照）。

疑似乱数生成器は決定論的です。つまりそれらは決定論的手法で生成され、常に同じ数列を生成し、この数列は循環します。x_i が i 番目の生成値（任意の開始点からの）を示す場合、生成された数列は以下で与えられます。

$$x_1, x_{2,\ldots,} x_p, x_1, x_{2,\ldots,}$$

この場合 p は数列が繰り返す前の数列の長さを示します（通常 period と呼ばれます）。開始点は乱数生成器の初期化に使用されるシードによって制御されます。シードがないためいつも同じ位置から開始するという点を除いて、 NAG ライブラリの準乱数生成器も疑似乱数生成器と類似しています。 C ライブラリの Mark 9 では（また Fortran ライブラリの Mark 22 では）疑似乱数生成器を急速に前進(skip-ahead)させるためのルーチンが追加されました。これは $\mathrm{j} > 0$ の場合に x_i から x_{i+j} へ間の点を通らずに進むことが可能であることを意味します。この機能により、使用されたスレッドの数に関係なく同じ順番の数が生成されるように、マルチスレッド環境でルーチンを支障なく呼び出すことができます。初期化ルーチンに現れる skip ahead パラメータを使用することによって、同じことを準乱数生成器に対しても行うことができます。

2　サンプルプログラムの概要

このレポートには６つの異なるサンプルプログラムが含まれおり、こちらからダウンロードいただくことができます。そのうちの３つは C のサンプルプログラムで NAG C ライブラリからルーチンを呼び出しています。１つ目は疑似乱数生成器（一様分布）を呼び出し、２つ目は準乱数生成器（Sobol生成器）を呼び出し、３つ目はスクランブル準乱数列（Sobol生成器に基づく）を生成します。他の３つのサンプルプログラムは同様の処理を行いますが、これらは Fortran で書かれており NAG Fortran ライブラリを使用しています。これらのサンプルでの応用、つまり生成された数列の利用のしかたは、平凡であり実用性はありません。数列の総計（あるいは準乱数生成器の場合は一次元からの値の総計）を行っているだけです。しかし、これらのルーチンをどのように呼び出すことができるかを説明するのに十分であり、またルーチンが使用されるスレッドの数に関係なく同じ結果をもたらすことを確認するのに十分です。ここで添え書きをしておく必要があります。合計は多くの場合等しくなりますが、生成された数列が合計される順番によっては実装時にわずかな差異がありえます。

このセクションの後半では、疑似乱数生成器の呼び出しに関する、最初の C のサンプルプログラム（CPseudo_OpenMP.c)を詳しく説明します。残りの５つのサンプルプログラムは類似した構造になっており、詳しく注釈されています。以下の記述は、それらのサンプルプログラムの中で何が行われているかを十分説明しています。

プログラム CPseudo_OpenMP.c は以下のように構造化されています：

(a)　コードのシリアル部分の開始。このサンプルではここでユーザ提供パラメータ:npaths、iskip と seed を読み込み、state 配列の長さを計算します。Wichmann Hill Ⅱ生成器を使用しているためこれを 29 に設定することができますが、そうはせずに nag_rand_init_repeatable (g05kfc)の機能を利用し、lstate = -1 にしてこのルーチンを呼び出して必要な長さを返しています。

```
lstate = -1;
nag_rand_init_repeatable(genid, subid, seed, lseed, state, &lstate, &fail);
```

この呼び出し後、lstate は必要な値を保持します。

(b)　コードのパラレルセクションの開始：

```
#pragma omp parallel default(none) ¥
  reduction(+:sumRandom)
    private(ThreadNo,tskip,tnpaths,x,fail,i,t1,NumberOfThreads,state)
      shared(npaths,exit_status,iskip,genid,subid,lstate,seed,lseed)
```

このプラグマは同じコードを実行する各スレッドと並行してコードの次のブロックが実行されることをコンパイラに知らせます。またprivate ステートメントの後に括弧内にリストされている全ての変数、ThreadNo, tskip　などがプライベート変数であり、したがって各スレッドはそのコピーを維持する必要があることをコンパイラに知らせます。同様にshared ステートメントの後に括弧内にリストされている各変数はスレッ

ド間で共有することができるため、一つのコピーだけが存在します。最後にreduction ステートメントは各スレッドに変数 sumRandom の独自のコピーがあることを示しており、parallel ブロックの最後でそれらのコピーがそれぞれまとめて追加されます。

(c) 各スレッドが生成する必要のあるパスの数 (tnpaths) とスレッドを開始位置まで持っていくためにスキップされる必要のある数列の量 (tskip)を計算します：

```
NumberOfThreads = omp_get_num_threads();
```

現在のスレッドのスレッドID を得ます (最初のスレッドは 0、２つ目のスレッドは 1 など) ：

```
ThreadNo = omp_get_thread_num();
```

tnpaths と tskip を以下のように計算します：

```
if (NumberOfThreads == 1) {
tskip = 0;
tnpaths = npaths;
} else {
t1 = npaths % NumberOfThreads;
t1 = (ThreadNo < t1) ? ThreadNo : t1;
tskip = ThreadNo * (npaths / NumberOfThreads) + t1;
tnpaths = (npaths + NumberOfThreads - ThreadNo - 1) / NumberOfThreads;
} tskip += iskip;
```

このようにtnpaths を計算すると全てのスレッドに渡りできるだけ均等に作業が分けられます。例えば、５つのスレッドがあり 104 個の値を生成したい場合、最初の４つのスレッドはそれぞれ 21個の値を生成し（つまりtnpaths = 21）、最後のスレッドは残りの 20個の値を生成します。最初のスレッドで生成される21個の値を超えてスキップする必要があるため、最初のスレッドのスキップは 0、２番目のスレッドのスキップは 21（つまりtskip = 21）です。同様に、３つ目、４つ目と５つ目のスレッドのスキップはそれぞれ 42、63 と 84 です。最終行 tskip += iskip; は最初のスキップの使用を可能にしています。従って例えばもし以前のNAG 生成器の呼び出しで1000個の値を生成し使用していた場合、以前に生成した値を再利用しないように iskip = 1000 を設定します。

(d) 各スレッドが NAG C ライブラリエラー処理構造 (NagError)の独自のコピーを必要とするため、parallel region 内のC ライブラリルーチンを呼び出す前にエラー処理構造を初期化することが重要です。

```
INIT_FAIL(fail);
```

このとき、プライベート配列用のメモリを割り当てる必要があります。この場合は生成された値を保持する x と乱数生成器の状態を保持する state です。

(e) 乱数生成器の初期化：

```
nag_rand_init_repeatable(genid, subid, seed, lseed, state, &lstate, &fail);
```

疑似乱数生成器の現在の状態が配列 state に格納されます。各数字が生成もしくはスキップされるときに配列が変わるため、各スレッドは配列の独自のコピーを必要とします。さらに、各スレッドは同じseedを使用して state 配列を初期化させる必要があります（同じ位置で開始できるようにします）。各スレッドで nag_rand_init_repeatable (g05kfc) を呼び出すのではなく、parallel セクションを始める前にこのルーチンを一度だけ呼び出すことが可能です。そして parallel ブロックでの nag_rand_init_repeatable (g05kfc) の呼び出しをmemcpy への呼びだしに置き換えて各スレッドに効果的に state 配列のコピーを作成させることができます。

(f) 各スレッドで tskip の数の skip-ahead を行う：

```
nag_rand_skip_ahead(tskip, state, &fail);
```

(g) 一様分布から tnpaths 値を生成:

```
nag_rand_basic(tnpaths, state, x, &fail);
```

(h) 値が生成されると、何らかの方法でそれらの値を使用することができます。次の例では、それらの値の合計を求めています：

```
for (i = 0; i < tnpaths; i++)
  sumRandom += x[i];
```

parallel region を初期化する際に reduction 命令で sumRandom を宣言しているので、各スレッドの個々の寄与度の合計はコンパイラによって処理されます。

(i) 最後に、コードの parallel セクションを終了する前に parallel セクションで割り当てられたメモリ、この例では x と state のメモリを解放する必要があります。

3 非一様疑似乱数

NAG ライブラリには 、CPseudo_OpenMP.c や FPseudo_OpenMP.f90 で使用される一様分布のほかに30個を超える異なる分布から値を生成するためのルーチンがあります。これらのルーチン全てをスレッドセーフで使用することができますが、統計的特性を保証する、あるいはスレッドの数に関係なく同じ順番の値が確実に生成されるように使用できるとは限りません。なぜなら非一様分布からの値が生成されるからです。

全ての非一様分布生成器は一様分布生成器から始めて、その一様分布生成器からの値を使用し必要な分布の値を得ます。NAG ライブラリはこれについて３つの基本的な手法を使用します：変換法、表検索法、棄却法です（g05 チャプター・イントロダクション参照）。全て理想的に進んだ場合には、興味のある分布の一つの値が一様分布の一つの値から生成されます。累積分布関数の逆数に基づいている場合に、これは表検索法と変換法で最も頻繁におきます。C ライブラリの Mark 8 では以下のルーチンは入力パラメータの全ての値に対して１対１の関係を持ちます。そのためセクション２で記述されているサンプルプログラムで説明されている方法と同一の方法で呼び出すことができます：

一様分布(nag_rand_basic (g05sac)), 指数分布(nag_rand_exp (g05sfc)), 正規分布(nag_rand_normal(g05skc)), ロジスティック分布(nag_rand_logistic (g05slc)), 対数正規分布 (nag_rand_lognormal (g05smc)),三角分布(nag_rand_triangular (g05spc)), Weibull分布 (nag_rand_weibull (g05ssc)), ２項分布binomial (nag_rand_binomial (g05tac)), 論理分布(nag_rand_logical (g05tbc)), 幾何分布(nag_rand_geom(g05tcc)), 離散分布(nag_rand_gen_discrete (g05tdc)), 超幾何分布(nag_rand_hypergeometric (g05tec)), 負の２項分布(nag_rand_neg_bin (g05thc)), ポワソン分布(nag_rand_poisson (g05tjc)) 及び離散一様分布 (nag_rand_discrete_uniform (g05tlc))。

Fortran ライブラリの Mark 22 の同等ルーチンも１対１の関係があります。上記の名前のルーチンと同等の Fortran ルーチンを得るには、ショートネームを見つけ、それを大文字に変えて最後の文字を F に置き換えます。たとえばショートネームが g05sac であるnag_rand_basic (g05sac) と同等のルーチンは、G05SAF です。１対１の関係をもたないルーチンの場合は、つまりその他の疑似乱数生成ルーチンの場合は、使用されるスレッド数に関係なく同一である数列を得るのは不可能です。さらに、数列の特性を保証するのは不可能です。これらのルーチンを処理する唯一の方法は、基礎となっている一様分布からの値が確実に再利用されないようにするために各スレッドを「十分に遠くまで」 skip-ahead させることです。しかし「十分に遠くまで」がどのくらい遠くであるかを説明することは不可能です。しかし遠くまで skip-ahead するほど古い値を再利用できなくなります。例えば、もし各スレッドで生成された値ごとに100 個スキップさせると、2 個スキップさせた場合と比較して一様分布生成器からの値を複数回使用する可能性が低くなります。

4　準乱数の例

準乱数列のサンプルプログラムは上記で説明した疑似乱数のサンプルプログラムに非常に似ています。そのためここでは詳細な説明は行いません。この２つの主な違いは準乱数初期化ルーチン nag_quasi_init (g05ylc) が state 配列を持たず、代わりに同じ役目をする iref 配列を持つという点です。そのため同じ様に処理されます(つまり各スレッドが独自のコピーをもちます)。その他の違いは、準乱数初期化ルーチンがインターフェースで skip パラメータ (iskip) を持つことです。そのため乱数列をスキップさせるために別のルーチンを呼び出さなくても、このパラメータは skip 値 (tskip) に設定されます。

疑似乱数生成器と準乱数生成器の両方ともに初期化する必要があるため、スクランブル準乱数列のサンプルプログラムは他の２つを融合しています。準乱数列用の２つの分布の生成器、 正規分布(nag_quasi_rand_normal (g05yjc))と対数正規分布 (nag_quasi_rand_lognormal (g05ykc)) は逆 CDF (逆累積分布関数) 法に基づいています。したがって基礎となっている一様分布生成器と１対１の関係があります。これらは CSobol_OpenMP.c と FSobol_OpenMP.f90 の一様分布生成器と同様に処理されます。

5　サンプルプログラムのコンパイル

サンプルプログラムは GNU コンパイラコレクション(http://gcc.gnu.org/)で以下のコマンドを用いてコンパイルすることができます：

C Examples

```
gcc CPseudo_OpenMP.c -I[INSTALL_DIR]/include [INSTALL_DIR]/lib/libnagc_nag.
a -lpthread -lm -fopenmp
```

Fortran Examples

```
gfortran FPseudo_OpenMP.f90 -fopenmp -lm [INSTALL_DIR]/lib/libnag_nag.a
```

ここで［INSTALL_DIR］はユーザがインストールした NAG ライブラリのディレクトリの名前です。プログラムの NAG ライブラリへのリンクについての詳細は、各実装のユーザノートをご参照下さい。ユーザノートはライブラリと一緒に送付されているか、もしくは http://www.nag.co.uk/numeric/CL/CLinuns.asp 及び http://www.nag.co.uk/numeric/FL/FLinuns.asp からご利用いただけます。他のコンパイラを用いた OpenMP プログラムのコンパイルの例につきましては、各コンパイラのドキュメントをご参照下さい。

6　SMP 及び Multicore 用 NAG ライブラリ

一連のライブラリと同様に、NAG は SMP & Multicore 専用のライブラリ（http://www.nag.co.uk/numeric/FL/FSdocumentation.asp）を提供しています。この SMP ライブラリには NAG Fortranライブラリで見られる全てのルーチンが含まれています。これらは同じインターフェースを持ちますが、多くのルーチンはチューニングされています。つまりマルチコアのマシン上で稼働する際に NAG Fortran ライブラリの同等ルーチンのパフォーマンスを改善するために並列化もしくは最適化されています。このレポートを執筆している時点ではこのライブラリの最新リリースである Mark 22 にはチューニングされた準乱数生成器が含まれています。最新の Mark ではチューニングされた疑似乱数生成器が含まれる予定です。それらは１対１マッピングを使用しない分布用ルーチンを含んでおり、そのためセクション３で記述されている方法を用いて簡単に並列化することはできません。