

多线程环境中调用 NAG 虚拟随机 (Pseudo) 与 准随机 (Quasi) 随机数生成器

1. 前言

本文包含相关的示例程序，我们将说明如何在多线程的环境中调用 NAG 随机数生成函数。这些例子中使用到 OpenMP (<http://openmp.org>) 来调用 NAG 函数，此架构采用与线程程序设计相同的方法。部分的 OpenMP 命令与编译控制指令 (pragmas) 在本文中有相当简略的介绍，其他有关 OpenMP 的说明与介绍，您可以直接在其网站上取得。

在开始前，我们先简单的介绍虚拟随机与准随机数。所谓虚拟随机随机数是以系统性的方法产生一连串的随机数序列，随机数序列都是独立的，且统计学上没有任何的区别。准随机 (或称低差异性) 随机数，是为了要在多维的空间中取得更为平均的分布，所以其并非独立的，而且很容易能够自真实的随机数序列中分辨出来。通常虚拟随机是应用在序列中具有独立性与随机数性的问题上。但另一方面，准随机的结构性往往在使用多维度的蒙地卡罗模拟上，比虚拟随机更有效率且合适。NAG 算法库中包含了第三种随机数序列 - scrambled quasi-random 随机数，它混和了以上两种的特质，保有准随机的结构但加上了虚拟随机随机数的随机数特性。您可以参考 [g05 章节](#) 中有关的介绍与说明。

虚拟随机随机数生成器具有确定性，他们是以确定性的方法产生随机数，所以会产生相同的随机数序列，且其序列会循环。若我们用 x_i 表示第 i 个产生的随机数 (自任意起始点开始)，则产生出来的序列如下：

$x_1 ; x_2 ; \dots ; x_p ; x_1 ; x_2 ; \dots$

其中 p 表示序列的长度，经过一个周期后又会有相同的序列。并透过起始种子 (seed) 来启动随机数生成器。在 NAG 算法库中的虚拟随机随机数生成器也相类似，但是他们并不需要种子 (seed)，所以随机数会有相同的开始值。在 NAG C 算法库第 9 版与 NAG Fortran 第 22 版中的虚拟随机函数都加入快速往前 (skip-ahead) 的功能。这表示我们可以轻易的在 $j > 0$ 条件下，不须要经过中间任何的点就可从 x_i 跳到 x_{i+j} 。这样的功能可以很轻易的让函数可在多线程的环境中安全的使用，不论配置多少线程都能获得相同的随机数。对准随机产生器而言，也可透过初始化阶段的函数设定使用此 skip ahead 功能。

2. 示例程序

本文中探讨了六个不同的示例程序，您可以自 [此处](#) 下载。三个是 C 程序，且调用 NAG C 算法库。第一个调用虚拟随机产生器 (均匀分配)，第二个调用准随机 (Sobol 产生器)，第三个也是采用 Sobol 产生器的 scrambled quasi-random 随机数函数。另外三个程序也是一样的方法，但是以 Fortran 语言编写，并调用 NAG Fortran 算法库。这些例子中的应用仅仅是产生随机数序列，且没有特别的加总序列 (在准随机随机数的例子中，仅加总第一维的随机数值)。然而，这些足以用来说明如何调用函数，并能在不同的线程环境下验证函数是否会产生相同的结果。在大多数状况下，多线程环境中加总的结果会是一致的，但在某些情况下会因为产生序列加总的顺序不同有可能会稍有稍微差异。

接下来，我们将会详细的说明 C 语言的一个例子 (CPseudo_OpenMP.c)，它调用 NAG C 算法库的虚拟随机产生器。其余的五个程序都有相同的架构，我们也在程序中加入大量的批注说明，希望您透过以下的说明可以了解其他部分的细节。

CPseudo_OpenMP.c 程序结构如下:

- (a) 开始时运行一些串行程序。在程序中, 我们读取须由使用者输入的参数: `npaths`、`iskip`、`seed`, 及状态数组 (`lstate`) 长度, 当我们采用 Wichmann Hill II 产生器时, 我们必须指定其为 29; 我们可以透过 `nag_rand_init_repeatable (g05kfc)` 给定 `lstate = -1`, 传回实际所需要长度值。

```
lstate = -1;
```

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

当运行此后, `lstate` 会记录所需的长度值。

- (b) 并行处理部分:

```
#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)
```

`pragma` 命令指示编译器接下来的区段必须以并行的方式让每个线程运行相同的代码。同时也告诉编译器, 在 `private` 叙述后括号内的变量 (`ThreadNo`, `tskip` 等) 都是私有的, 也就是说每个线程都保有属于自己的一份。同样的, 在 `shared` 叙述后括号内的变量是共享的, 也就是每个线程共同使用同一个变量。最后, `reduction` 叙述表明每一个线程有一个自己的 `sumRandom` 变量, 当所有并行部分运行后, 每个线程的 `sumRandom` 相加起来。

- (c) 计算每个线程需要产生的随机数总量 (`tnpaths`) 与需要自起始点跳过的序列总数 (`tskip`):

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

取得目前线程的编号 (ID) (第一个线程编号为 0, 第二个为 1, 依此类推):

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

计算 `tnpaths` 与 `tskip`:

```
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;
```

以这个方法来计算 `tnpaths`，将会尽可能地在所有的线程中平均分配工作。举例来说：若有五个线程且要产生 104 个值，前面的四个线程会各自产生 21 个值 (也就是 `tnpaths = 21`)，最后一个线程将产生剩下的 20 个随机数值。第一个线程的 `skip` 值是 0，第二个线程会跳过 21 (也就是 `tskip= 21`)，跳过由第一个线程所产生的 21 个随机数值。同样的第三与第四第五的线程分别跳过 42、63 与 84。最后一行 `tskip +=iskip;` 让使用者加上原先就设定默认需跳过的位置。所以，举例来说，若我们先前已经产生了 1000 个值了，我们可以设定 `iskip = 1000`，用来避免重复使用到先前所产生的值。

- (d) 由于每个线程中都需要各自的 NAG C 算法库错误处理结构 (`NagError`)，所以若要在并行区块中调用 NAG C 函数时，就必须要先初始化此错误处理变量：

```
INIT_FAIL(fail);
```

在此，我们也必须要配置各线程中的私有变量 (程序中的 `x` 变量)，将会储存所产生出来的值，还有 `state`，它会记录随机数生成器的状态。

- (e) 启动随机数生成器：

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

随机数生成器的状态会被储存在 `state` 数组中。每一个线程需要各自保有这个变量，因为每个线程中所产生的数值与跳离的位置都不同。除此之外，各线程必须使用相同的种子初始化 `state` 数组 (也就是他们都从相同的起点位置开始)。而并不是每个线程各自调用 `nag_rand_init_repeatabe (g05kfc)` 函数，它将会在进行并行区块前被调用一次，然后每一个线程透过 `memcpy` 保有各自的一份。

- (f) 每个线程向前跳跃 `tskip`：

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

- (g) 自均匀分配中计算取得随机数值，总数量为 `tnpaths`：

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

- (h) 当得到计算的数值后，我们可以使用它来做加总的计算：

```
for (i = 0; i < tnpaths; i++)
```

```
    sumRandom += x[i];
```

由于我们一开始在并行区域以 `reduction` 指令声明 `sumRandom` 变量，个别线程所得到的加总最后将会由编译器负责。

- (i) 最后，当要离开并行区域时，我们需要释放所有并行区域的内存，也就是程序中的 `x` 与 `state` 变量。

3. 非均匀虚拟随机随机数序列

NAG 算法库中有 30 多种不同的分配函数，在 CPseudo_OpenMP.c 与 FPseudo_OpenMP.f90 程序中使用均匀分配函数。虽然我们也可以在多线程安全的情况下使用其他函数，但是我们无法保证能得到相同的统计特性，也不能确保在不同数量的线程环境中是能得到相同的随机数。这是因为非均匀分配的原因。

所有非均匀分配产生器都是以均匀分配产生器开始的，并从所需要的分配中取得数值。NAG 算法库使用三种基本方法：transformation、table search 与 rejection 方法 (请参阅 g05 章节说明)。最佳的情况是能够透过均匀分配一对一的对应到所要分配的分布值。基于反向累积分配函数的状况下，大部分都可采用 table search 与 transformation 方法。在 NAG C 算法库第 8 版中，下面函数的输入参数都能有一对一的关系，所以可以透过以上第二点所说的方式搭配 OpenMP 使用：

uniform (nag_rand_basic (g05sac)), exponential (nag_rand_exp (g05sfc)), Normal (nag_rand_normal (g05skc)), logistic (nag_rand_logistic (g05slc)), log-normal (nag_rand_lognormal (g05smc)), triangular (nag_rand_triangular (g05spc)), Weibull (nag_rand_weibull (g05ssc)), binomial (nag_rand_binomial (g05tac)), logical (nag_rand_logical (g05tbc)), geometric (nag_rand_geom (g05tcc)), discrete (nag_rand_gen_discrete (g05tdc)), hypergeometric (nag_rand_hypergeometric (g05tec)), negative binomial (nag_rand_neg_bin (g05thc)), Poisson (nag_rand_poisson (g05tjc)) 与 discrete uniform (nag_rand_discrete_uniform (g05tlc)) 等。

在 NAG Fortran 算法库第 22 版中也有相同对应的函数。若您想得知以上对应的 Fortran 函数，您只需将以上较短函数名改为大写，并将最后一个字符改为 F 即可。例如：Fortran 算法库对应于 NAG C 的 nag_rand_basic (g05sac) 函数是 G05SAF。

对于那些没有相对应一对一关系的函数，也就是其他的虚拟随机随机数生成器函数，将无法在不同数量的线程环境中取得一致的随机数序列。也因此，无法保证能够取得一致的统计特征。这是因为唯一能处理这些函数的方式，是需要保证每个线程都能跳的“足够远”，以确保自均匀分配得到的值都不会被重复使用到，但是我们无从说明“足够远”是多远。然而，若函数能“跳”得更远，重复使用到的一样数值的可能性就越低，例如，若每次在均匀分配产生器中，让每个线程跳过 100 个位置一定远比让每个线程跳过 2 个位置得到相同值的机率要低多了。

4. 准随机随机数序列示例

有关准随机随机数与以上说明的虚拟随机例子相类似，我们不再提供详细说明。两者之间主要的差异在于准随机的起始函数 nag_quasi_init (g05ylc) 并不需提供 state 数组，但需要有 iref 数组，其扮演相同的角色，所以也以相同的方式处理 (也就是每个线程必须自己保留一份)。其他不同处是准随机的起始函数有一个 skip 参数 (iskip)，并不需要再透过调用另一个函数的参数 (tskip) 往前跳跃到不同的序列中。

Scrambled quasi-random 结合了其他两种的方法，所以必须同时起始虚拟随机与准随机产生器。

准随机的两个分配：Normal (nag_quasi_rand_normal (g05yjc)) 与 log-normal (nag_quasi_rand_lognormal (g05ykc)) 都采用 inverse CDF 方法，因此与均匀分配也会有一对

一的关系，所以在 CSobol_OpenMP.c 与 FSobol_OpenMP.f90 程序中会以相同于均匀分布的方式进行处理。

5. 编译示例程序

本文的程序可以透过 [GNU 编译器](#) 进行编译，以下是编译命令：

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
```

其中 [INSTALL_DIR] 表示 NAG 算法库所安装路径。若您想要详细了解 NAG 算法库不同版本的编译方式，您可以参考 <http://tw.nag-gc.com/numeric/CL/CLinuns.asp> 与 <http://tw.nag-gc.com/numeric/FL/FLinuns.asp>。其他不同编译器中编译 OpenMP 程序的方法，请您参考各编译的编译说明。

6. NAG 多核算法库

除了串行算法库外，NAG 也提供 [多核算法库](#)。多核算法库中包含了所有 NAG Fortran 算法库中的函数，具有相同的调用接口，但是其中大部分的函数都经过多核环境的并行程序规划与设计，能在多核的环境中拥有更佳的性能。本文撰写时，第 22 版的算法库已经具备准随机随机数生成器的优化版本，第 23 版中将会包含许多优化的虚拟随机的函数，便不须如上文所提的需要自行编写并行部分。