Calling the NAG Pseudo and Quasi Random Number Generators From a Multi-Threaded Environment

1 Introduction

In this article, and the associated example programs, we will show how to call the NAG random number generators within a multi-threaded enviroment. The examples are written making use of OpenMP (http://openmp.org/), however the basic structure of the NAG calls will be the same irrespective of the threading mechanism used. Some OpenMP commands and pragmas are briefly described in this document, however additional information is available on the OpenMP website, alternatively NAG offer training courses in OpenMP, details of which can be obtained from mailto:nagmarketing@nag.co.uk.

To start with we give a (very) brief introduction to pseudo and quasi-random numbers. A sequence of pseudorandom numbers is a sequence of numbers, generated in some systematic way, that are independent and statistically indistinguishable from a truly random sequence. Quasi-random (or low discrepancy) numbers are designed to give a more even distribution in multidimensional space and are therefore not independent and can be easily distinguised from a truly random sequence. Pseudorandom numbers tend to be used where the independence and "randomness" of the sequence is important often because these properties are integral to the theory underyling the methodology being used. On the other hand the structure of quasi-random numbers often makes them more efficient than pseudorandom numbers when using multidimensional Monte-Carlo methods. The NAG libraries include a third type of sequence, scrambled quasi-random numbers, which is a hybrid of the two and attempts to keep the structure of quasi-random numbers whilst adding some of the randomness of pseudorandom numbers (see the g05 Chapter Introduction and the references given there in for more detail).

Pseudorandom generators are deterministic, that is they are generated in some deterministic way and so always generate the same sequence of values and this sequence is circular. Letting x_i denote the ith generated value (from some arbitary starting point), then the generated sequence is given by

$$x_1, x_2, \ldots, x_p, x_1, x_2, \ldots$$

where p indicates the length of the sequence before it repeats itself (usually called the period). The starting point is controlled by the seed used to initialise the random number generator. The quasi-random generators in the NAG Library are similar, except they have no seed so will always start in the same place. In Mark 9 of the C Library (and Mark 22 of the Fortran Library) routines were added to rapidly advance (skip-ahead) the pseudorandom number generators. This means it is possible to go from x_i to x_{i+j} for $j > 0$ without passing through the intervening points. This functionality allows the routines to be safely called in a multi-threaded environment in such a way that the same sequence of numbers is generated irrespective of the number of threads being used. The same can be done for the quasi-random number generators by making use of the skip ahead parameter present in the initialization routines.

2 Walk Through of an Example Program

Six different example programs are included with this article, and can be downloaded from here. Three are C example programs and call routines from the NAG C Library, the first calls a pseudorandom generator (a uniform distribution), the second calls a quasi-random general (the Sobol generator) and the last generates a scrambled quasi-random sequence (also based on the Sobol generator). The other three programs do the same, but are written in Fortran and make use of the NAG Fortran Library. The application used in these examples, that is the way that the generated sequence of numbers is used, is trivial and has no practical purpose as it only involves summing the sequence (or in the case of the quasi-random generators the summing of the values from the first dimension). However, this is sufficient to show how these routines can be called and to confirm that the routines produce the same result (sum) irrespective of the number of threads used. This last statement needs to come with a partial rider, whilst the sum will be identical in most cases it is possible that there may be slight variations on some implementations due to the order in which the generated sequence is summed.

The rest of this section will go through, in some detail, the first of the C example programs (`CPseudo_OpenMP.c`), which is concerned with calling a pseudorandom generator. The remaining five

example programs follow a similar structure and are heavily commented so hopefully the following descriptions should be sufficient to explain what is going on in those as well.

The program `CPseudo_OpenMP.c` is structured as follows:

(a) Start of the serial part of the code. In this example this is where we read in the user supplied parameters: `npaths`, `iskip` and `seed`, and calculate the length of the **state** array. This could have been set to 29 as we are using the Wichmann Hill II generator, but instead we are using the facility of nag_rand_init_repeatable (g05kfc) to return the required length for us by calling it with `lstate = -1`.

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

After this call `lstate` will hold the required value.

(b) Start a parallel section of code:

```
#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)
```

this pragma informs the compiler that the next block of code should be executed in parallel with each thread executing the same code. It also tells the compiler that all the variables listed in the parenthesis after the `private` statement, that is `ThreadNo`, `tskip` etc. are private and hence each thread should maintain their own copy. Similarly each variable listed in parenthesis after the `shared` statement can be shared between the threads and hence only a single copy will exist. Finally the `reduction` statement indicates that each thread will have its own copy of the variable `sumRandom` and at the end of this parallel block each of those copies will be added together.

(c) Calculate the number of paths that each thread needs to produce (`tnpaths`) and the amount of the sequence that needs to be skipped to take the thread to its starting position (`tskip`) by getting the number of available threads:

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

getting the thread ID of the current thread (this will be 0 for the first thread, 1 for the second etc.):

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

calculating `tnpaths` and `tskip` as:

```
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;
```

Calculating `tnpaths` in this way splits the work as evenly as possible across all the threads. For example, if we had 5 threads and want to generate 104 values, the first four threads would each generate 21 values (i.e., `tnpaths = 21`) and the last thread would generate the remaining 20 values. The skip for the first thread would be zero, the skip for the second thread would be 21 (i.e., `tskip = 21`) as we need to skip over the 21 values generated by the first thread. Similarly, the skips for the third, fourth and fifth threads would be 42, 63 and 84 respectively. The last line `tskip += iskip;` allows for the use of an initial skip. So, for example, if we had previously generated and used 1000 values in an earlier call to the NAG generators, we would set `iskip = 1000` so as not to reuse the previously generated values.

(d) Because each thread requires its own copy of the NAG C Library error handling structure (`NagError`), prior to calling any of the C Library routines inside the parallel region it is important to initialise the error handling structure

```
INIT_FAIL(fail);
```

at this point we also need to allocate memory for any private arrays, in this case `x`, which will hold the generated values, and `state` which will hold the state of the random number generators.

(e) Initialise the random number generators:

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

The current state of the pseudorandom number generators is stored in the array **state**. Each thread requires its own copy of this array because as each number is generated or skipped the array will change. In addition, each thread must have its state array initialised using the same seed (so that they all start at the same place). Rather than having each thread call nag_rand_init_repeatable (g05kfc) it is also possible to call this routine once, prior to starting the parallel section and then have each thread make a copy of the state array effectively replacing the calls to nag_rand_init_repeatable (g05kfc) in the parallel blocks with calls to `memcpy`.

(f) Skip ahead each thread `tskip` places:

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

(g) Generate the `tnpaths` values from a uniform distribution:

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

(h) Once the values have been generated we can use them in any way we want, in this example we are just summing them:

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

Because we declared `sumRandom` in the `reduction` directive when initialising the parallel region the summing of the individual contributions from each thread will be taken care of by the compiler.

(i) Finally, before exiting the parallel section of the code we need to free any memory allocated in the parallel section, which in this example is just `x` and `state`.

3 Non-Uniform Pseudorandom Sequences

The NAG libraries have routines for generating values from over 30 different distributions as well as the uniform distribution used in `CPseudo_OpenMP.c` and `FPseudo_OpenMP.f90`. Whilst it is possible to use all of these routines in a thread safe manner it is not always possible to do so in such a way that you can guarantee the statistical properties or ensure that the same sequence of values is produced irrespective of the number of threads. This is because of the way values from non-uniform distributions are generated.

All non-uniform distribution generators start with a uniform generator and use values from that generator to obtain values from the required distribution. The NAG Library uses three basic methods for this: transformation, table search and rejection methods (see the g05 Chapter Introduction). In the best case scenario one value from the distribution of interest is generated from one uniform value. This happens most often for the table search method and the transformation method when it is based on the inverse of the cumulative distribution function. As of Mark 8 of the C Library the following routines have a one-to-one relationship for all values of the input parameters, and so can be called in an identical manner as demonstrated in the example program described in Section 2:

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)) and the discrete uniform (nag_rand_discrete_uniform (g05tlc)).

The equivalent routines in the Mark 22 of the Fortran Library also have a one-to-one relationship. To get the equivalent Fortran routines to those named above, take the short name, change it to upper case and replace the last letter with an F, i.e., the Fortran Library equivalent to nag_rand_basic (g05sac) which has the short name `g05sac` is `G05SAF`.

For routines that do not have a one-to-one relationship, i.e., all remaining pseudorandom number generation routines, it is not possible to obtain a sequence that is identical irrespective of the number of

threads being used. In, addition it is impossible to guarantee the properties of the sequence. This is because the only way of handling these routines is to make each thread skip ahead "sufficiently far" to ensure that no values from the underlying uniform distribution are reused, but it is impossible to say how far "sufficiently far" is. However, the further you skip ahead the less likely you are to be reusing old values, for example you would be less likely to use a value from the uniform generator multiple times if you had each thread skip 100 places per generated value compared to having each thread skip 2 places per generated value.

4 Quasi-Random Sequence Examples

The example program for the quasi-random sequence is very similar to that of the pseudorandom example detailed above, therefore we will not go through it in detail. The main differences between the two is that the quasi-random initialisation routine nag_quasi_init (g05ylc) does not have the **state** array, rather it has the **iref** array which plays much the same role and is therefore handled in a similar manner (i.e., each thread must have its own copy). The other difference is that the quasi-random initialisation routines have a skip parameter in their interface (**iskip**), so rather than having to call a separate routine to skip the sequence ahead, this parameter is set to the skip value (`tskip`).

The example program for the scrambled quasi-random sequence is an amalgamation of the other two, as both the pseudo and quasi-random generators must be initialised.

Both of the two distributional generators for quasi-random sequences, the Normal (nag_quasi_rand_normal (g05yjc)) and log-normal (nag_quasi_rand_lognormal (g05ykc)) are based on the inverse CDF method and hence have a one-to-one relationship with the underlying uniform generator. They can therefore be treated in the same manner as the uniform generator in `CSobol_OpenMP.c` and `FSobol_OpenMP.f90`.

5 Compiling the Examples

The example programs can be compiled with the GNU compiler collection (http://gcc.gnu.org/) using the following commands:

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
```

where `[INSTALL_DIR]` is the name of the directory your version of the NAG Library is installed in. For additional information of linking programs to the NAG Library please see the user notes for your particular implementation, which are shipped with the library or available from http://www.nag.co.uk/numeric/CL/CLinuns.asp and http://www.nag.co.uk/numeric/FL/FLinuns.asp. For examples of compiling OpenMP programs using other compilers please refer to the documentation of the compiler in question.

6 NAG Library for SMP and Multicore

As well as its serial libraries, NAG offers a library specifically for SMP and Multicore (http://www.nag.co.uk/numeric/FL/FSdocumentation.asp). This SMP library contains all of the routines found in the NAG Fortran Library, with the same interfaces, but a number of the routines have been tuned, that is parallelized or otherwise optimized to give improved performance over the equivalent routine in the NAG Fortran Library when run on machines with multiple cores. At the time of writing the latest release of this library, Mark 22, has tuned versions of the quasi-random generators. Later marks will contain tuned versions of the pseudorandom generators, including routines for distributions that do not use a one-to-one mapping and so can not be easily parallelized using the method described in Section 3.