Chapter 13

Random Number Generators

This chapter describes the CM Fortran interface for the CMSSL random number generators (RNGs). One section is devoted to each of the following topics:

- introduction
- state tables
- safety checkpointing
- alternate stream checkpointing
- Fast RNG
- VP RNG
- references

Man pages for the RNG routines follow these sections.

13.1 Introduction

CMSSL includes two RNGs:

- the Fast RNG
- the VP RNG

These operations use a lagged-Fibonacci algorithm.

Copyright © 1994 Thinking Machines Corporation

13.1.1 The Fast RNG and the VP RNG Compared

To construct pseudo-random values, the CMSSL random number generators use *state tables*. The difference between the Fast RNG and the VP RNG lies in the allocation of their state table arrays, as follows:

- The Fast RNG operation stores one state table per processing element.
- The VP RNG operation stores one state table per destination array element.

The Fast RNG thus consumes substantially less processing element memory than the VP RNG. The VP RNG can produce identical results on CM partitions of different sizes. Also, the VP RNG is slightly faster than the Fast RNG, albeit at a potentially high cost in memory.

The VP RNG, which mimics the Fast RNG, is generally used if access to a CM partition of a specific size is not guaranteed. For instance, if you are using a 16-processing-element partition while developing an application that will ultimately run on a 64-processing-element partition, use the VP RNG to produce the same results that the Fast RNG will produce when the application is finally run on the larger partition. The VP RNG produces the same result for each array element, regardless of partition size or array shape — provided that the same *total* number of array elements is maintained and that the same seed is used.

NOTE

The Fast RNG is sensitive to the CM Fortran **–noaxisreorder** switch; that is, in general it generates different numbers under **–noaxisreorder** than under **–axisreorder**, for a fixed element of a multidimensional destination array. To obtain the same results under **–noaxisreorder** as under **–axisreorder**, you must either ensure that the array layout is the same in both cases, or use the VP RNG.

Copyright © 1994 Thinking Machines Corporation

13.1.2 The RNG Routines

Most applications that use the CMSSL RNG simply require you to call the Fast or VP RNG and then deallocate the state table. The following operations provide this functionality:

initialize_fast_rng
fast_rng
deallocate_fast_rng

initialize_vp_rng
vp_rng
deallocate_vp_rng

Note that explicitly initializing these RNGs is only necessary if the default table parameters are not suitable for your application. Section 13.2 discusses the state table parameters.

To guard against the effects of forced interruption, or to use more than one stream of random values, some applications require *checkpointing*. Checkpointing is the process of recording state information at a specific point in the random number stream generation. Later, the stream generation can be continued from the checkpoint.

In s*afety checkpointing*, you save the RNG state to a file in case of forced interruption — for instance, during long periods of application execution. If an interruption occurs, you can restore the state and continue processing. Safety checkpointing is accomplished with the following routines:

save_fast_rng_temps
restore_fast_rng_temps

save_vp_rng_temps
restore_vp_rng_temps

Copyright © 1994 Thinking Machines Corporation

NOTE

The save and restore routines do not work with the nodal CMSSL library.

In *alternate stream checkpointing*, you save the RNG states associated with two distinct streams of random numbers in order to switch back and forth between the streams. Alternate stream checkpointing is accomplished with the following operations:

fast_rng_state_field
fast_rng_residue
reinitialize_fast_rng

vp_rng_state_field
vp_rng_residue
reinitialize_vp_rng

See Sections 13.3 and 13.4 for detailed descriptions of safety checkpointing and alternate stream checkpointing, respectively.

13.1.3 Implementation

The lagged-Fibonacci algorithm used by both CMSSL random number generators is widely used to produce a uniform distribution of random values. This implementation has been subjected to a battery of statistical tests, both on the stream of values within each processing element and for cross-processing-element correlation. The only test that the CMSSL RNGs fail is the Birthday Spacings Test, as predicted in reference 2 listed in Section 13.7. Despite this failure, these lagged-Fibonacci RNGs are recommended for the most rigorous applications such as Monte Carlo simulations of lattice gases.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

13.2 State Tables

If you do not initialize the Fast RNG or VP RNG explicitly, the CM Fortran interface uses default parameters to initialize the RNG automatically the first time you call it. If you want to use non-default state table parameters, you must initialize the VP or Fast RNG explicitly.

13.2.1 Fast RNG State Tables

The **initialize_fast_rng** routine allocates the state tables for the Fast RNG and initializes them with values generated internally by the CM Fortran utility **CMF_RANDOM**. The **initialize_fast_rng** routine does not have a *seed* argument. Therefore, if you want to provide a seed, you must call the **CMF_RANDOMIZE** utility before calling **initialize_fast_rng**. (**CMF_RANDOM** and **CMF_RANDOMIZE** are described in the CM Fortran documentation set.)

NOTE

CMF_RANDOMIZE itself generates a call internally to **initialize_fast_rng** when CMSSL is linked in. Therefore, if you call **CMF_RANDOMIZE** prior to an explicit call to **initialize_fast_rng**, **initialize_fast_rng** will return an *ier* code of −1. This code, which indicates that a prior initialization was overwritten, is informational only, and does not indicate an error.

Figure 39 shows a Fast RNG state table. In the Fast RNG, one state table is associated with each processing element; from this table, random values are constructed for each subgrid element. (The *subgrid* associated with a processing element is the set of array elements residing in that processing element's memory.) Thus, each time you call **fast_rng**, the number of pseudo-random values produced by each state table is equal to the subgrid size.

Copyright © 1994 Thinking Machines Corporation

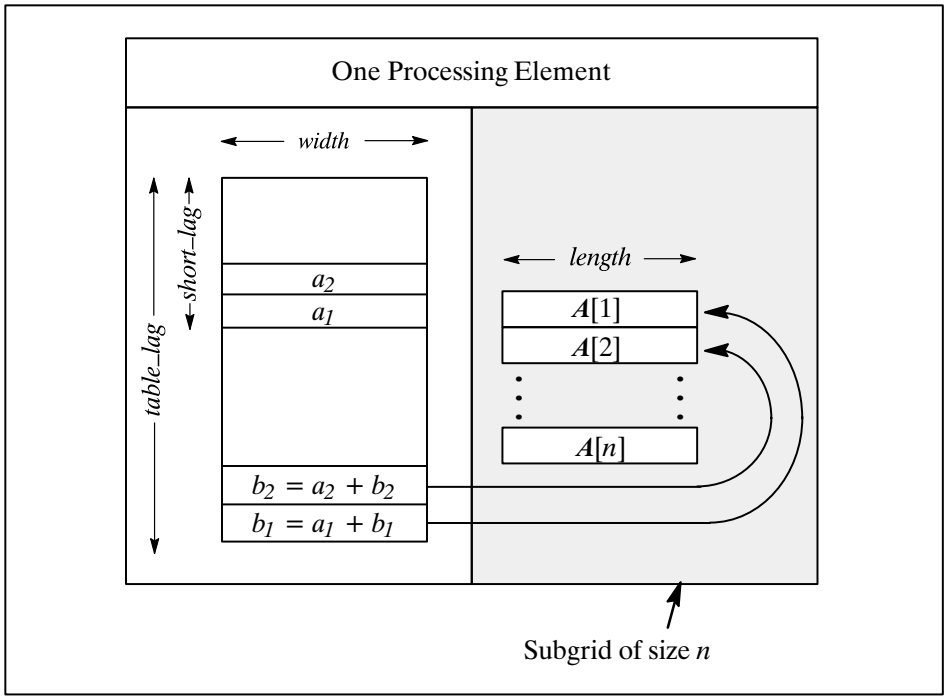

Figure 39. Fast RNG state table.

If you do not explicitly initialize **CMF_RANDOM** and the Fast RNG, initialization occurs automatically when you first call **fast_rng**. The Fast RNG selects a dynamic seed value based on the system time for the **CMF_RANDOM** seed. As a result, with default initialization, different results are obtained in different runs. Explicit initialization is required to obtain identical results.

13.2.2 VP RNG State Tables

The **initialize_vp_rng** routine allocates the state tables for the VP RNG and initializes them with values generated internally by **CMF_RANDOM**. The application provides a seed for **CMF_RANDOM** in the **initialize_vp_rng** *seed* argument. Figure 40, below, shows one VP RNG state table. In contrast to the Fast RNG state table, each subgrid element has its own state table. Each time you call **vp_rng**, each state table produces only one value.

Copyright © 1994 Thinking Machines Corporation

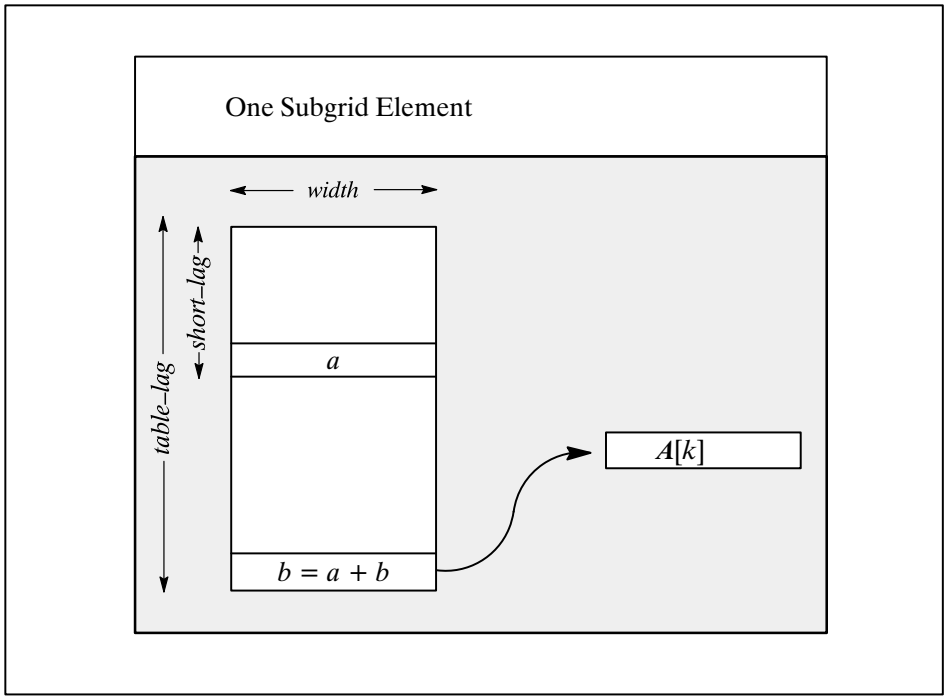

Figure 40. VP RNG state table.

If you do not call **initialize_vp_rng** explicitly, **vp_rng** selects a dynamic seed value based on the system time. As a result, with default initialization, different results are obtained in different runs. Explicit initialization is required to obtain identical results.

13.2.3 State Table Parameters

For either CMSSL RNG, each state table has *table_lag* elements and a width of *width* bits. The state table is treated as a circular buffer. Internal variables are initialized so that they point into the table at offsets of *short_lag* and *table_lag* elements. The operation of generating a pseudo-random value (termed *stepping* the RNG) consists of adding the element at the *short_lag* pointer to the element at the *table_lag* pointer. This sum is shown as $b = a + b$ in the figures above, where the element shown as b is overwritten by the sum. All or a portion of the bits in element b are used to construct a pseudo-random value, which is then copied to the destination. The pointers are then either decremented by one element or cycled around the buffer.

Copyright © 1994 Thinking Machines Corporation

The state table parameters are critical to proper CMSSL RNG operation. The defaults used for automatic initialization are (17, 5) for *table_lag* and *short_lag*. For information about the defaults for *width*, refer to the man page for Fast RNG or VP RNG. Also, refer to the man pages for specific requirements and recommendations regarding *table_lag*, *short_lag*, and *width*.

The *period* of a random number generator is the number of discrete pseudo-random values it can generate before repeating the original stream. For applications that call a CMSSL RNG many times, it is important to make sure the period is long enough. The period should be greater than the total number of values produced by any one state table during all calls to the RNG. Otherwise, it may be possible to detect correlations within the stream produced by one processing element or between streams in different processing elements. See the man page for Fast RNG or VP RNG, later in this chapter, for information on calculating an RNG period from different table parameters.

13.2.4 Need for Deallocation

Both the Fast RNG and the VP RNG allocate state tables as heap memory in the processing elements; until explicitly deallocated, the tables occupy processing element memory. The operations **deallocate_fast_rng** and **deallocate_vp_rng** deallocate the memory allocated for the state tables. It is important that you use the deallocation routines, especially for the VP RNG, in which the state tables can occupy a significant amount of processing element storage space. (In the Fast RNG, the state table size is only on the order of *table_lag*.)

13.2.5 Parameters Saved During Checkpointing

Figure 41 shows how a CMSSL RNG state table is stepped. Cycling through the circular buffer requires *table_lag* steps. During one cycle, each element is used once as the a value and once as the b value in the computation $b = a + b$; thus, all element values are modified. Then the cycle repeats using the new values. The *residue* is the number of steps taken so far in a cycle. At any point, if you save the residue and the values in the state table, you can resume stream generation with the next step.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

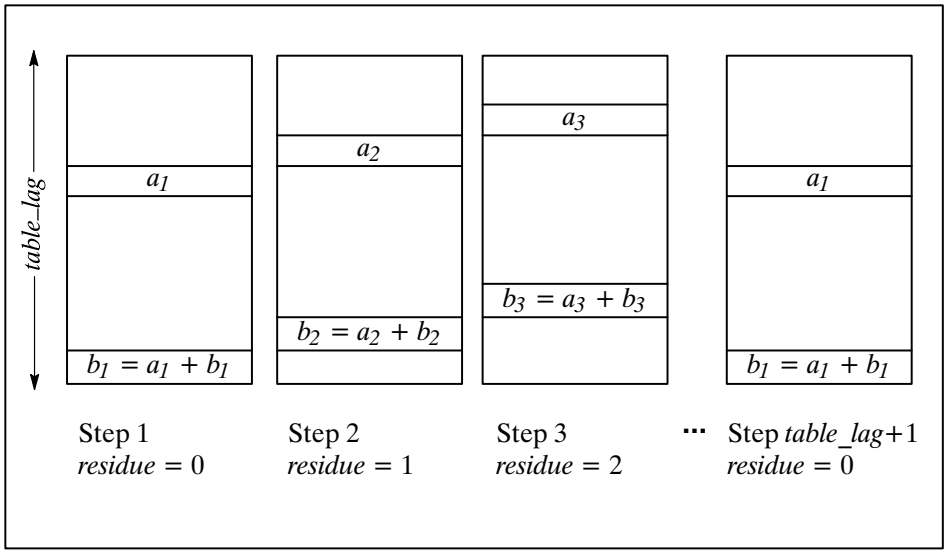

Figure 41. Stepping the state table.

The following routines save or return information for checkpointing:

- The **save_fast_rng_temps** and **save_vp_rng_temps** routines save the contents of the state table array, as well as all state table parameters (including the residue), to a file.
- The **fast_rng_state_field** and **vp_rng_state_field** routines return internal information used to determine the current state of the state table.
- The **fast_rng_residue** and **vp_rng_residue** routines return the residue value.

As explained in Sections 13.3 and 13.4, for safety checkpointing you must copy both the residue value and the contents of the state table array to a file; whereas for alternate-stream checkpointing, you save the information returned by the **fast_rng_state_field** and **fast_rng_residue** (or **vp_rng_state_field** and **vp_rng_residue**) routines, but not the actual contents of the state table, and external storage is not required.

Copyright © 1994 Thinking Machines Corporation

13.3 Safety Checkpointing

If you are running a long application that makes repeated use of one of the CMSSL RNGs, it is a good idea to safety checkpoint. Should a forced interruption occur, the application can be resumed at a checkpoint rather than having to be restarted from the beginning.

To perform safety checkpointing, you must use the save and restore routines:

save_fast_rng_temps	**save_vp_rng_temps**
restore_fast_rng_temps	**restore_vp_rng_temps**

To use this method, insert calls to the save routine periodically among your RNG calls. If a forced interruption occurs, call the restore routine, which restores the state table to exactly the values it had when the state was saved. No reinitialization is required; you can call the RNG again immediately after restoring the state. Figure 42 is a flowchart for safety checkpointing.

The Fast RNG generates different random number sequences for destination arrays on different CM partition sizes. Therefore, if you are using the Fast RNG, the partition size must be the same when you restore the Fast RNG state as when you saved it.

In contrast, the VP RNG state table reflects the underlying geometry of the destination array originally used to initialize the table. Therefore, the dimensions of the destination array must be the same when you restore the VP RNG state as when you saved it.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

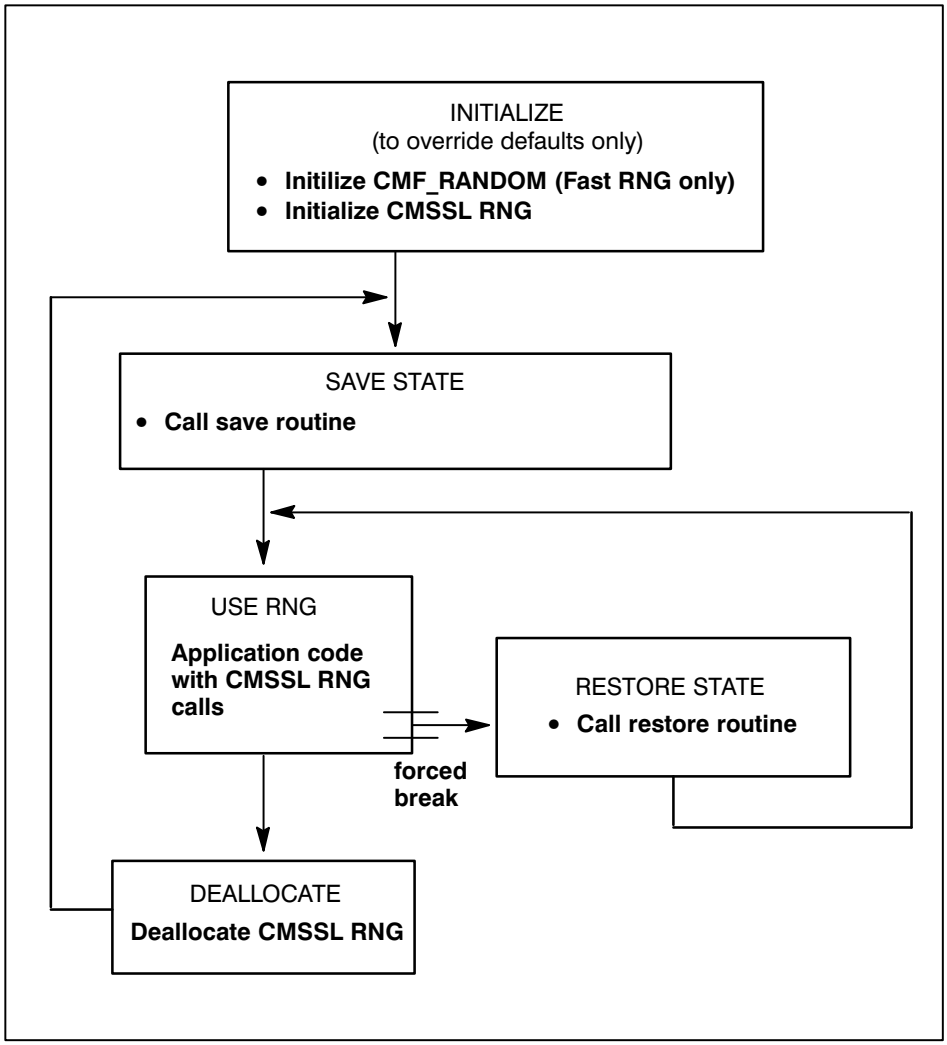

Figure 42. Safety checkpointing.

13.4 Alternate-Stream Checkpointing

It is possible to switch back and forth between two or more Fast or VP RNG streams by saving the state and reinitializing one RNG after the other. For instance, you can use this strategy to make separate random picks from two databases that need to be merged or compared in some way. You can also

Copyright © 1994 Thinking Machines Corporation

use it to increase efficiency when one stream of random numbers requires a wide state table and another can use a narrower state table.

If two or more VP RNG calls use destination arrays with different shapes, axis orderings, or axis weights, you *must* use alternate-stream checkpointing.

Figure 43 is a flowchart for alternate-stream checkpointing using two random number streams. The first RNG is initialized and its state is saved. Then the second RNG is initialized and its state is saved. To alternate between streams, first one and then the other RNG is looped through the reinitialization, use, and state-saving phases.

In contrast to safety checkpointing, the state-saving phase of alternate-stream checkpointing does not require saving a copy of the state table contents. Two state tables are allocated during initialization; their internal states are used repeatedly during reinitialization; and finally both are deallocated.

Copyright © 1994 Thinking Machines Corporation

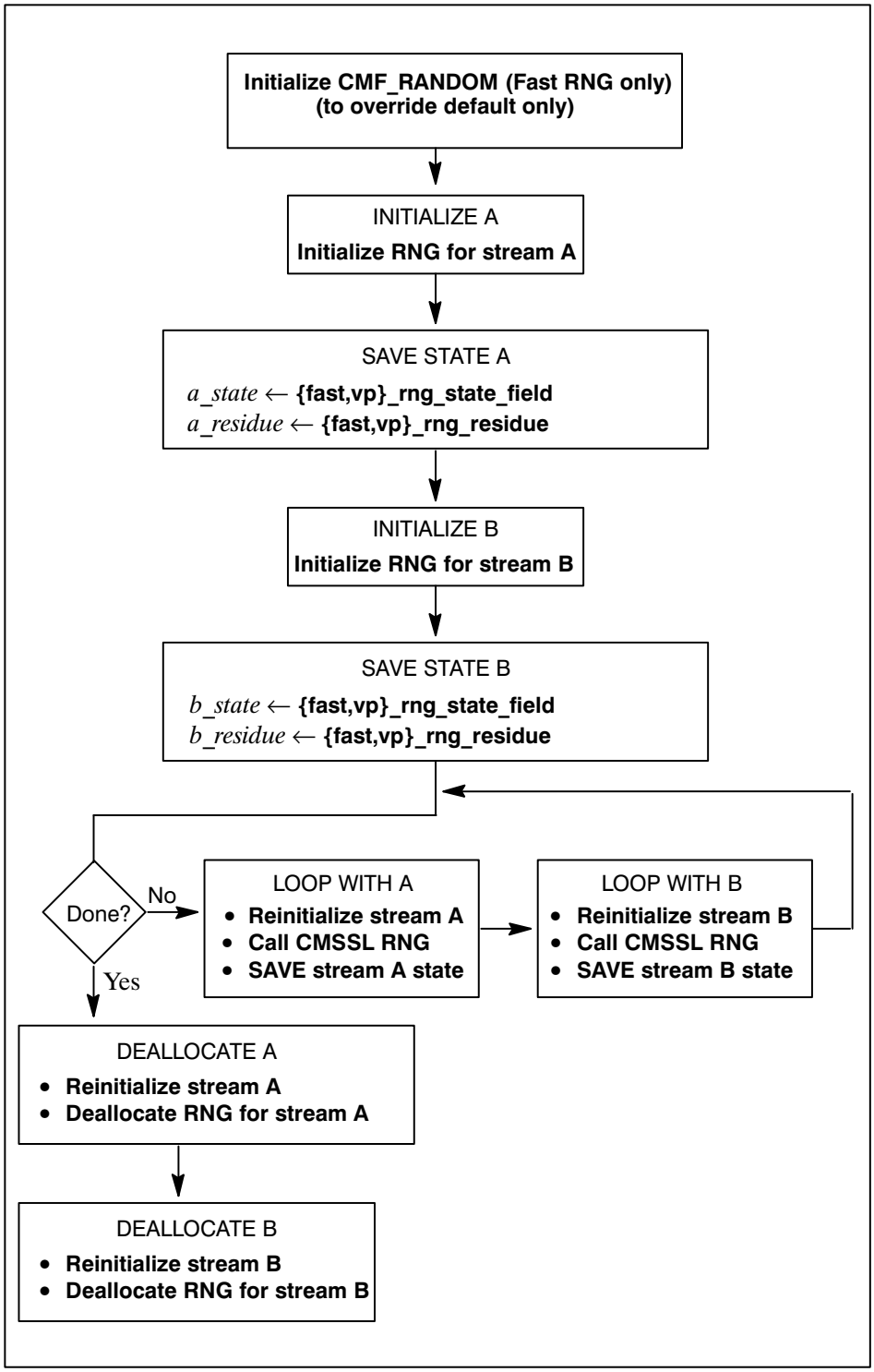

Figure 43. Alternate-stream checkpointing.

Copyright © 1994 Thinking Machines Corporation

13.5 Fast RNG

The Fast RNG uses a lagged-Fibonacci algorithm to generate pseudo-random numbers and store them in a destination array. One state table per processing element is stored. The man page that follows provides details.

Copyright © 1994 Thinking Machines Corporation

Fast RNG

The CMSSL Fast RNG routines use a lagged-Fibonacci algorithm to generate pseudo-random numbers and store them in a destination array. Results may be integer values subject to a limit, or real values between 0.0 and 1.0.

SYNTAX

initialize_fast_rng (*table_lag, short_lag, width, ier*)

fast_rng (*A, limit, ier)*

save_fast_rng_temps (*unit, iostat, ier*)

restore_fast_rng_temps (*unit, iostat, ier*)

state_table = **fast_rng_state_field** (*ier*)

residue = **fast_rng_residue** (*ier*)

reinitialize_fast_rng (*table_lag, short_lag, width, state_table, residue, ier*)

deallocate_fast_rng (*ier)*

ARGUMENTS

table_lag — Scalar integer specifying the length of the state table. The default value for automatic initialization is 17. When you call **reinitialize_fast_rng**, supply the same value that was used in the original **initialize_fast_rng** call.

short_lag — Scalar integer used as an offset into the state table. Must be less than *table_lag*. The default value for automatic initialization is 5. When you call **reinitialize_fast_rng**, supply the same value that was used in the original **initialize_fast_rng** call.

width — Scalar integer specifying the width of the state table in bits. Regardless of the value you supply, the RNG is *always* initialized with the following *width* values:

- 32, if ***A*** is integer and *limit* is 0.

Copyright © 1994 Thinking Machines Corporation

- 64, if ***A*** is integer and *limit* is not 0.
- 32, if ***A*** is single-precision real (unless you explicitly specify a *width* of 64, in which case the RNG is initialized with a *width* of 64).
- 64, if ***A*** is double-precision real.

A CM array of type real or integer. Upon successful completion of **fast_rng**, this array is overwritten with the RNG results.

limit Scalar integer variable. Ignored if ***A*** is real. If ***A*** is a 64-bit integer, *limit* must also be a 64-bit integer (even if it is 0). If ***A*** is a 32-bit integer, *limit* must also be a 32-bit integer. In either case, *limit* must be ≥ 0; it is the exclusive positive upper bound on the pseudo-random values generated. A *limit* value of 0 is interpreted as allowing any 32-bit pattern, so that the full range of positive and negative integers is permitted.

unit Scalar integer. Valid unit number associated with the file to or from which the array ***A*** is to be written or read. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a device or socket with a unit number). The **save_fast_rng_temps** and **restore_fast_rng_temps** routines write and read data using **CMF_ARRAY_TO_FILE_SO** and **CMF_ARRAY_FROM_FILE_SO**, respectively. You must rewind the file before calling **restore_fast_rng_temps**.

iostat Scalar integer variable. Upon return, contains the status of the I/O operation. If *ier* = 0, *iostat* contains the number of bytes written or read. For the meanings of other *iostat* codes (returned if *ier* = −100), refer to the descriptions of **CMF_ARRAY_TO_FILE_SO** (for **save_fast_rng_temps**) and **CMF_ARRAY_FROM_FILE_SO** (for **restore_fast_rng_temps**) in the CM Fortran documentation set.

state_table Scalar integer variable returned by **fast_rng_state_field**; contains internal information used to determine the current state of the state table. When you call **reinitialize_fast_rng**, supply the value returned by a previous call to **fast_rng_state_field**.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

residue Scalar integer variable returned by **fast_rng_residue**. Contains the checkpointed value of the state table residue. This value indicates how many times the state has been stepped, modulo the *table_lag*. When you call **reinitialize_fast_rng**, supply the value returned by a previous call to **fast_rng_residue**.

ier Scalar integer variable. Error code. Upon successful return from **initialize_fast_rng**, *ier* contains −1 if this initialization overwrote a previous initialization, or 0 if it did not. (Note: if you called **CMF_RANDOMIZE** before calling **initialize_fast_rng**, **initialize_fast_rng** returns *ier* = −1, because **CMF_RANDOMIZE** itself calls **initialize_fast_rng** internally.)

Upon successful return from **fast_rng**, *ier* contains −2 if default initialization was used, or 0 if default initialization was not used.

Upon return from **save_fast_rng_temps** or **restore_fast_rng_temps**, *ier* contains 0 if the routine was successful, or −100 if an I/O error was encountered. If *ier* = −100, check the returned value of *iostat* for more information.

Upon successful return from **fast_rng_state_field**, *ier* contains −2 if default initialization was used, or 0 if default initialization was not used.

Upon return from **fast_rng_residue**, *ier* contains 0 if the routine succeeded.

Upon successful return from **reinitialize_fast_rng**, *ier* contains −1 if this initialization overwrote a previous initialization, or 0 if it did not.

Upon return from **deallocate_fast_rng**, *ier* contains 0 if the routine succeeded or −1 if there is no previous state to deallocate.

RETURNED VALUE

state_table Scalar integer variable containing internal information used to determine the current state of the state table.

residue Scalar integer variable containing the checkpointed value of the state table residue. This value indicates how many times the state has been stepped, modulo the *table_lag*.

Copyright © 1994 Thinking Machines Corporation

DESCRIPTION

The Fast RNG is much faster than **CMF_RANDOM** and uses far less processing element memory than the VP RNG.

Usage. Follow these steps to use the Fast RNG:

1. Call **initialize_fast_rng** (optional). This step is required only if the default initialization parameters are not suitable for your application.

2. Call **fast_rng** to generate the pseudo-random numbers. You may repeat this step as many times as you wish, but the state table parameters with which the RNG was initialized (explicitly or by default) must be appropriate for each call. If a **fast_rng** call requires different state table parameters, you must initiate a new state table by calling **initialize_fast_rng** with the new parameter values.

3. After all **fast_rng** calls associated with one set of state table parameters have finished, call **deallocate_fast_rng** to deallocate the state table.

To perform safety checkpointing, use the **save_fast_rng_temps** and **restore_fast_rng_temps** routines. To perform alternate stream checkpointing, use the **fast_rng_state_field**, **fast_rng_residue**, and **reinitialize_fast_rng** routines.

Initialization. The **initialize_fast_rng** routine allocates one Fast RNG state table as heap memory in each processing element. The state table is initialized with values generated internally by **CMF_RANDOM**. The initialization routine also initializes internal state, including the state table information referenced by **fast_rng_state_field** and the residue referenced by **fast_rng_residue**.

Separate state tables are allocated in each processing element. The size of the state table in bits per processing element is the product of the *table_lag* and *width* parameters. For example, given a *width* of 32 and the recommended *table_lag* of 17, the state table occupies 544 bits per processing element.

If your application requires a state table array configured differently from the default, you must call **CMF_RANDOMIZE** to initialize **CMF_RANDOM** with a seed, and then call **initialize_fast_rng**, before using **fast_rng**. This is important because **CMF_RANDOM** is used to fill the initial state table for the Fast RNG. Note that if you call **CMF_RANDOMIZE** followed by **initialize_fast_rng**, **initialize_fast_rng** returns $ier = -1$. This code is not an error; it occurs because **CMF_RANDOMIZE** itself calls **initialize_fast_rng** internally.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

If you do not explicitly initialize **CMF_RANDOM** and the Fast RNG, initialization occurs automatically when you first call **fast_rng**. The Fast RNG selects a dynamic seed value based on the system time for the **CMF_RANDOM** seed. The other defaults used for automatic initialization are 17 for *table_lag* and 5 for *short_lag*; for *width* defaults, see the Arguments section above.

For reproducible results, use the same **CMF_RANDOM** seed and the same Fast RNG parameters each time. If you need reproducible results on different partition sizes, use the VP RNG.

The state table parameters have an enormous effect on the results obtained by calls to **fast_rng**. Use the following guidelines for proper state table initialization:

- The *period* of a random number generator is the number of random values it produces before repeating the original stream. To avoid correlation, the period should be greater than the total number of random values produced by any processing element; that is,

  $$desired_period > \text{subgrid size} \times invocations$$

  where *invocations* is the number of times the program calls **fast_rng**.

- The period is very sensitive to the choice of *table_lag* and *short_lag* values. The default pair of values, (17, 5), has been carefully chosen to produce the maximum period for the minimum storage. Other suggested values are (55, 24) and (71, 35). When the lag pairs are properly chosen, the period of the Fast RNG depends exponentially on the state table length (*table_lag*) and on the state table width (*width*), such that

  $$period = (2^{table_lag} - 1) \times 2^{width}$$

  For a discussion on choosing proper lag value pairs, see the book by Knuth referenced in Section 13.7.

- The RNG is *always* initialized with the *width* values shown in the Arguments section.

Random Number Generation. The **fast_rng** routine copies a pseudo-random value, chosen from a uniform distribution, into each element of ***A***. The distribution range in the floating-point case is from 0.0 (inclusive) to 1.0 (exclusive). The distribution range in the integer case with a positive *limit* is from 0 (inclusive) to the specified *limit* (exclusive). The distribution range in the integer case with a 0 *limit* is all integer values from -2^{31} to $2^{31} - 1$ (in other words, any 32-bit pattern).

Copyright © 1994 Thinking Machines Corporation

The Save and Restore Routines. The **save_fast_rng_temps** and **restore_fast_rng_temps** routines provide a mechanism for safety checkpointing, allowing an application to resume processing from a checkpoint after a forced interruption.

To use the save and restore routines, insert calls to the save routine periodically among your RNG calls. If a forced interruption occurs, call the restore routine, which restores the state table to exactly the values it had when the state was saved. (You must rewind the file before calling **restore_fast_rng_temps**.) No reinitialization is required; you can call the RNG again immediately after restoring the state. The partition size must be the same when you restore the fast RNG state as when you saved it.

The State Field, Residue, and Reinitialization Routines. The **fast_rng_state_field**, **fast_rng_residue**, and **reinitialize_fast_rng** routines provide a method for alternate-stream checkpointing, in which the application switches back and forth between two or more Fast RNG streams by saving the state and reinitializing one RNG after the other.

To perform checkpointing using these routines, follow these steps:

1. After the Fast RNG has been initialized either implicitly or explicitly by **initialize_fast_rng**, and after **fast_rng** has been called zero or more times, call **fast_rng_state_field** and **fast_rng_residue**.
2. Save the values of *state_table* and *residue*.
3. To restart a previously checkpointed Fast RNG number stream, call **reinitialize_fast_rng** with the previously saved *state_table* and *residue* values.

The **fast_rng_state_field** routine returns internal information used to determine the current state of the Fast RNG state table; this information was generated previously by an implicit or explicit call to **initialize_fast_rng**. The state table resides in the processing elements; its size is the product of *table_lag* and *width*.

The **fast_rng_residue** routine returns the residue: a count of the number of times that the Fast RNG state has been stepped, modulo the *table_lag*. The residue is the product of the subgrid size and the number of calls to **fast_rng** that have occurred since the last call to **initialize_fast_rng** or **reinitialize_fast_rng**.

The **reinitialize_fast_rng** routine reinitializes the Fast RNG from a previously checkpointed state so that an interrupted computation can be resumed.

Copyright © 1994 Thinking Machines Corporation

Deallocation. The **deallocate_fast_rng** routine deallocates the heap field that has been used to store the state table for the Fast RNG. Call this routine when you are finished with the Fast RNG.

NOTES

No Nodal Library Support for Save and Restore Routines. The save and restore routines do not work with the nodal CMSSL library.

Axis Reordering. The Fast RNG is sensitive to the CM Fortran **–noaxisreorder** switch; that is, in general it generates different numbers under **–noaxisreorder** than under **–axisreorder**, for a fixed element of a multidimensional destination array. To obtain the same results under **–noaxisreorder** as under **–axisreorder**, you must either ensure that the array layout is the same in both cases, or use the VP RNG.

Numerical Performance. The lagged-Fibonacci algorithm implemented by this RNG is widely used to produce a uniform distribution of random values.

For a *width* of 32 bits and using the default *table_lag* and *short_lag* values, (17, 5), the period of the fast RNG is $(2^{17}-1)2^{32} \approx 5.6e15$ bits. By comparison, the period for **CMF_RANDOM** is estimated to be 6.8e10 bits, with greater danger of cross-processing-element correlation.

Running time for the Fast RNG increases with the state table width and the number of bits used. For best results, reduce the table width to the number of bits required and use a *limit* value of 0.

Applications. The Fast RNG should be used in applications such as Monte Carlo simulations where speed is a priority and there is enough room for the state table.

Include the CMSSL Header File. Because the **fast_rng_residue** and **fast_rng_state_field** calls are functions, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

in program units that contain calls to these routines. This file declares the types of the CMSSL functions and symbolic constants.

Reproducible Results. To obtain reproducible results from the Fast RNG, initialize **CMF_RANDOM** with the same seed each time, and call **initialize_fast_rng** explicitly.

Copyright © 1994 Thinking Machines Corporation

In contrast, checkpointing and reinitializing an RNG is used to *continue* random value stream generation from a previous or alternate state.

No Error Checking on Reinitialization. The **reinitialize_fast_rng** routine does not perform error checking on the input parameters. Unpredictible results or halted execution are likely under the following conditions:

- The size of *state_table* is less than (*table_lag* × *width*).
- *residue* is negative or *residue* ≥ *table_lag*.

EXAMPLES

Sample CM Fortran code that uses the Fast RNG routines can be found on-line in the subdirectory

```
random/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

13.6 VP RNG

The VP RNG uses a lagged-Fibonacci algorithm to generate pseudo-random numbers and store them in a destination array. One state table per destination array element is stored. The VP RNG produces the same result for each array element, regardless of partition size or array shape, provided that the same *total* number of array elements is maintained and that the same seed is used.

The man page that follows provides details.

Copyright © 1994 Thinking Machines Corporation

VP RNG

The CMSSL VP RNG routines use a lagged-Fibonacci algorithm to generate pseudo-random numbers and store them in a destination array. Results may be integer values subject to a limit, or real values between 0.0 and 1.0.

SYNTAX

initialize_vp_rng (***A_template***, *table_lag, short_lag, width, seed, ier*)

vp_rng (***A***, *limit, ier*)

save_vp_rng_temps (*A, unit, iostat, ier*)

restore_vp_rng_temps (*A, unit, iostat, ier*)

state_table = **vp_rng_state_field** (***A_template***, *ier*)

residue = **vp_rng_residue** (***A_template***, *ier*)

reinitialize_vp_rng (*table_lag, short_lag, width, state_table, residue, ier*)

deallocate_vp_rng (***A_template***, *ier*)

ARGUMENTS

A_template	CM array of the same shape and size as the ***A*** argument supplied to any **vp_rng** call for this RNG stream. Information about geometry and layout is taken from this argument.
table_lag	Scalar integer specifying the length of the state table. The default value for automatic initialization is 17. When you call **reinitialize_vp_rng**, supply the same value that was used in the original **initialize_vp_rng** call.
short_lag	Scalar integer used as an offset into the state table. Must be less than *table_lag*. The default value for automatic initialization is 5. When you call **reinitialize_vp_rng**, supply the same value that was used in the original **initialize_vp_rng** call.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

width Scalar integer specifying the width of the state table in bits. Regardless of the value you supply, the RNG is *always* initialized with the following *width* values:

- 32, if ***A*** is integer and *limit* is 0.
- 64, if ***A*** is integer and *limit* is not 0.
- 32, if ***A*** is single-precision real (unless you explicitly specify a *width* of 64, in which case the RNG is initialized with a *width* of 64).
- 64, if ***A*** is double-precision real.

seed Scalar integer used to initialize **CMF_RANDOM** for reproducible results. If you do not explicitly initialize the VP RNG, the VP RNG selects a dynamic seed value based on the system time.

A CM array of type real or integer. Upon successful completion of **vp_rng**, this array is overwritten with the RNG results. When you call **save_vp_rng_temps**, ***A*** must contain the array you want to save. Upon return from **restore_vp_rng_temps**, ***A*** contains the array restored from the file.

limit Scalar integer variable. Ignored if ***A*** is real. If ***A*** is a 64-bit integer, *limit* must also be a 64-bit integer (even if it is 0). If ***A*** is a 32-bit integer, *limit* must also be a 32-bit integer. In either case, *limit* must be ≥ 0; it is the exclusive positive upper bound on the pseudo-random values generated. A *limit* value of 0 is interpreted as allowing any 32-bit pattern, so that the full range of positive and negative integers is permitted.

unit Scalar integer. Valid unit number associated with the file to or from which the array ***A*** is to be written or read. Use the CM Fortran utility **CMF_FILE_OPEN** to associate a file with a unit number (or use the equivalent utility to associate a device or socket with a unit number). The **save_vp_rng_temps** and **restore_vp_rng_temps** routines write and read data using **CMF_ARRAY_TO_FILE_SO** and **CMF_ARRAY_FROM_FILE_SO**, respectively. You must rewind the file before calling **restore_vp_rng_temps**.

Copyright © 1994 Thinking Machines Corporation

iostat Scalar integer variable. Upon return, contains the status of the I/O operation. If *ier* = 0, *iostat* contains the number of bytes written or read. For the meanings of other *iostat* codes (returned if *ier* = −100), refer to the descriptions of **CMF_ARRAY_TO_FILE_SO** (for **save_vp_rng_temps**) and **CMF_ARRAY_FROM_FILE_SO** (for **restore_vp_rng_temps**) in the CM Fortran documentation set.

state_table Scalar integer variable returned by **vp_rng_state_field**; contains internal information used to determine the current state of the state table. When you call **reinitialize_vp_rng**, supply the value returned by a previous call to **vp_rng_state_field**.

residue Scalar integer variable returned by **vp_rng_residue**. Contains the checkpointed value of the state table residue. This value indicates how many times the state has been stepped, modulo the *table_lag*. When you call **reinitialize_vp_rng**, supply the value returned by a previous call to **vp_rng_residue**.

ier Scalar integer variable. Error code. Upon successful return from **initialize_vp_rng**, *ier* contains −1 if this initialization overwrote a previous initialization, or 0 if it did not.

Upon successful return from **vp_rng**, *ier* contains −2 if default initialization was used, or 0 if default initialization was not used.

Upon return from **save_vp_rng_temps** or **restore_vp_rng_temps**, *ier* contains 0 if the routine was successful, or −100 if an I/O error was encountered. If *ier* = −100, check the returned value of *iostat* for more information.

Upon successful return from **vp_rng_state_field**, *ier* contains −2 if default initialization was used, or 0 if default initialization was not used.

Upon return from **vp_rng_residue**, *ier* contains 0 if the routine succeeded.

Upon successful return from **reinitialize_vp_rng**, *ier* contains −1 if this initialization overwrote a previous initialization, or 0 if it did not.

Upon return from **deallocate_vp_rng**, *ier* contains 0 if the routine succeeded or −1 if there is no previous state to deallocate.

Copyright © 1994 Thinking Machines Corporation

RETURNED VALUE

state_table Scalar integer variable containing internal information used to determine the current state of the state table.

residue Scalar integer variable containing the checkpointed value of the state table residue. This value indicates how many times the state has been stepped, modulo the *table_lag*.

DESCRIPTION

The VP RNG is useful for producing identical streams of random numbers on partitions of different sizes, as long as the number of array elements is constant. The VP RNG is much faster than **CMF_RANDOM** and slightly faster than the Fast RNG; however, it uses far more processing element memory than either.

Usage. Follow these steps to use the VP RNG:

1. Call **initialize_vp_rng** (optional). This step is required only if the default initialization parameters are not suitable for your application.

2. Call **vp_rng** to generate the pseudo-random numbers. You may repeat this step as many times as you wish, but the state table parameters with which the RNG was initialized (explicitly or by default) must be appropriate for each call. If a **vp_rng** call requires different state table parameters, you must initiate a new state table by calling **initialize_vp_rng** with the new parameter values.

3. After all **vp_rng** calls associated with one set of state table parameters have finished, call **deallocate_vp_rng** to deallocate the state table.

To perform safety checkpointing, use the **save_vp_rng_temps** and **restore_vp_rng_temps** routines. To perform alternate stream checkpointing, use the **vp_rng_state_field**, **vp_rng_residue**, and **reinitialize_vp_rng** routines.

Initialization. The **initialize_vp_rng** routine allocates a VP RNG state table array as heap memory in the processing elements. The state table is initialized with values generated internally by **CMF_RANDOM**. The initialization routine also initializes internal state, including the state table information referenced by **vp_rng_state_field** and the residue referenced by **vp_rng_residue**.

Copyright © 1994 Thinking Machines Corporation

The VP RNG is designed to produce identical results on partitions of different sizes. To get identical results, be sure the following conditions are true:

- The size of ***A_template*** does not change between runs; this ensures that the same total number of array elements are used each time.
- The same *seed* is used each time. If you do not call **initialize_vp_rng** explicitly, **vp_rng** selects a dynamic seed value based on the system time. As a result, with default initialization, different results are obtained in different runs. Explicit initialization is required to obtain identical results.

The ***A_template*** parameter is used to determine the array geometry in which random values are used. A separate state table is allocated for each element of ***A***. The size of the state table in bits per element of ***A*** is the product of the *table_lag* and *width* parameters. For example, given a *width* of 32 and the recommended *table_lag* of 17, the state table occupies 544 bits per element of ***A***.

If your application requires a state table array configured differently from the default, you must call **initialize_vp_rng** (supplying a seed for **CMF_RANDOM** in the *seed* argument) before using **vp_rng**.

To produce the same results as the Fast RNG would produce, you must ensure that the total number of array elements used by the VP RNG equals the total number of processing elements used by the Fast RNG. Also, the *seed* you use to initialize the VP RNG state table must be the same as the seed you use when you call **CMF_RANDOMIZE** before calling **initialize_fast_rng**.

If you do not explicitly initialize the VP RNG, initialization occurs automatically when you first call **vp_rng**. The VP RNG selects a dynamic *seed* value based on the system time. The other defaults used for automatic initialization are 17 for *table_lag* and 5 for *short_lag*; for *width* defaults, see the Arguments section above. The ***A*** supplied in the first invocation of **vp_rng** is used as the default ***A_template***.

The state table parameters have an enormous effect on the results obtained by calls to **vp_rng**. Use the following guidelines for proper state table initialization:

- The *period* of a random number generator is the number of random values it produces before repeating the original stream. To avoid correlation, the period should be greater than the total number of random values produced for any subgrid element; that is,

  desired_period > *invocations*

  where *invocations* is the number of times the program calls **vp_rng**.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

- The period is very sensitive to the choice of *table_lag* and *short_lag* values. The default pair of values, (17, 5), has been carefully chosen to produce the maximum period for the minimum storage. Other suggested values are (55, 24) and (71, 35). When the lag pairs are properly chosen, the period of the VP RNG depends exponentially on the state table length (*table_lag*) and on the state table width (*width*), such that

  $$period = (2^{table_lag} - 1) \times 2^{width}$$

  For a discussion on choosing proper lag value pairs, see the book by Knuth referenced in Section 13.7.

- The RNG is *always* initialized with the values shown in the Arguments section, above.

If the subgrid size times the number of processing elements is *N*, the state table in each subgrid position is initialized as though it were a Fast RNG state table in a partition with *N* processing elements.

Random Number Generation. The **vp_rng** routine copies a pseudo-random value, chosen from a uniform distribution, into each element of ***A***. The distribution range in the floating-point case is from 0.0 (inclusive) to 1.0 (exclusive). The distribution range in the integer case with a positive *limit* is from 0 (inclusive) to the specified *limit* (exclusive). The distribution range in the integer case with a 0 *limit* is all integer values from -2^{31} to $2^{31}-1$ (in other words, any 32-bit pattern).

Each time the VP RNG is called, it generates one random value per subgrid element, and places the random values in ***A***. For any one stream of random values, all ***A*** arguments must have the same shape, size, and axis orderings. To generate random values for destination arrays that differ in these attributes, use alternate-stream checkpointing (described below).

The Save and Restore Routines. The **save_vp_rng_temps** and **restore_vp_rng_temps** routines provide a mechanism for safety checkpointing, allowing an application to resume processing from a checkpoint after a forced interruption.

To use the save and restore routines, insert calls to the save routine periodically among your RNG calls. If a forced interruption occurs, call the restore routine, which restores the state table to exactly the values it had when the state was saved. (You must rewind the file before calling **restore_vp_rng_temps**.) No reinitialization is required; you can call the RNG again immediately after restoring the state. The dimensions of the destination array must be the same when you restore the VP RNG state as when you saved it.

Copyright © 1994 Thinking Machines Corporation

The State Field, Residue, and Reinitialization Routines. The **vp_rng_state_field**, **vp_rng_residue**, and **reinitialize_vp_rng** routines provide a method for alternate-stream checkpointing, in which the application switches back and forth between two or more VP RNG streams by saving the state and reinitializing one RNG after the other.

To perform checkpointing using these routines, follow these steps:

1. After the VP RNG has been initialized either implicitly or explicitly by **initialize_vp_rng**, and after **vp_rng** has been called zero or more times, call **vp_rng_state_field** and **vp_rng_residue**.
2. Save the values of *state_table* and *residue*.
3. To restart a previously checkpointed VP RNG number stream, call **reinitialize_vp_rng** with the previously saved *state_table* and *residue* values.

The **vp_rng_state_field** routine returns internal information used to determine the current state of the VP RNG state table; this information was generated previously by an implicit or explicit call to **initialize_vp_rng**. The state table resides in the processing elements; its size is the product of the *table_lag* and *width*.

The **vp_rng_residue** routine returns the residue: a count of the number of times that the VP RNG state has been stepped, modulo the *table_lag*. The residue is the product of the subgrid size and the number of calls to **vp_rng** that have occurred since the last call to **initialize_vp_rng** (implicit or explicit) or to **reinitialize_vp_rng**.

The **reinitialize_vp_rng** routine reinitializes the VP RNG from a previously checkpointed state so that an interrupted computation can be resumed.

Deallocation. The **deallocate_vp_rng** routine deallocates the heap field that has been used to store the state table for the VP RNG. Call this routine when you are finished with the VP RNG. Using **deallocate_vp_rng** is important because the VP RNG state table field can use a significant amount of processing element memory.

NOTES

No Nodal Library Support for Save and Restore Routines. The save and restore routines do not work with the nodal CMSSL library.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Numerical Performance. The lagged-Fibonacci algorithm implemented by this RNG is widely used to produce a uniform distribution of random values.

For a *width* of 32 bits and using the default *table_lag* and *short_lag* values, (17, 5), the period of the VP RNG is $(2^{17}-1)2^{32} \approx 5.6\text{e}15$ bits. By comparison, the period for **CMF_RANDOM** is estimated to be 6.8e10 bits, with greater danger of cross-processing-element correlation.

Running time for the VP RNG increases with the state table width and the number of bits used. For best results, reduce the table width to the number of bits required and use a *limit* value of 0.

Applications. The VP RNG mimics the Fast RNG. It should be used in applications such as Monte Carlo simulations where reproducible results across different partition sizes must be verified.

Include the CMSSL Header File. Because the **vp_rng_residue** and **vp_rng_state_field** calls are functions, you must include the line

```
INCLUDE '/usr/include/cm/cmssl-cmf.h'
```

in program units that include calls to these routines. This file declares the types of the CMSSL functions and symbolic constants.

Reproducible Results. To obtain reproducible results from the VP RNG, call **initialize_vp_rng** using the same *seed* value each time.

In contrast, checkpointing and reinitializing an RNG is used to *continue* random value stream generation from a previous or alternate state.

No Error Checking on Reinitialization. The **reinitialize_vp_rng** routine does not perform error checking on the input parameters. Unpredictible results or halted execution are likely under the following conditions:

- The size of *state_table* is less than (*table_lag* × *width*).
- *residue* is negative or *residue* ≥ *table_lag*.

Use of CMF_RANDOM to Initialize the State Tables. The state table in each subgrid element is initialized as though it were a Fast RNG state table in a partition with size equal to the subgrid size times the number of processing elements.

Copyright © 1994 Thinking Machines Corporation

EXAMPLES

Sample CM Fortran code that uses the VP RNG routines can be found on-line in the subdirectory

```
random/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

13.7 References

For an analysis of the lagged-Fibonacci algorithm and a discussion of optimal lag parameter choices, see

1. Knuth, D. *The Art of Computer Programming*, Vol. 2: *Seminumerical Algorithms*. Reading, Mass.: Addison-Wesley, 1973. Pp. 26−28.

For a discussion of the vulnerability of lagged-Fibonacci generators to the Birthday Spacings Test, see

2. Marsaglia, G. A Current View of Random Number Generators. In *Computer Science and Statistics*, 16th Symposium on the Interface, Atlanta, March 1985.

Copyright © 1994 Thinking Machines Corporation