# Contents

1 Overview  
   1.1 Introduction .................................................. 5  
   1.2 License for use and copyright ................................. 5  

2 Before use  
   2.1 Software required for KMATH_RANDOM installation .......... 7  
   2.2 Obtaining KMATH_RANDOM ......................................... 7  
   2.3 KMATH_RANDOM directory configuration ....................... 7  
   2.4 Compile and install procedure ................................. 7  
      2.4.1 NTL installation ............................................ 8  
      2.4.2 Makefile.machinel settings ............................... 8  
      2.4.3 Selection of random number period ....................... 9  
      2.4.4 SIMD instruction enabling and disabling ................. 10  
      2.4.5 make ..................................................... 11  
   2.5 Application build ............................................. 11  
      2.5.1 For use of C interface .................................... 11  
      2.5.2 For use of C++ interface .................................. 12  
      2.5.3 For use of Fortran90 interface ......................... 12  

3 Interface explanation  
   3.1 KMATH_Random_Init ............................................ 13  
   3.2 KMATH_Random_Finalize ......................................... 14  
   3.3 KMATH_Random_Seed ............................................. 15  
   3.4 KMATH_Random_Get ............................................. 15  
   3.5 KMATH_Random_Vector .......................................... 16  
   3.6 KMATH_Random.Serialize ....................................... 17  
   3.7 KMATH_Random_Deserialize ................................. 18  
   3.8 Environment variables: KMATH_RAND_JUMP_FILE_PATH ........ 19  
   3.9 Environment variables: KMATH_RAND_JUMP_FILE_PREFIX .......... 19  

4 KMATH_RANDOM method of use  
   4.1 Creation of jump file ........................................ 21  
      4.1.1 Front-end/back-end tool builds ......................... 21  
      4.1.2 Jump tool execution .................................... 22  
   4.2 Basic method of use ........................................ 25  
   4.3 Internal state saving and restoring .......................... 25  
   4.4 Benchmark .................................................. 27  
      4.4.1 From benchmark creation to job submission .......... 27  
      4.4.2 Analysis of benchmark results ....................... 30
5 Conclusion

5.1 KMATH\_RANDOM, present and future

5.2 Acknowledgements
Chapter 1

Overview

1.1 Introduction

KMATH_RANDOM is a large-scale parallel random number generator routine which uses the dSFMT Mersenne twister random number generating algorithm [1]. This program set includes a test program for verification of its operation. KMATH_RANDOM supports C+, C++, and Fortran90 interfaces.

Its purpose is to help speed up programs in large-scale parallel computer environments by providing a fast, high-quality random number generating function to operate in those environments. In large-scale Monte Carlo simulation programs and other programs, it is necessary to generate pseudorandom numbers in large quantities with minimal bias. The function provided by KMATH_RANDOM generates highly uniform random number sequences with the extremely long periods needed to meet that requirement. It is designed to operate at a high speed in a parallel computer environment, to ensure that random number generation does not become the governing factor for the speed of overall program execution in large-scale parallel computer environments.

The dSFMT random number generating algorithm used in KMATH_RANDOM performs high-speed random number generation with an extremely long pseudorandom number sequence period of $2^{521} - 1$ to $2^{216091} - 1$ and a highly uniform distribution, and is thus characterized by excellence in both execution speed and random number quality. KMATH_RANDOM employs the dSFMT internally in generating the random number sequences and thereby shares similar properties with dSFMT.

To ensure that partial sequences of generated random numbers do not overlap between parallel execution ranks, KMATH_RANDOM reads and restores from a file (hereafter termed a “jump file”) a different random number internal state for each rank (by parallel processing in the flow shown in Fig. 1.1) on initialization of the random number generation routine. The random number internal state recorded in this file is created using the dSFMT jump function, by performing the jump operation sequentially for 2,000-rank portions, with a default random number generation range of 2100 per rank.

1.2 License for use and copyright

Permission to use KMATH_RANDOM is granted on the basis of the BSD 2-Clause License (found in LICENCE.txt in the library).
CHAPTER 1. OVERVIEW

Figure 1.1: KMATH_RANDOM process flow.

---

LICENCE.txt

Copyright (C) 2014 RIKEN.

------------------------------------------------------------------------
Copyright notice is from here
------------------------------------------------------------------------

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Chapter 2

Before use

2.1 Software required for KMATH_RANDOM installation

Several software packages are needed to compile KMATH_RANDOM. Operation has been verified for the following software.

- **NTL** Version 5.5 or later (needed to create the dSFMT jump file)
- **MPI** MPICH2 version 1.5 later, MPICH version 3.0.2 or later, OpenMPI version 1.6.4 or later
- **compiler** GNU compiler (gcc, gfortran, g++ version 4.1.2 or later) or Fujitsu compiler (mpifrtpx, mpifcpx, mpiFCC (cross compiler for K FX10))

2.2 Obtaining KMATH_RANDOM

Relevant information on KMATH_RANDOM can be obtained at the following URL.

http://www.aics.riken.jp/labs/lpnctrt/KMATH_RANDOM.html

Planning is in progress for provision of information on tarballs and on bugs and versions.

2.3 KMATH_RANDOM directory configuration

The directory configuration of this program is shown in Table 2.1.

It comprises random/ for storage of the random number library itself and the directory doc/ for documents (c/, c++/, f90/) for creation of the libraries for each language interface and the directories (tool/, ptool/) for jump tools used to create the jump files. It also includes a test program (test/) designed for development users.

2.4 Compile and install procedure

A number of steps are necessary to compile KMATH_RANDOM. Proceed in the order shown below.
Table 2.1: KMATH_RANDOM version 1.1 directory configuration.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>random/</td>
<td>Storage file</td>
</tr>
<tr>
<td>arch/</td>
<td>Random number library storage directory</td>
</tr>
<tr>
<td>c/</td>
<td>Included files for Makefile for each architecture</td>
</tr>
<tr>
<td>c++/</td>
<td>C++ interface source code</td>
</tr>
<tr>
<td>f90/</td>
<td>Fortran90 interface source code</td>
</tr>
<tr>
<td>dsfmt/</td>
<td>dSFMT source code, common source code for interfaces</td>
</tr>
<tr>
<td>test/</td>
<td>Source code for operation verification</td>
</tr>
<tr>
<td>0_comm_split/</td>
<td>MPI communicator split test</td>
</tr>
<tr>
<td>1_interface/</td>
<td>Interface test</td>
</tr>
<tr>
<td>2_serialize/</td>
<td>Serialization test</td>
</tr>
<tr>
<td>3_comparison/</td>
<td>Comparison test</td>
</tr>
<tr>
<td>dSFMT-src-2.2/</td>
<td>dSFMT original source code (for comparison of results)</td>
</tr>
<tr>
<td>4_benchmark/</td>
<td>Benchmark</td>
</tr>
<tr>
<td>kmath_random_v1.0/</td>
<td>KMATH Random v1.0 code (for speed comparison)</td>
</tr>
<tr>
<td>tool/</td>
<td>Jump file generation source code (for front end)</td>
</tr>
<tr>
<td>jump/</td>
<td>Jump file storage directory</td>
</tr>
<tr>
<td>ptool/</td>
<td>Jump file generation tool source code (for back end)</td>
</tr>
<tr>
<td>jump/</td>
<td>Jump file storage directory</td>
</tr>
<tr>
<td>doc/</td>
<td>Document storage directory</td>
</tr>
</tbody>
</table>

2.4.1 NTL installation

NTL developed by Victor Shoup is needed to control the KMATH_RANDOM jump files. It is used to generate the data required for jump file configuration. In the cross-compiling environment of front-end nodes in the K computer and other computers, it is accordingly sufficient to install NTL only on the front end and its installation on/in the back end is thus unnecessary (except where the objective is to generate a jump file in the back end, with the objective of reducing file transfers).

First, get the tarball from the developer’s site [2] (http://www.shoup.net/ntl/), and deploy it on an appropriate working directory and then move the directory to the subdirectory src/.

There, execute configure and make. The NTL website provides detailed directions on installing NTL.

```
ntl-7.0.1 compile
% tar zxvf ntl-7.0.1.tgz
% cd ntl-7.0.1/src
% /bin/sh ./configure
% make
```

2.4.2 Makefile.machine settings

Next, move to the KMATH_RANDOM directory and implement the KMATH_RANDOM build. Set the path to the NTL library directory created as described in the previous section, in the Makefile.machine file.
2.4. COMPILE AND INSTALL PROCEDURE

Makefile.machine

$ cd <project random root directory>
$ cat Makefile.machine

# compilers
F90 = mpifrtpx
CC = mpifccpx
CXX = mpiFCCpx
xCC = fcc
xCXX= FCC

CFLAGS += -I/home/ra000005/a03137/include
CPPFLAGS += -I/home/ra000005/a03137/include

F90FLAGS = $(FFLAGS) -Free
LFLAGS = -L/home/ra000005/a03137/lib
ARFLAGS =

LFLAGS_CPP = $(LFLAGS) -lntl
LFLAGS_C = $(LFLAGS) -lntl -lstd -lstd_mt -lstdc++
LFLAGS_F90 = $(LFLAGS) -lntl -lstd -lstd_mt -lstdc++

$

Settings are made in several places.

1. MPI compiler (F90, CC, CXX)

2. Compiler for front end (xCC, xCXX)

3. Addition to CFLAGS and CPPFLAGS of include path to CFLAG NTL

4. Addition to LFLAGS of library path to LFLAGS NTL

2.4.3 Selection of random number period

The default random number generation period of dSMFT is 219937. To change the period, change the following Makefile.machine file settings.
Makefile.machine

$ cat Makefile.machine

: 

#-- no debug
FFLAGS = -c -Kfast -Ksimd=2 -Cpp -DNDEBUG
CFLAGS = -c -Kfast -Ksimd=2 -DNDEBUG -DDSFMT_MEXP=19937
CPPFLAGS = -c -Kfast -Ksimd=2 -DNDEBUG -DDSFMT_MEXP=19937

#-- debug
# FFLAGS = -c -O0 -g -Cpp -DDEBUG
# CFLAGS = -c -O0 -g -DDEBUG -DDSFMT_MEXP=19937
# CPPFLAGS = -c -O0 -g -DDEBUG -DDSFMT_MEXP=19937

: 

$

The following periods are selectable. Select one and write it to Makefile.machine.

- DDSFMT_MEXP=521
- DDSFMT_MEXP=1279
- DDSFMT_MEXP=2203
- DDSFMT_MEXP=4253
- DDSFMT_MEXP=11213
- DDSFMT_MEXP=19937
- DDSFMT_MEXP=44497
- DDSFMT_MEXP=86243
- DDSFMT_MEXP=132049
- DDSFMT_MEXP=216091

If these settings are changed, it is then essential to re-create the jump files for restoring the random number internal state and the serialization files described below. Care is necessary, as abnormal termination of the program may occur if these files are used in the library without having been re-created.

2.4.4 SIMD instruction enabling and disabling

dSFMT SIMD instruction use is enabled by default. To disable this setting, comment out the following Makefile.machine settings.
2.4.5 make

The build methods for the C, C++, and Fortran interface libraries are as follows.

```
$ cd <kmath random root path>
$ make
$ find . -name "*.a"
./c++/libkm_random.a
./f90/libkm_random.a
./c/libkm_random.a
```

The libraries indicated by the find command following the make command are generated static libraries.

2.5 Application build

When the application build is performed using this routine on the K computer, it is necessary to specify the following parameters in compiling and linking. The order of specifying the library option is important, and if the order shown in the following example is not followed, link failure may occur.

2.5.1 For use of C interface

Compile:

```
-I<kmath random root directory>/c
```
2.5.2 For use of C++ interface

Compile:

-include <kmath random root directory>/c
-include <NTL path>/lib
-lkm_random -lntl -lstd -lstd_mt -lstdc++

Link:

-include <kmath random root directory>/c
-include <NTL path>/lib
-lkm_random -lntl -lstd -lstdc++

2.5.3 For use of Fortran90 interface

Compile:

-include <kmath random root directory>/f90

Link:

-include <kmath random root directory>/f90
-include <NTL path>/lib
-lkm_random -lntl -lstd -lstdc++
Chapter 3

Interface explanation

This chapter describes the interfaces of the large-scale massively parallel random number generation routine.

Note This interface group in its current version (ver. 1.1) is not thread-safe. For use in a multithreaded environment, the caller must properly perform mutual-exclusion processing.

3.1 KMATH_Random_Init

C Syntax

```c
#include <kmath_random.h>
void* KMATH_Random_init(MPI_Comm comm);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm</td>
<td>MPI_Comm</td>
<td>In</td>
<td>MPI communicator</td>
</tr>
<tr>
<td>return value</td>
<td>void*</td>
<td>Ret</td>
<td>handle</td>
</tr>
</tbody>
</table>

C++ Syntax

```c
#include <kmath_random.h>
bool KMATH_Random::init(MPI_Comm comm);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>comm</td>
<td>MPI_Comm</td>
<td>In</td>
<td>MPI communicator</td>
</tr>
<tr>
<td>return value</td>
<td>bool</td>
<td>Ret</td>
<td>state (true: normal)</td>
</tr>
</tbody>
</table>

Fortran90 Syntax

```fortran
use kmath_random.mod
subroutine KMATH_Random_Init(handle, comm, ierr)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>type(s_km_rand)</td>
<td>Out</td>
<td>handle</td>
</tr>
<tr>
<td>comm</td>
<td>integer</td>
<td>In</td>
<td>MPI communicator</td>
</tr>
<tr>
<td>ierr</td>
<td>integer</td>
<td>Out</td>
<td>State (0: normal)</td>
</tr>
</tbody>
</table>
CHAPTER 3. INTERFACE EXPLANATION

Specify the communicator comm and initialize the large-scale massively parallel random number generation routine with the initial seed value (1). The interface is a collective operation and must be called for all ranks simultaneously.

At the time of execution of the interface, the jump file(s) corresponding to seed value 1 is read and the random number internal state of each rank is restored. If the rank number in the communicator exceeds the limit (the maximum rank number recorded in the jump file) and the jump file thus cannot be read normally, the initialization will fail.

There is one jump file for each seed value, with the following (Fig. 3.1) binary formats.

<table>
<thead>
<tr>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum rank number</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Rank 0 initial random number internal state</td>
<td>3080 bytes</td>
</tr>
<tr>
<td>Rank 1 initial random number internal state</td>
<td>3080 bytes</td>
</tr>
<tr>
<td>...</td>
<td>3080 bytes</td>
</tr>
<tr>
<td>Rank N - 1 initial random number internal state</td>
<td>3080 bytes</td>
</tr>
</tbody>
</table>

Figure 3.1: Jump file internal formats.

3.2 KMATH_Random_Finalize

C Syntax

```c
#include <kmath_random.h>
int KMATH_Random_finalize(void* handle);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>void*</td>
<td>In</td>
<td>handle</td>
</tr>
</tbody>
</table>

C++ Syntax

```cpp
#include <kmath_random.h>
bool KMATH_Random::finalize();
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return value</td>
<td>int</td>
<td>Ret</td>
<td>state (0: success)</td>
</tr>
</tbody>
</table>

Fortran90 Syntax

```fortran
use kmath_random_mod
subroutine KMATH_Random_Finalize(handle, ierr)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>type(s_km_rand)</td>
<td>In</td>
<td>handle</td>
</tr>
<tr>
<td>ierr</td>
<td>integer</td>
<td>Out</td>
<td>State (0: normal)</td>
</tr>
</tbody>
</table>

Specify the handle and finalize the large-scale massively parallel random number generation routine. This interface, like the initialization, is a collective operation, and thus all ranks must be called simultaneously.
3.3 KMATH_RANDOM_SEED

3.3.1 KMATH_Random_Seed

C Syntax

```c
#include <kmath_random.h>
int KMATH_Random_seed(void* handle, int seed);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>void*</td>
<td>In</td>
<td>handle</td>
</tr>
<tr>
<td>seed</td>
<td>int</td>
<td>In</td>
<td>seed value</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
<td>Ret</td>
<td>State (0: success)</td>
</tr>
</tbody>
</table>

C++ Syntax

```c
#include <kmath_random.h>
bool KMATH_Random::seed(int seed);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seed</td>
<td>int</td>
<td>In</td>
<td>seed value</td>
</tr>
<tr>
<td>return value</td>
<td>bool</td>
<td>Ret</td>
<td>State (true: normal)</td>
</tr>
</tbody>
</table>

Fortran90 Syntax

```fortran
use kmath_random_mod
subroutine KMATH_Random_Seed(handle, seed, ierr)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>type(s_km_rand)</td>
<td>In</td>
<td>handle</td>
</tr>
<tr>
<td>seed</td>
<td>integer</td>
<td>In</td>
<td>seed value</td>
</tr>
<tr>
<td>ierr</td>
<td>integer</td>
<td>Out</td>
<td>state (0: normal)</td>
</tr>
</tbody>
</table>

The seed value is assigned for the random number. The jump file corresponding to the seed value is then read, and the random number internal state of each rank is restored. If the specified seed value or the rank number in the communicator exceeds the limit and the jump file thus cannot be read normally, this call will fail.

3.4 KMATH_Random_Get

C Syntax

```c
#include <kmath_random.h>
int KMATH_Random_get(void* handle, double* value);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>void*</td>
<td>In</td>
<td>handle</td>
</tr>
<tr>
<td>value</td>
<td>double*</td>
<td>In</td>
<td>random number value</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
<td>Ret</td>
<td>state (0: success)</td>
</tr>
</tbody>
</table>

C++ Syntax

```c
#include <kmath_random.h>
bool KMATH_Random::get(double& value) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>double&amp;</td>
<td>Out</td>
<td>random number value</td>
</tr>
<tr>
<td>return value</td>
<td>bool</td>
<td>Ret</td>
<td>state (true: normal)</td>
</tr>
</tbody>
</table>
**Fortran90 Syntax**

```fortran
use kmath_random_mod
subroutine KMATH_Random_Get(handle, value, ierr)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>type(s_km_rand)</td>
<td>In</td>
<td>handle</td>
</tr>
<tr>
<td>value</td>
<td>double precision</td>
<td>Out</td>
<td>random number value</td>
</tr>
<tr>
<td>ierr</td>
<td>integer</td>
<td>Out</td>
<td>state (0: normal)</td>
</tr>
</tbody>
</table>

One random number value is obtained. The obtained random number is normalized in the range $1.0 < v \leq 2.0$.

### 3.5 KMATH_Random_Vector

**C Syntax**

```c
#include <kmath_random.h>
int KMATH_Random_vector(void* handle, double* values, int size);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>void*</td>
<td>In</td>
<td>handle</td>
</tr>
<tr>
<td>values</td>
<td>double*</td>
<td>Out</td>
<td>pointer to random number sequence</td>
</tr>
<tr>
<td>size</td>
<td>int</td>
<td>In</td>
<td>obtained number</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
<td>Ret</td>
<td>state (0: success)</td>
</tr>
</tbody>
</table>

**C++ Syntax**

```cpp
#include <kmath_random.h>
bool KMATH_Random::get(double* values, int size) const;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>double&amp;</td>
<td>Out</td>
<td>pointer to random number sequence</td>
</tr>
<tr>
<td>size</td>
<td>int</td>
<td>In</td>
<td>obtained number</td>
</tr>
<tr>
<td>return value</td>
<td>bool</td>
<td>Ret</td>
<td>state (true: normal)</td>
</tr>
</tbody>
</table>

**Fortran90 Syntax**

```fortran
use kmath_random_mod
subroutine KMATH_Random_Vector(handle, values, nvalue, ierr)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>type(s_km_rand)</td>
<td>In</td>
<td>handle</td>
</tr>
<tr>
<td>values</td>
<td>double precision</td>
<td>Out</td>
<td>random number sequence</td>
</tr>
<tr>
<td>size</td>
<td>integer</td>
<td>In</td>
<td>obtained number</td>
</tr>
<tr>
<td>ierr</td>
<td>integer</td>
<td>Out</td>
<td>state: (0: normal)</td>
</tr>
</tbody>
</table>

The specified number of random numbers are obtained and stored in the array values. The obtained number must be 386 or more and must be divisible by 2. The obtained random number is normalized in the range $1.0 < v \leq 2.0$. 
3.6 KMATH_Random_Serialize

C Syntax

```c
#include <kmath_random.h>
int KMATH_Random_serialize(void* handle, const char* filename);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>void*</td>
<td>In</td>
<td>handle</td>
</tr>
<tr>
<td>filename</td>
<td>const char*</td>
<td>In</td>
<td>file name</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
<td>Ret</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0: no error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−1: unexecuted initialization</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−2: MPI failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−3: file I/O failure</td>
</tr>
</tbody>
</table>

C++ Syntax

```c
#include <kmath_random.h>
int KMATH_Random::serialize(const char* filename);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>const char*</td>
<td>In</td>
<td>file name</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
<td>Ret</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0: no error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−1: unexecuted initialization</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−2: MPI failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−3: file I/O failure</td>
</tr>
</tbody>
</table>

Fortran90 Syntax

```fortran
use kmath_random_mod
subroutine KMATH_Random_Serialize(handle, filename, ierr)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>type(s_km_rand)</td>
<td>In</td>
<td>handle</td>
</tr>
<tr>
<td>filename</td>
<td>character(*)</td>
<td>In</td>
<td>file name</td>
</tr>
<tr>
<td>ierr</td>
<td>integer</td>
<td>Out</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0: no error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−1: invalid handle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−2: MPI failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>−3: file I/O failure</td>
</tr>
</tbody>
</table>

The current random number internal state is serialized (saved) to the file specified by the filename. The interface is a collective operation and must be called for all ranks simultaneously.

The random number internal states in all ranks in the communicator are recorded in the serialized file. The rank 0 process is responsible for the actual serialization processing, and one file is thus created for one communicator.

The files are in the binary formats shown below (Fig. 3.2). The interfaces for C, C++, and Fortran90 are mutually compatible.
CHAPTER 3. INTERFACE EXPLANATION

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum rank number</td>
<td></td>
<td></td>
<td>4 bytes</td>
</tr>
<tr>
<td>Rank 0 initial random number internal state</td>
<td></td>
<td></td>
<td>3080 bytes</td>
</tr>
<tr>
<td>Rank 1 initial random number internal state</td>
<td></td>
<td></td>
<td>3080 bytes</td>
</tr>
<tr>
<td>\ldots</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rank (N-1) initial random number internal state</td>
<td></td>
<td></td>
<td>3080 bytes</td>
</tr>
</tbody>
</table>

Figure 3.2: Jump file internal formats.

3.7 KMATH\_Random\_Deserialize

C Syntax

```c
#include <kmath_random.h>
int KMATH_Random\_deserialize(void* handle, const char* filename);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handle</td>
<td>void*</td>
<td>Inout</td>
<td>handle</td>
</tr>
<tr>
<td>filename</td>
<td>const char*</td>
<td>In</td>
<td>file name</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
<td>Ret</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0: no error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-1: invalid handle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-2: MPI failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-3: file I/O failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-4: rank number mismatch</td>
</tr>
</tbody>
</table>

C++ Syntax

```cpp
#include <kmath_random.h>
int KMATH\_Random\::deserialize(const char* filename);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>IO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>const char*</td>
<td>In</td>
<td>file name</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
<td>Ret</td>
<td>error</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0: no error (normal)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-1: invalid handle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-2: MPI failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-3: file I/O failure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-4: rank number mismatch</td>
</tr>
</tbody>
</table>

Fortran90 Syntax

```fortran
use kmath_random_mod
subroutine KMATH\_Random\_Deserialize(handle, filename, ierr)
```
### 3.8 Environment variables: `KMATH_RAND_JUMP_FILE_PATH`

Specifies the jump file reference path. If this environment variable is not set, the default reference path will be as follows.

```
/etc/kmath/random/jump
```

### 3.9 Environment variables: `KMATH_RAND_JUMP_FILE_PREFIX`

Specifies the jump file prefix. If this environment variable is not set, the default prefix will be as follows.

```
file
```

The actual jump file is controlled with the assignment of the ID for the given seed type only (integer value) given, as in file_00001, file_00002, ....
Chapter 4

**KMATH_RANDOM method of use**

The KMATH_RANDOM process flow is as shown in Fig. 4 and also in Chapter 1. The following is a more detailed description.

1. Jump file generation
2. Jump file installation (store in appropriate directory)
3. Startup of program using KMATH_RANDOM
4. Reference to jump file storage location (environment variables KMATH_RAND_JUMP_FILE_PATH)
5. Initialization by `KMATH_Random_Init` or recovery of internal state from jump file by `KMATH_Random.Deserialize`
6. Obtaining of random number by `KMATH_Random_Get`, etc., independently for each process
7. KMATH_RANDOM finalization

In this chapter, Section 4.1 describes the method of jump file generation (step 1) and Sections 4.2 and 4.3 describe the method of program creation (executing steps 3 to 7) using KMATH_RANDOM. Section 4.4 provides a benchmark test and analysis for KMATH_RANDOM. For the build method for the KMATH_RANDOM library itself and the program build method using KMATH_RANDOM, which are not described in this chapter, refer to Sections 2.4 and 2.5, respectively.

### 4.1 Creation of jump file

This section describes the method of creating the jump files for restoration of the random number internal state in each rank in routine initialization.

#### 4.1.1 Front-end/back-end tool builds

The random number internal state tool (hereinafter called the “jump file”) builds are different for the front and back ends. The tool is created by `make tool` for the front end, and by `make ptool` for the back end. The source code is actually nearly the same for both. It differs
only in the need to distinguish between the compilers in creating the two, since the back end must be an MPI compiler.

Jump tool build for front end

```bash
$ cd <kmath random root path>
$ make tool
$ cd tool
$ ls -l
Makefile
dSFMT-calc-jump.hpp
dSFMT-jump.cpp
dSFMT-jump.h
dSFMT-jump.o
gen.sh
jump
dsfmt_jump.cpp
dsfmt_jump.h
dsfmt_jump.o

: 
$```

4.1.2 Jump tool execution

The two tools obtained by the builds in the previous section are `km_rand_gen_jump`, which creates the jump files, and `km_rand_chk_jump`, which validates the created jump file. Their use
4.1. **CREATION OF JUMP FILE**

is illustrated in the following examples.

**km_rand_gen_jump**

This tool creates the jump files, which are created only for numbers in the specified seed value range. The format is as follows.

```
./km_rand_gen_jump [parameter 1 [parameter 2]..]
```

The parameters that can be specified are as follows.

- **-seed** `seed_start` `seed_end`
  
  Seed value range. The jump file is created only for this seed value number. The default value is `seed_start=1, seed_end=1`.

- **-max_ranks** `<rank>`
  
  Maximum value of rank number. In random number generation, the rank size in the communicator must be no larger than this value. The default value is 1.

- **-rand_range** `<range>`
  
  Range of random number generation per rank. Specified as `2^range`. The default value is 100.

- **-install_dir** `<path>`
  
  Constructed jump file save destination. The default value is `./jump`.

- **-file_prefix** `<prefix>`
  
  Prefix of created jump file. In the actual jump file name, the seed number is given after this prefix. The default value is `file`.

**km_rand_chk_jump**

This tool validates the created jump file. It calls dSFMT directly without going through any interface. The random number internal states for all rank numbers recorded in the jump file are restored in the structure `dsfmt.t`, then one random number is generated and the value is output to the standard output file. The format is as follows.

```
./km_rand_chk_jump <jump file>
```

**Example of km_rand_get_gen_jump execution (for front end)**

```
jump file creation

$ mkdir jump
km_rand_gen_jump -seed 1 10 -max_ranks 2000
$ ls ./jump
file_00001 file_00003 file_00005 file_00007 file_00009
file_00002 file_00004 file_00006 file_00008 file_00010
$ 
```
The created files, file_00001, file_00002, ..., are used to initialize the random number generator. The file information is as follows.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seed value range</td>
<td>1 ~ 10</td>
</tr>
<tr>
<td>Maximum rank number</td>
<td>2000</td>
</tr>
<tr>
<td>Random number generation range per rank</td>
<td>2^{100}</td>
</tr>
<tr>
<td>Jump file save destination</td>
<td>./jump</td>
</tr>
<tr>
<td>Jump file prefix</td>
<td>file</td>
</tr>
</tbody>
</table>

When random numbers are generated by KMATH_Random_Get and KMATH_Random_Vector, these files are read and the random number internal state controlled on memory changes. Care is necessary in this regard, since the states serialized by KMATH_Random.Serialize and newly written to the files differ from the initial internal states constructed by the tool. In any instance where reproduction from the initial state or snapshot reproduction is desired, the jump files must then be saved each time and passed to KMATH_Random.Deserialize.

It is also necessary to recreate the jump files if there is a change in the setting for use or non-use of SIMD instructions by dSFMT and also if the random number period is changed.

**Example of km_rand_get_gen_jump execution (batch process for back end)**

The jump tool for the back end is used to generate jump files at the back end. To use it, refer back to the previous section, as its use is illustrated by the same example as that given for the front end in that section.

A job submission to the queue system is required to execute the tool at the back end. For use in an interactive job, there is no substantial difference from the front end. For use in a batch job, however, refer to the following example of job script following ptool/ for batch job input.

```
Batch job execution

$ cat gen.sh
#!/bin/bash -x
#
#PJM --rsc-list "node=10"
#PJM --rsc-list "elapse=01:00:00"
#PJM --stg-transfiles all
#PJM --stgin "./km_rand_gen_jump ./"
#PJM --stgout-dir "./jump ./jump"
#PJM -s
#
./work/system/Env_base
mkdir jump
mpiexec -n 10 ./km_rand_gen_jump -seed 1 10 -max_ranks 2000
$ pjsub gen.sh
[INFO] PJM 0000 pjsub Job 2243314 submitted.
$
$: 
$: ls ./jump
file_00001 file_00003 file_00005 file_00007 file_00009
file_00002 file_00004 file_00006 file_00008 file_00010
$```
4.2 Basic method of use

For random number generation using KMATH_RANDOM, in addition to calling the generating subroutine itself, it is necessary to call several subroutines for preprocessing and post-processing. The following shows the type of procedure needed, by illustrating the method for creation of a simple program for performing random number generation by multiple processes with sample program test/1_interface/test_c_seq.c (Fig. 4.2) as an example. The program test_c_seq.c starts up the number of MPI processes (ranks) specified at the time of execution, and for all ranks performs 10 iterations of random number generation and writing to files.

**Header file reading** Reads header file for use of the function KMATH_RANDOM (Fig. 4.2, line 13). For C++ and Fortran90 interfaces, similarly reads the C++ header file (#include <kmath_random.h>) and the Fortran module (use kmath_random_mod), respectively.

**Required variable declaration** Declares the variable that records the KMATH_RANDOM handle (Fig. 4.2, line 19).

**Random number generation routine initialization** Specifies the MPI communicator used for random number generation, executes the function KMATH_Random_Init in all ranks, and initializes the random number generation routine (Fig. 4.2, line 22).

**Random number seed setting** Sets the random number seed using the function KMATH_Random_Seed (Fig. 4.2, line 36). At this time, the jump files determined by the environment variables KMATH_JUMP_FILE_PATH and KMATH_JUMP_FILE_PREFIX and the random number seed value are read, and the random number internal state of each rank is restored.

**Random number generation** Random numbers are generated independently for each rank (Fig. 4.2, line 41). As shown by the example in Fig. 4.2, it is possible to generate each random number individually by the function KMATH_Random_Get, or else generate multiple random numbers together by the function KMATH_Random_Vector.

**Random number generation routine finalization** Executes the function KMATH_Random_Finalize in all ranks, and finalizes the random number generation routine (Fig. 4.2, line 47).

4.3 Internal state saving and restoring

KMATH_RANDOM includes the functions KMATH_Random.Serialize and KMATH_Random.Deserialize for saving (serializing) and restoring the internal state of the random number generation routine. If these functions are used, then following KMATH_Random_Init it is possible to save and restore the internal state at any point up to the end of the execution by KMATH_Random_Finalize. See Sections 3.6 and 3.7 for details on these functions. Both functions are collective operations, and care is essential as all ranks must be called simultaneously.

The sample program test/2_serialize/test_c_io.c (Figs. 4.3 and 4.4) performs saving and restoring of the internal state. In this program, the mode number is given by the first argument in execution and the number of random numbers generated is given by the second, where the processing is performed in accordance with the mode number as follows.

- If mode number 1: Random numbers in the specified number are generated and the results are written to the file.

/**
 * file: test_c_seq.c
 * brief: test of KMath random C module
 * authors: Toshiyuki Inamura (TI)
 * date: 2013/02/04 (NT)
 * date: 2013/12/17 (NT)
 * (c) Copyright 2013 RIKEN. All rights reserved.
 */

#include "kmath_random.h"
#include <stdio.h>

int main(int argc, char** argv)
{
    int comm_rank, i;
    void* h;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);
    h = KMATH_Random_init(MPI_COMM_WORLD);
    if (h == NULL)
    {
        printf("Failed to initialize. rank:%d\n", comm_rank);
        MPI_Finalize();
        return -1;
    }
    char file[128];
    sprintf(file, "out_c_seq_rnk%04d", comm_rank);
    FILE* fp = fopen(file, "w");
    KMATH_Random_seed(h, 1);
    for(i = 0; i < 10; ++i)
    {
        double v;
        KMATH_Random_get(h, &v);
        fprintf(fp, "%17.15f\n", v);
    }
    fclose(fp);
    KMATH_Random_finalize(h);
    MPI_Finalize();
    return 0;
}

Figure 4.2: Source code test/1_interface/test_c_seq.c
4.4 Benchmark

A benchmark test is performed on the routines of both the current version (1.1) and the previous version (1.0). In the previous version, the build is done without SIMD instruction execution enabled. In either case, random numbers are generated one billion times and the processing time is measured.

4.4.1 From benchmark creation to job submission

From benchmark program creation to job submission

```bash
$ cd <kmath random root directory>/random/test/4_benchmark
$
$ ls -1
Makefile
kmath_random_v1.0
run.sh
run_small.sh
test.c
$
$ make
:
$ ls -1F | grep \
  test*
$ $ cd kmath_random_v1.0/__comparison/
$ make
:
$ ls -1F | grep \
  test*

$ cd <kmath random root directory>/test/4_benchmark
$ pjsub run.sh
[INFO] PJM 0000 pjsub Job 2246378 submitted.

$ kmath_random_v1.0/__comparison/
$ pjsub run.sh
[INFO] PJM 0000 pjsub Job 2246379 submitted.
```

- If mode number 2: Random numbers in the specified number are generated and the results are written to the file, and the internal states of each rank are then collected and written together to the file (Fig. 4.4, line 15) by the function `KMATH Random Serialize`.

- If mode number 3: The file recording the internal states is read by the function `KMATH Random Deserialize` (Fig. 4.4, line 29), the internal state of each rank is restored, and random numbers in the specified number are then generated and written to the file.
CHAPTER 4. KMath_RANDOM METHOD OF USE

/*
 * 
 * file  test_c_io.c
 * brief  serialize test of KMath random C module
 * authors Toshiyuki Imamura (TI)
 * date   2013/12/17 (NT)
 * 
 * (c) Copyright 2013 RIKEN. All rights reserved.
 */

#include "kmath_random.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    int comm_rank, i, mode, count;
    void* h;
    if (argc < 3)
        return -1;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank);
    h = KMATH_Random_init(MPI_COMM_WORLD);
    if (h == NULL)
    {
        printf("Failed to initialize. rank:%d\n", comm_rank);
        goto ERR;
    }
    mode = atoi(argv[1]);
    count = atoi(argv[2]);
    FILE* fp;
    char ofile[256];
    switch(mode)
    {
    case 1:
        KMATH_Random_seed(h, 1);
        sprintf(ofile, "out_c_io_1_%04d", comm_rank);
        fp = fopen(ofile, "w");
        for(i = 0; i < count; ++i)
        {
            double v;
            KMATH_Random_get(h, &v);
            fprintf(fp, "Rank:%04d V:%f\n", comm_rank, v);
        }
        fclose(fp);
        break;
    }
}

Figure 4.3: Source code test/1_interfaced/test_c_io.c (1/2).
4.4. BENCHMARK

```c

case 2:
    KMATH_Random_seed(h, 1);
    sprintf(ofile, "out_c_io_2_%04d", comm_rank);
    fp = fopen(ofile, "w");
    for(i = 0; i < count; ++i)
    {
        double v;
        KMATH_Random_get(h, &v);
        fprintf(fp, "Rank:%04d V:%f\n", comm_rank, v);
    }
    if (KMATH_Random_serialize(h, ".\data\c\io") != 0)
    {
        printf("Write ERROR\n");
        break;
    }
    fclose(fp);
    break;

case 3:
    sprintf(ofile, "out_c_io_2_%04d", comm_rank);
    fp = fopen(ofile, "a");
    if (KMATH_Random_deserialize(h, ".\data\c\io") != 0)
    {
        printf("Read ERROR\n");
        break;
    }
    for(i = 0; i < count; ++i)
    {
        double v;
        KMATH_Random_get(h, &v);
        fprintf(fp, "Rank:%04d V:%f\n", comm_rank, v);
    }
    fclose(fp);
    break;

    KMATH_Random_finalize(h);

ERR:
    MPI_Finalize();
    return 0;
```

Figure 4.4: Source code test/1_interface/test_c_io.c (2/2).
Runs the built execution file. Refer to the shell script `run.sh` for batch job submission, which is given in the benchmark program directory. One command line parameter can be specified and the random number seed can be changed.

### 4.4.2 Analysis of benchmark results

With the above job submission, both the former and the present version execute a 1024-process parallel test program and generate random numbers. The following is a comparison of the results obtained on the two versions. The log shows five items: KMATH, Rand, Diff, First, and Last, which give the following information.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMATH</td>
<td>Time from <code>KMATH_Init</code> call to <code>KMATH_Finalize</code> call</td>
</tr>
<tr>
<td>Rand</td>
<td>Time of 1 billion calls of <code>KMATH_PseudoRand</code></td>
</tr>
<tr>
<td>Diff</td>
<td>Time required for initialization and discard (= KMATH Rand)</td>
</tr>
<tr>
<td>First</td>
<td>First random number value</td>
</tr>
<tr>
<td>Last</td>
<td>Last random number value (the billionth)</td>
</tr>
</tbody>
</table>

**Computing time**

In this example analysis of the execution results, they are sorted by routine initialization time with the rank requiring the longest time at the bottom. The processing time is found to be somewhat shorter on the current version, which is presumably attributable to its elimination of the overhead due to initialization of the jump and other operations at initialization.

**File IO time**

The jump file read time depends on the rank number (i.e., the jump file size) and the initialization of a large number of computing nodes takes time. As roughly estimated from experiments on the K computer, a jump file recording the random number internal state for 10000 ranks reaches approximately 3 MB. For staging in and reading a jump file with a file size of around 3 MB, file opening (`fopen`) takes about 0.35 seconds processing time in each rank and each file reading (`fread`) takes about 0.2 to 0.4 seconds. In the execution of the above benchmark, the jump file size is 6032 KB.

**SIMD performance**

The number of executions of SIMD instructions can be determined by comparing system logs. Basically, SIMD parallelization is performed by the compiler. In the previous version, as in the current version, SIMD instructions are issued. In the current version, the number of SIMD instructions issued is about 20 to 30%.
### 4.4. BENCHMARK

Results of benchmark program execution

```bash
$ cat run.sh.o2246378 | sort -k6 | tail
Rank00672: KMATH: 18.765974 | Rand: 18.463989 | Diff: 0.301985 | First: 1.562129 | Last: 1.176186
Rank00765: KMATH: 18.751880 | Rand: 18.464001 | Diff: 0.287879 | First: 1.171197 | Last: 1.862207
Rank00788: KMATH: 18.757943 | Rand: 18.464019 | Diff: 0.293924 | First: 1.881494 | Last: 1.628772
Rank00042: KMATH: 18.877124 | Rand: 18.464134 | Diff: 0.412990 | First: 1.516410 | Last: 1.848048
Rank00258: KMATH: 18.864152 | Rand: 18.464231 | Diff: 0.399921 | First: 1.045596 | Last: 1.779265
Rank00121: KMATH: 18.891643 | Rand: 18.464245 | Diff: 0.427398 | First: 1.216335 | Last: 1.173056
Rank00473: KMATH: 18.994540 | Rand: 18.464328 | Diff: 0.530212 | First: 1.968918 | Last: 1.118621
Rank00179: KMATH: 18.925652 | Rand: 18.464336 | Diff: 0.461316 | First: 1.260199 | Last: 1.467264

$ cat kmath_random_v1.0/__comparison/run.sh.o2246379 | sort -k6 | tail

$ cat run.sh.o2246378 | sort -k9 | tail
Rank00843: KMATH: 19.070450 | Rand: 18.459434 | Diff: 0.611016 | First: 1.077725 | Last: 1.516480
Rank00987: KMATH: 19.039628 | Rand: 18.428569 | Diff: 0.611059 | First: 1.127347 | Last: 1.708528

$ cat kmath_random_v1.0/__comparison/run.sh.o2246379 | sort -k9 | tail

$ tail -n 5 run.sh.12246378

$ cat kmath_random_v1.0/__comparison/run.sh.12246379 | sort -k9 | tail

$ tail kmath_random_v1.0/__comparison/run.sh.12246379

$ tail -n 5 run.sh.12246378

DISK SIZE (USE): -
I/O SIZE: 8982.3 MB (8982214990)
FILE I/O SIZE: 8943.3 MB (8943265297)
EXEC INST NUM: 3851825469617
EXEC SIMD NUM: 2165386352

$ tail kmath_random_v1.0/__comparison/run.sh.12246379

$ tail -n 5 run.sh.12246378

DISK SIZE (USE): -
I/O SIZE: 392.4 MB (392336626)
FILE I/O SIZE: 353.4 MB (353387891)
EXEC INST NUM: 4696517045425
EXEC SIMD NUM: 1896914464

$
Chapter 5

Conclusion

5.1 KMATH_RANDOM, present and future

KMATH_RANDOM currently has the following limitations. Certain functions are under consideration for addition, which may bring improvements in future.

Firstly, as noted in Chapter 3, KMATH_RANDOM is not currently thread-safe. For this reason, even when execution is parallel in the nodes, the need arises to generate multiple MPI processes.

Secondly, the period of the random number sequence is determined at the time of the KMATH_RANDOM build, and this parameter is limited in that it cannot be changed dynamically. For this reason, users wishing to use several different random number periods must build the same number of libraries as the required number of periods. We are now considering resolution of this problem by adding a function for selecting periods arbitrarily at execution.

Finally, we are considering the addition of a function that can replace the current random number generation algorithm, which is fixed in dSFMT, with an arbitrary random number generation library.

5.2 Acknowledgements

The results of KMATH_RANDOM execution described in this user manual were obtained on the K computer of RIKEN.
Bibliography

[1] SIMD-oriented Fast Mersenne Twister (SFMT): twice faster than Mersenne Twister,
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index-jp.html

[2] NTL: A Library for doing Number Theory,
http://www.shoup.net/ntl/