ユーザズ・マニュアル

KMATH_RANDOM

Version 1.1

大規模並列数値計算技術研究チーム
理化学研究所
計算科学研究機構

2014 年 12 月 12 日

目 次

第1章　概要

1.1　はじめに

KMATH_RANDOM はメルセンヌツイスタ乱数生成アルゴリズム dSFMT[1] を用いた大規模並列乱数生成ルーチンである。本プログラムセットは、その動作検証のためのテストプログラムを含む。KMATH_RANDOM はインタフェースとして、C、C++、Fortran90 をサポートする。

KMATH_RANDOM は、並列計算機環境において動作する高速・高品質な乱数生成機能を提供することにより、大規模並列計算機環境におけるプログラム高速化を支援することを目的としている。大規模モンテカルロ・シミュレーションをはじめとするプログラムでは、偏りの少ない大量の擬似乱数を生成する必要がある。KMATH_RANDOM はその需要を満たすべく周期が極めて長く一様性の高い乱数列を生成する機能を提供する。また、大規模並列計算機環境におけるプログラムの実行時に乱数生成がプログラム全体の実行速度を律することのないよう、KMATH_RANDOM は並列計算機環境で高速に動作するように設計されている。

KMATH_RANDOM で使用される乱数生成アルゴリズム dSFMT は、乱数生成速度が高速であり、擬似乱数列の周期が $2^{521}-1$ から $2^{216091}-1$ と極めて長く、さらに、高い均等分布性を備えるという、実行速度、乱数の品質の両面において優れた性質をもつ。KMATH_RANDOM は、内部的に dSFMT を利用して乱数列を生成する。このため、KMATH_RANDOM は dSFMT と同様の性質をもつ。

KMATH_RANDOM は生成する乱数の部分列が並列実行のランク間で重複しないよう、乱数生成ルーチンの初期化時に、各ランク毎に異なる乱数内部状態をファイルから読み込み復元する (下図 1.1 の流れで、並列処理される)。以降、このファイルをジャンプファイルと呼ぶ。このファイルに記録される乱数内部状態は、dSFMT ジャンプ機能を使用して、デフォルトでは 1 ランクあたり乱数生成範囲を 2^{100} とし 2000 ランク分だけ順次ジャンプ操作をおこなうことで作成されるものである。

1.2　利用許諾/Copyright

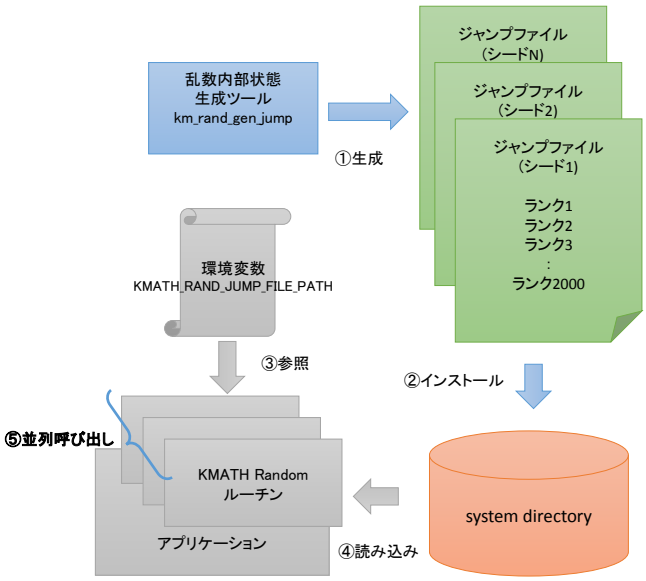

図 1.1: KMATH_RANDOM の処理の流れ

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.
------------------------------------------------------------------------
```

第2章　利用の前に

2.1　KMATH_RANDOMインストールに必要なソフトウェア

KMATH_RANDOM をコンパイルするためにはいくつかのソフトウェアパッケージが必要である。現在動作確認されているソフトウェアを以下に示す。

NTL	Version 5.5 以降 (dSFMT のジャンプファイル作成に必要)
MPI	MPICH2 version 1.5 以降、MPICH version 3.0.2 以降
	OpenMPI version 1.6.4 以降
コンパイラ	GNU コンパイラ（gcc, gfortran, g++ version 4.1.2 以降）または
	富士通コンパイラ
	（mpifrtpx, mpifccpx, mpiFCC（京, FX10 用のクロスコンパイラ)）

2.2　KMATH_RANDOMの入手方法

KMATH_RANDOM に関する情報は次の URL から入手可能である。

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

tarball の他、バグ、バージョン情報も提供されていく予定である。

2.3　KMATH_RANDOMのディレクトリ構成

本プログラムのディレクトリ構成は表 2.1 のとおりである。

乱数ライブラリ本体を格納する random/とドキュメント用ディレクトリ doc/、その下には各言語インターフェース向けのライブラリ作成ディレクトリ (c/, c++/, f90/) とジャンプファイルを作成するジャンプツール用ディレクトリ (tool/, ptool/) がある。また、開発ユーザ向けにテストプログラム (test/) が含まれている。

2.4　コンパイルとインストール手順

KMATH_RANDOM のコンパイルには幾つかの手順が必要である。以下の手順に沿って進めてほしい。

表 2.1: KMATH_RANDOM version 1.1 のディレクトリ構成

ディレクトリ	格納ファイル
random/	乱数ライブラリ本体を格納ディレクトリ
arch/	アーキテクチャ別 Makefile 用インクルードファイル
c/	C 用インタフェースのソースコード
c++/	C++用インタフェースのソースコード
f90/	Fortran90 用インタフェースのソースコード
dsfmt/	dSFMT ソースコード、各インタフェース共通利用ソースコード
test/	動作検証用のソースコード
1_interface/	インタフェーステスト
2_serialize/	直列化テスト
3_comparison/	比較テスト
dSFMT-src-2.2/	dSFMT オリジナルソースコード（結果比較用）
4_benchmark/	ベンチマーク
kmath_random_v1.0/	KMATH Random v1.0 のコード（速度比較用）
tool/	ジャンプファイル生成ツールソースコード (フロントエンド用)
jump/	ジャンプファイル保管ディレクトリ
ptool/	ジャンプファイル生成ツールソースコード (バックエンド用)
jump/	ジャンプファイル保管ディレクトリ
doc/	ドキュメント格納ディレクトリ

2.4.1　NTL のインストール

KMATH_RANDOM のジャンプファイルの管理には Victor Shoup 氏が開発する NTL が必要である。NTL はジャンプファイル構成に必要なデータを生成するために用いる。従って京コンピュータなどのフロントエンドノードでクロスコンパイルする環境では、フロントエンド側にのみ NTL を整備すればよく、バックエンド側では特に NTL の整備は必要ない (ファイル転送を減らす目的でバックエンド上でジャンプファイルを生成するときには必要である)。

まず、開発者のサイト [2] (`http://www.shoup.net/ntl/`) より tarball を入手し、適当なワーキングディレクトリ上で展開する。そして、ディレクトリをサブディレクトリ src/まで移動する。そこで, configure と make を実行する。NTL インストールに関する詳しい説明は NTL のホームページにある。

ntl-7.0.1 のコンパイル

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

2.4.2 Makefile.machine の設定

次に、KMATH_RANDOM のディレクトリに移動し、KMATH_RANDOM のビルドを行う。まず、前節で作成した NTL ライブラリへのパスを Makefile.machine ファイルに設定する。

Makefile.machine

```
$ cd <kmath 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++

$
```

設定箇所は数箇所ある。

1. MPI コンパイラ (F90, CC, CXX)
2. フロントエンド用コンパイラ (xCC, xCXX)
3. CFLAG NTL へのインクルードパスを CFLAGS と CPPFLAGS に追加する。
4. LFLAGS NTL へのライブラリパスを LFLAGS に追加する。

2.4.3　乱数周期の選択

dSFMT の乱数生成の周期はデフォルトで 2^{19937} となっている。この周期を変更するには、Makefile.machine ファイルの以下の設定を変更する。

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

:

$
```

選択可能な周期は以下のとおりで、一つを選択して Makefile.machine に記述する。

```
-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
```

この設定を変更した場合、乱数内部状態復元のためのジャンプファイルおよび後述の直列化ファイルは作成し直さなくてはならない。そのままこれらのファイルをライブラリで利用した場合、プログラムは異常終了する場合があるので注意が必要である。

2.4.4 SIMD 命令の有効化・無効化

dSFMT の SIMD 命令の使用がデフォルトで有効となっている。この設定を無効にする場合は、Makefile.machine の以下の設定をコメントアウトする。

Makefile.machine

```
$ cat Makefile.machine

# compilers
F90   = mpifrtpx
CC    = mpifccpx
CXX   = mpiFCCpx

:

#-- SSE2
CFLAGS          += -DHAVE_SSE2
CPPFLAGS        += -DHAVE_SSE2

:

$
```

この設定を変更した場合、乱数内部状態復元のためのジャンプファイルおよび後述の直列化ファイルは作成し直さなくてはならない。そのままこれらのファイルをライブラリで利用した場合、プログラムは異常終了する。

また、この設定が有効になっている場合とそうでない場合とで、生成される乱数列は異なるので注意が必要である。

2.4.5 make

C、C++、Fortran90 用それぞれのインタフェースライブラリのビルド方法は以下のとおり行う。

ライブラリのビルド

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

make コマンド後の find コマンドで表示されたライブラリは、生成されたスタティックライブラリファイルである。

2.5　アプリケーションのビルド

京コンピュータ上で、本ルーチンを利用したアプリケーションをビルドする場合、コンパイル、リンク時にそれぞれ以下のパラメータを指定する必要がある。なお、ライブラリオプションの指定順は重要であり、以下の例に従わない場合リンクに失敗する可能性がある。

2.5.1　C インタフェースを利用する場合

コンパイル：

```
-I<kmath random root directory>/c
```

リンク：

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

2.5.2　C++インタフェースを利用する場合

コンパイル：

```
-I<kmath random root directory>/c++
```

リンク：

```
-L<kmath random root directory>/c++
-L<NTL path>/lib
-lkm_random -lntl
```

2.5.3　Fortran90 インタフェースを利用する場合

コンパイル：

```
-I<kmath random root directory>/f90
```

リンク：

```
-L<kmath random root directory>/f90
-L<NTL path>/lib
-lkm_random -lntl -lstd -lstd_mt -lstdc++
```

第3章　インターフェイス解説

本章は大規模超並列乱数生成ルーチンのインタフェース解説である。

注意　本インタフェース群は現バージョン (ver. 1.1) ではスレッドセーフではない。マルチスレッド環境で利用する場合は、呼び出し側が適切に排他処理を行う必要がある。

3.1　KMATH_Random_Init

C Syntax

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

引数	型	IO	説明
comm	MPI_Comm	In	MPI コミュニケータ
戻り値	void*	Ret	ハンドル

C++ Syntax

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

引数	型	IO	説明
comm	MPI_Comm	In	MPI コミュニケータ
戻り値	bool	Ret	状態（true: 正常)

Fortran90 Syntax

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

引数	型	IO	説明
handle	type(s_km_rand)	Out	ハンドル
comm	integer	In	MPI コミュニケータ
ierr	integer	Out	状態（0: 正常)

コミュニケータ comm を指定し大規模超並列乱数生成ルーチンを初期シード値 (1) で初期化する。このインタフェースは、集団操作であり全てのランクで同時にコールしなければならない。

このインタフェース実行時、シード値 1 に対応するジャンプファイルが読み込まれ、各ランクの乱数内部状態が復元される。もし、コミュニケータ内のランク数が、上限であるジャンプファイルに記録された最大ランク数を超えてしまい、ジャンプファイルの読み込みが正常にできなかった場合、初期化に失敗する。

ジャンプファイルは、1 つのシード値に 1 つ存在し、それぞれ以下 (図 3.1) のバイナリ形式のフォーマットを有する。

最大ランク数	4 バイト
ランク 0 の初期乱数内部状態	3080 バイト
ランク 1 の初期乱数内部状態	3080 バイト
⋮	
ランク N-1 の初期乱数内部状態	3080 バイト

図 3.1: ジャンプファイル内部フォーマット

3.2 KMATH_Random_Finalize

C Syntax

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

引数	型	IO	説明
handle	void*	In	ハンドル
戻り値	int	Ret	状態 (0:成功)

C++ Syntax

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

引数	型	IO	説明
戻り値	bool	Ret	状態（true: 正常)

Fortran90 Syntax

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

引数	型	IO	説明
handle	type(s_km_rand)	In	ハンドル
ierr	integer	Out	状態（0: 正常)

ハンドルを指定して、大規模超並列乱数生成ルーチンをファイナライズする。このインタフェースは、初期化と同様に集団操作であるため、全てのランクで同時にコールしなければならない。

3.3 KMATH_Random_Seed

C Syntax

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

引数	型	IO	説明
handle	void*	In	ハンドル
seed	int	In	シード値
戻り値	int	Ret	状態 (0:成功)

C++ Syntax

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

引数	型	IO	説明
seed	int	In	シード値
戻り値	bool	Ret	状態（true: 正常)

Fortran90 Syntax

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

引数	型	IO	説明
handle	type(s_km_rand)	In	ハンドル
seed	integer	In	シード値
ierr	integer	Out	状態（0: 正常)

乱数のシード値を与える。このとき、シード値に対応するジャンプファイルが読み込まれ、各ランクの乱数内部状態が復元される。もし、指定したシード値または、コミュニケータ内のランク数が上限を超え、ジャンプファイルの読み込みが正常にできなかった場合、この呼び出しは失敗する。

3.4 KMATH_Random_Get

C Syntax

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

引数	型	IO	説明
handle	void*	In	ハンドル
value	double*	In	乱数値
戻り値	int	Ret	状態 (0:成功)

C++ Syntax

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

引数	型	IO	説明
value	double&	Out	乱数値
戻り値	bool	Ret	状態（true: 正常)

Fortran90 Syntax

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

引数	型	IO	説明
handle	type(s_km_rand)	In	ハンドル
value	double precision	Out	乱数値
ierr	integer	Out	状態（0: 正常)

乱数値を 1 つ取得する。得られた乱数は $1.0 < v \leq 2.0$ の範囲で正規化されている。

3.5 KMATH_Random_Vector

C Syntax

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

引数	型	IO	説明
handle	void*	In	ハンドル
values	double*	Out	乱数配列へのポインタ
size	int	In	取得数
戻り値	int	Ret	状態 (0:成功)

C++ Syntax

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

引数	型	IO	説明
values	double&	Out	乱数配列へのポインタ
size	int	In	取得数
戻り値	bool	Ret	状態（true: 正常)

Fortran90 Syntax

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

引数	型	IO	説明
handle	type(s_km_rand)	In	ハンドル
values(:)	double precision	Out	乱数配列
size	integer	In	取得数
ierr	integer	Out	状態（0: 正常)

指定した取得数分だけ乱数値を配列に取得する。ただし、取得数は、386 個以上でかつ、2 で割り切れる値でなければならない。得られた乱数は $1.0 < v \leq 2.0$ の範囲で正規化されている

3.6 KMATH_Random_Serialize

C Syntax

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

引数	型	IO	説明
handle	void*	In	ハンドル
filename	const char*	In	ファイル名
戻り値	int	Ret	エラー 0: エラーなし (正常) −1: 初期化の未実行 −2 : MPI 失敗 −3: ファイル I/O 失敗

C++ Syntax

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

引数	型	IO	説明
filename	const char*	In	ファイル名
戻り値	int	Ret	エラー 0: エラーなし (正常) −1: 初期化の未実行 −2: MPI 失敗 −3: ファイル I/O 失敗

Fortran90 Syntax

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

引数	型	IO	説明
handle	type(s_km_rand)	In	ハンドル
filename	character(*)	In	ファイル名
ierr	integer	Out	エラー 0: エラーなし (正常) −1: 無効なハンドラ −2: MPI 失敗 −3: ファイル I/O 失敗

現在の乱数内部状態を、filename で指定したファイルに直列化（保存）する。このインタフェースは集団操作であり、全てのランクで同時にコールされなければならない。

直列化されたファイルには、コミュニケータに属するすべてのランクの乱数内部状態が記録されている。実際の直列化処理はランク 0 のプロセスが担当し、1 つのコミュニケータに対して 1 つのファイルが作成されることになる。

ファイルはバイナリ形式で、フォーマットは以下 (図 3.2) のとおりである。また、C、C++および Fortran90 用インタフェースそれぞれの間で互換性がある。

3.7 KMATH_Random_Deserialize

C Syntax

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

最大ランク数	4 バイト
ランク 0 の初期乱数内部状態	3080 バイト
ランク 1 の初期乱数内部状態	3080 バイト
⋮	
ランク N-1 の初期乱数内部状態	3080 バイト

図 3.2: ジャンプファイル内部フォーマット

引数	型	IO	説明
handle	void*	Inout	ハンドル
filename	const char*	In	ファイル名
戻り値	int	Ret	エラー 0: エラーなし（正常） −1: 無効なハンドル −2: MPI 失敗 −3: ファイル I/O 失敗 −4: ランク数の不一致

C++ Syntax

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

引数	型	IO	説明
filename	const char*	In	ファイル名
戻り値	int	Ret	エラー 0: エラーなし（正常） −1: 無効なハンドル −2: MPI 失敗 −3: ファイル I/O 失敗 −4: ランク数の不一致

Fortran90 Syntax

```
use kmath_random_mod
subroutine KMATH_Random_Deserialize(handle, filename, ierr)
```

引数	型	IO	説明
handle	type(s_km_rand)	Inout	ハンドル
filename	character(*)	In	ファイル名
ierr	integer	Out	エラー 0: エラーなし（正常） −1: 無効なハンドル −2: MPI 失敗 −3: ファイル I/O 失敗 −4: ランク数の不一致

直列化されたファイルを読み込み、乱数内部状態を復元する。このインタフェースは、直列化同様に集団操作であり全てのランクで同時にコールされなければならない。

現在のコミュニケータに属するランク数と、ファイル作成時のランク数が異なる場合エラーとなり、エラー番号 −4 が返却される。

読み込むファイルの書式については、前節を参照のこと。

3.8　環境変数: KMATH_RAND_JUMP_FILE_PATH

ジャンプファイルの参照パスを指定する。この環境変数が設定されていない場合、以下がデフォルト参照パスとなる。

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

3.9　環境変数: KMATH_RAND_JUMP_FILE_PREFIX

ジャンプファイルのプレフィックスを指定する。この環境変数が設定されていない場合、以下がデフォルトプレフィックスとなる。

```
file
```

実際のジャンプファイルは file_00001, file_00002, ... の様に、シードの種分だけの ID (整数値) が付与されて管理される。

第4章　動作手順

KMATH_RANDOM の処理の流れは、第一章でも示したように図 4.1 のようになる。

1. ジャンプファイルの生成
2. ジャンプファイルのインストール
3. KMATH_RANDOM を使用するプログラムの起動
4. ジャンプファイル格納箇所 (環境変数 `KMATH_RAND_JUMP_FILE_PATH`) の参照
5. `KMATH_Random_Init` による初期化、もしくは `KMATH_Random_Deserialize` でジャンプファイルから内部状態の回復
6. `KMATH_Random_Get` などで各プロセス独立に乱数を取得
7. KMATH_RANDOM のファイナライズ

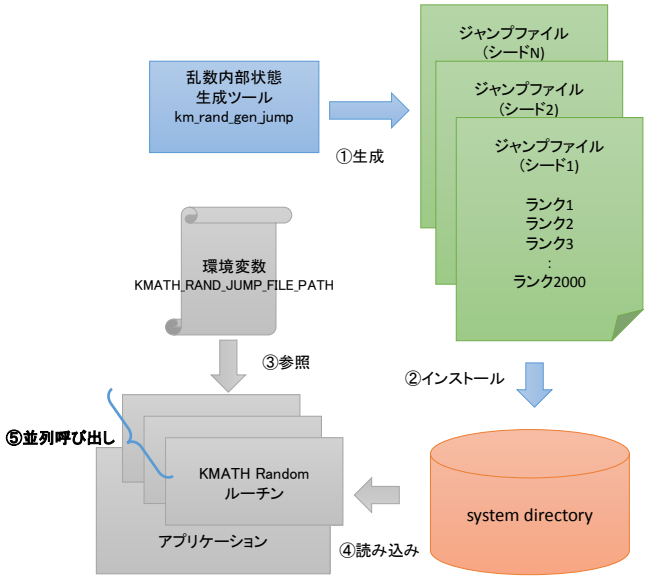

図 4.1: KMATH_RANDOM の処理の流れ

4.1　ジャンプファイルの作成

本節では、ルーチン初期化時に各ランクが乱数内部状態を復元するためのジャンプファイルを作成方法について説明する。

4.1.1　フロントエンド/バックエンド向けツールのビルド

まず、乱数内部状態生成ツール (以下、ジャンプツールと呼ぶ) のビルドはフロントエンド用とバックエンド用で異なり、フロントエンド向けツールは `make tool` によって作成し、バックエンド向けツールは `make ptool` によって作成する。なお、実際にはソースコードはほぼ同じであり、バックエンド側は MPI コンパイラでなくてはならないなど、コンパイラを区別して作成できる点が異なるにすぎない。

フロントエンド向けジャンプツールのビルド

```
$ cd <kmath random root path>
$ make tool
$ cd tool
$ ls -1
Makefile
dSFMT-calc-jump.hpp
dSFMT-jump.cpp
dSFMT-jump.h
dSFMT-jump.o
gen.sh
jump
km_dsfmt_jump.cpp
km_dsfmt_jump.h
km_dsfmt_jump.o

:

$
```

4.2　ジャンプツールの実行

前節でビルドされるツールは、ジャンプファイルを作成する `km_rand_gen_jump` と作成されたジャンプファイルの検証をおこなう `km_rand_chk_jump` の 2 つである。以下にその用例を示す。

4.2.1 km_rand_gen_jump

ジャンプファイルを作成するツールである。指定したシード値の範囲の数だけジャンプファイルが作成される。書式は以下のとおりである。

./km_rand_gen_jump [パラメータ 1 [パラメータ 2]..]

指定可能なパラメータは以下のとおりである。

-seed <seed_start> <seed_end>

シード値の範囲。このシード値の数だけ、ジャンプファイルが作成される。デフォルト値は、seed_start=1, seed_end=1 である。

-max_ranks <rank>

ランク数の最大値。乱数生成時、コミュニケータ内のランクサイズは、この値以下でなくてはならない。デフォルト値は 1。

-rand_range <range>

1 ランクあたりの乱数生成範囲。2^{range} として指定する。デフォルト値は 100。

-install_dir <path>

作成するジャンプファイルの保存先。デフォルト値は./jump。

-file_prefix <prefix>

作成するジャンプファイルのプレフィックス。実際のジャンプファイル名は、このプレフィックスの後にシード番号が付与される。デフォルト値は file である。

4.2.2 km_rand_chk_jump

作成されたジャンプファイルの検証をおこなうツールである。本ツールは、どのインタフェースも介さず直接 dSFMT を呼び出す。ジャンプファイルに記録されたすべてのランク数分の乱数内部状態を構造体 dsfmt_t に復元し、乱数を 1 つ発生させ標準出力に値を出力する。

書式は以下のとおりである。

./km_rand_chk_jump < ジャンプファイル >

4.2.3 km_rand_get_gen_jump の実行例 (フロントエンド用)

ジャンプファイルの作成

```
$ 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
$
```

作成されたファイル file_00001, file_00002, ... を乱数生成器初期化に利用する。ファイル情報は以下のとおりである。

シード値の範囲	$1 \sim 10$
最大ランク数	2000
1 ランクあたりの乱数生成範囲	2^{100}
ジャンプファイルの保存先	./jump
ジャンプファイルのプレフィックス	file

KMATH_Random_Get や KMATH_Random_Vector で乱数を生成していくと、このファイルを読み込みメモリ上で管理された乱数内部状態が変化していく。従って、KMATH_Random_Serialize によって直列化し新たにファイル書き込まれた状態は、このジャンプツールが作成した初期内部状態とは異なるので注意が必要である。初期状態から再現を試みたり、スナップショットを再現したい場合にはジャンプファイルをその都度保存して、KMATH_Random_Deserialize にそれらを渡す必要がある。

なお、dSFMT で SIMD 命令を使用するかしないかの設定を変更した場合や乱数周期を変更した場合も、ジャンプファイルを作成しなおす必要がある。

4.2.4 km_rand_get_gen_jump の実行例 (バックエンド用バッチ処理)

ジャンプファイルをバックエンド側で生成するときは、バックエンド用のジャンプツールを使用する。ジャンプツールの用例はフロントエンド版と同様であるので、前節の内容を参考にして利用する。

バックエンド側で実行するにはキューシステムに投入して実行する必要がある。インタラクティブジョブの場合はフロントエンドと大きな差はないが、バッチジョブにて使用する際は、ptool/ 以下にバッチジョブ投入用のジョブスクリプトの例があるので以下の実行例を参考にされたい。

バッチジョブの実行

```
$ 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
$
```

4.3 ベンチマーク

本バージョン (1.1) のルーチンと、旧バージョン (1.0) のルーチンのベンチマークテストを行う。旧バージョンは、SIMD 命令実行が有効化されていない状態でビルドされている。それぞれ、10 億回の乱数を生成し、処理時間を計測する。

4.3.1　ベンチマークの作成からジョブ投入

ベンチマークプログラムの作成からジョブ投入まで

```
$ 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.

$
```

ビルドした実行ファイルを実行する。バッチジョブ投入用のシェルスクリプトがベンチマークプログラムのディレクトリ上に `run.sh` があるので、それを参考にされたい。なお、コマンドライン引数を一つ指定することができ、乱数種（シード）を変更することができる。

4.3.2 ベンチマーク結果の解析

上記のジョブ投入で、旧バージョン、新バージョン、それぞれ、1024 プロセス並列でテストプログラムを実行し乱数を生成する。以下、それらの結果の比較について解説する。ログには、KMATH, Rand, Diff, First, Last の 5 項目の表示がある。それらは以下の情報を意味している。

KMATH	`KMATH_Init` 呼び出しから、`KMATH_Finalize` 呼び出しまでの時間
Rand	`KMATH_Random_get` ルーチンを 10 億回呼び出す時間
Diff	初期化と破棄に要した時間（ = KMATH - Rand）
First	最初の乱数値
Last	最後の乱数値（10 億回目）

計算時間

実行結果の解析例では、ルーチン初期化時間でソートし、最も時間を要したランクが一番下に表示されている。旧バージョンと比較して、処理時間が若干速いことが確認できる。これは、新バージョンでは初期化時のジャンプ操作などの初期化のオーバヘッドがなくなったためと考えられる。

ファイル IO 時間

ただし、ジャンプファイルの読み込み時間はランク数（つまり、ジャンプファイルのサイズ）に依存しており、多数の計算ノードでの初期化には時間を要することになる。京コンピュータでの実験から概算される値ではあるが、10000 ランク分の乱数内部状態が記録されたジャンプファイルは約 3MB に達する。ファイルサイズ 3MB 程度のジャンプファイルをステージインして読み込ませた場合、各ランクでファイルオープン（`fopen`）に 0.35 秒程度、ファイルリード（`fread`）に 0.2〜0.4 秒程度の処理時間を要する。尚、上記ベンチマークを実施した際のジャンプファイルのサイズは、6032KB である。

SIMD 性能

また、システムログを比較することで SIMD 命令の実行回数を確認できる。もともとコンパイラによる SIMD 並列化がなされており、旧バージョンでも SIMD 命令が発行されている。現バージョンでは 20〜30%程度の SIMD 命令発行数が増加しており、実行時間短縮に寄与していると考えられる。

ベンチマークプログラム実行結果

```
$ 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: 1.954946 | Last: 1.241992
Rank00860: KMATH: 19.021160 | Rand: 18.464068 | Diff: 0.557092 | 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
Rank00671: KMATH: 19.015893 | Rand: 18.464410 | Diff: 0.551483 | First: 1.394975 | Last: 1.487162
$ cat kmath_random_v1.0/__comparison/run.sh.o2246379 | sort -k6 |tail
Rank00515: KMATH: 20.874865 | Rand: 18.937350 | Diff: 1.937515 | First: 1.384332 | Last: 1.185851
Rank00003: KMATH: 20.891247 | Rand: 18.937434 | Diff: 1.953813 | First: 1.914003 | Last: 1.897960
Rank00120: KMATH: 20.876069 | Rand: 18.937446 | Diff: 1.938623 | First: 1.113646 | Last: 1.811843
Rank00903: KMATH: 20.882967 | Rand: 18.937597 | Diff: 1.945370 | First: 1.486338 | Last: 1.852805
Rank00532: KMATH: 20.861776 | Rand: 18.937662 | Diff: 1.924114 | First: 1.176010 | Last: 1.982283
Rank00448: KMATH: 20.900649 | Rand: 18.937692 | Diff: 1.962957 | First: 1.971146 | Last: 1.862640
Rank00177: KMATH: 20.880725 | Rand: 18.937693 | Diff: 1.943032 | First: 1.366891 | Last: 1.543183
Rank00852: KMATH: 20.851202 | Rand: 18.937843 | Diff: 1.913359 | First: 1.457589 | Last: 1.216967
Rank00508: KMATH: 20.897802 | Rand: 18.938084 | Diff: 1.959718 | First: 1.856374 | Last: 1.194903
Rank00334: KMATH: 20.875466 | Rand: 18.939159 | Diff: 1.936307 | First: 1.736941 | Last: 1.230808
$ 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
Rank00838: KMATH: 19.074573 | Rand: 18.463469 | Diff: 0.611104 | First: 1.332175 | Last: 1.534236
Rank00916: KMATH: 19.051415 | Rand: 18.440010 | Diff: 0.611405 | First: 1.550704 | Last: 1.216761
Rank00891: KMATH: 19.073662 | Rand: 18.462248 | Diff: 0.611414 | First: 1.629879 | Last: 1.243444
Rank00969: KMATH: 19.054204 | Rand: 18.442119 | Diff: 0.612085 | First: 1.478234 | Last: 1.548668
Rank00761: KMATH: 19.076265 | Rand: 18.460603 | Diff: 0.615662 | First: 1.482196 | Last: 1.492482
Rank00759: KMATH: 19.080420 | Rand: 18.461416 | Diff: 0.619004 | First: 1.825657 | Last: 1.428102
Rank00754: KMATH: 19.061866 | Rand: 18.441747 | Diff: 0.620119 | First: 1.044430 | Last: 1.346183
Rank00776: KMATH: 19.056881 | Rand: 18.435444 | Diff: 0.621437 | First: 1.037025 | Last: 1.647820
$ cat kmath_random_v1.0/__comparison/run.sh.o2246379 | sort -k9 |tail
Rank00204: KMATH: 20.908349 | Rand: 18.933447 | Diff: 1.974902 | First: 1.964720 | Last: 1.836223
Rank00832: KMATH: 20.908210 | Rand: 18.932970 | Diff: 1.975240 | First: 1.906966 | Last: 1.504693
Rank00207: KMATH: 20.909952 | Rand: 18.934280 | Diff: 1.975672 | First: 1.516986 | Last: 1.625560
Rank00190: KMATH: 20.910559 | Rand: 18.934203 | Diff: 1.976356 | First: 1.281674 | Last: 1.643301
Rank00147: KMATH: 20.911662 | Rand: 18.934374 | Diff: 1.977288 | First: 1.624243 | Last: 1.016437
Rank00167: KMATH: 20.912492 | Rand: 18.935168 | Diff: 1.977324 | First: 1.525658 | Last: 1.731941
Rank00405: KMATH: 20.912696 | Rand: 18.934336 | Diff: 1.978360 | First: 1.604427 | Last: 1.056995
Rank00165: KMATH: 20.914527 | Rand: 18.935986 | Diff: 1.978541 | First: 1.380315 | Last: 1.031952
Rank00967: KMATH: 20.917316 | Rand: 18.935278 | Diff: 1.982038 | First: 1.779593 | Last: 1.610631
Rank00953: KMATH: 20.916208 | Rand: 18.933213 | Diff: 1.982995 | First: 1.444039 | Last: 1.926610
$ tail -n 5 run.sh.i2246378
 DISK SIZE (USE)           : -
 I/O SIZE                  : 8982.3 MB (8982214990)
 FILE I/O SIZE             : 8943.3 MB (8943265297)
 EXEC INST NUM             : 38518205469617
 EXEC SIMD NUM             : 2156386352
$ tail kmath_random_v1.0/__comparison/run.sh.i2246379
 DISK SIZE (USE)           : -
 I/O SIZE                  : 392.4 MB (392336626)
 FILE I/O SIZE             : 353.4 MB (353387891)
 EXEC INST NUM             : 46965170405425
 EXEC SIMD NUM             : 1896914464
$
```

第5章　結び

5.1　KMATH_RANDOMの現在と今後

現在のKMATH_RANDOMには、以下に述べる制約や追加が検討される機能があり、これらは今後改善、実現される可能性がある。

第1に、第3章で述べたとおり、現在のKMATH_RANDOMはスレッドセーフではないという問題がある。このため、ノード内において並列実行する場合であっても複数のMPIプロセスを生成する必要が生じている。

第2に、乱数列の周期がKMATH_RANDOMのビルド時に決定され、動的に変更できないという制約がある。このため、複数の乱数周期を使い分けたいユーザは、必要な周期の数だけライブラリをビルドしなければならない。これを改善し、実行時に任意に周期を選択する機能の追加が検討されている。

第3に、現在はdSFMTに固定されている乱数生成アルゴリズムを、任意の乱数生成ライブラリに置き換えられる機能の追加が検討されている。

5.2　謝辞

本ユーザマニュアル掲載の実行結果は、理化学研究所のスーパーコンピュータ「京」を利用して得られたものである。

関連図書

[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/