DD-AVX User's Manual

バージョン 1.0.0 (based on Lis 1.4.58)

Toshiaki Hishinuma[a], Akihiro Fujii[a], Teruo Tanaka[a], Hidehiko Hasegawa[b]

http://www.slis.tsukuba.ac.jp/∼s1530534/DD-AVX.html

[a]Kogakuin University, [b]University of Tsukuba

表紙: Warsaw, Poland by Toshiaki Hishinuma (2013).

目 次

1 はじめに

“DD-AVX” は，高速な倍々精度反復解法に向けた倍々精度演算ライブラリである．
反復解法ライブラリ Lis[1](http://www.ssisc.org/lis/) と組み合わせて使用することを想定している，Lis に含まれる，

- 倍々精度ベクトル演算
- 倍精度疎行列と倍々精度ベクトルの積 (転置含む)

を SIMD 拡張命令 AVX, AVX2 を用いて高速化したルーチン群 [86] からなり，Lis とマージすることで AVX, AVX2 を用いた倍々精度反復解法を解くことを想定している．

また，我々が明らかにした以下の 3 つの倍々精度の高速化技法も追加機能として実装した．

- AVX に適した疎行列の格納形式 BSR[87](AVX 対応は 4x1 のみ)
- OpenMP のスレッド負荷分散スケジューリング方式 balanced[88]
- OpenMP のスケジューリング方式の自動選択 [88]

これらは．2.6.1 DD-AVX_test1 と，4 章の 4 倍精度演算に利用方法を記載した．

また，各関数の使い方は別紙の Function Reference を参照のこと．DD-AVX によって追加された 2 つの関数は，Funtion Reference の 4.9 節に記載した．

本ライブラリはマージ後の使い方は Lis と同じであるように設計したため，Lis を用いて作られたプログラムは，変更せずに利用することが可能である．
本ライブラリは，Lis の User’s Manual をベースに，DD-AVX の使用方法について述べる．

2 導入

2.1 システム要件

本章では，AVX，AVX2 を用いた動作確認環境について述べる．
DD-AVX の導入には C コンパイラが必要である．また，DD-AVX は，UNIX, Linux オペレーティングシステム以外は動作未確認である．

並列計算環境では, OpenMP ライブラリまたは MPI ライブラリを使用する．
SIMD 拡張命令 AVX，AVX2 に対応しているかは，導入環境の “/proc/cpuinfo” およびコンパイラの対応状況の確認が必要である．

表 1 に DD-AVX の AVX での動作確認環境を示す.

表 1: DD-AVX の動作確認環境 (AVX)

CPU (Code name)	C Compiler	OS
Intel Core i7 2600 K (Sandy Bridge)	Intel C/C++ Compiler 12.0.3, 13.1.3, 15.0.0 gcc 4.6.3, 4.8.3	Fedora 16
Intel Core i7 3770 K (Ivy Bridge)	Intel C/C++ Compiler 12.0.3, 13.1.3, 15.0.0 gcc 4.6.3, 4.8.3	Cent OS 6.4
Intel Core i7 4770 (Haswell)	Intel C/C++ Compiler 12.0.3, 13.1.3, 15.0.0 gcc 4.6.3, 4.8.3	Cent OS 6.4
Intel Core i7 4690 S (Haswell Refresh)	Intel C/C++ Compiler 12.0.3, 13.1.3, 15.0.0 gcc 4.6.3, 4.8.3	Cygwin on Windows 7

表 2 に DD-AVX の AVX2 での動作確認環境を示す.

表 2: DD-AVX の動作確認環境 (AVX2)

CPU (Code name)	C Compiler	OS
Intel Core i7 4770 (Haswell)	Intel C/C++ Compiler 12.0.3, 13.1.3, 15.0.0 gcc 4.8.3	Cent OS 6.4
Intel Core i7 4690 S (Haswell Refresh)	Intel C/C++ Compiler 12.0.3, 13.1.3, 15.0.0 gcc 4.8.3	Cygwin on Windows 7

2.2 ライブラリのダウンロード・展開

本章では，Lis, DD-AVX を Web ページからダウンロードし，展開する方法を述べる．
本ライブラリは，反復解法ライブラリ Lis(ver.1.4.58) に対し，DD-AVX ライブラリをマージして使用することを想定している．

はじめに，Lis と DD-AVX を同一のフォルダにダウンロードする．
ダウンロードをコマンドライン上において行うためには，

- Lis1.4.58 のダウンロード

```
> wget http://www.ssisc.org/lis/dl/lis-1.4.58.tar.gz
```

- DD-AVX のダウンロード

```
> wget http://www.slis.tsukuba.ac.jp/~s1530534/DD-AVX-files/DD-AVX1.2.0.tar.gz
```

を用いる.

次に，これらのアーカイブの展開を行うために，コマンドライン上において，

- Lis-1.4.58 の展開

```
> gunzip -c lis-1.4.58.tar.gz | tar -xv
```

- DD-AVX の展開

```
> gunzip -c DD-AVX1.0.0.tar.gz | tar -xv
```

を行う.

これにより，以下の 3 つが同様のフォルダに展開される.

1. DD-AVX/
2. Lis-1.4.58/
3. DD-AVX-merge.sh

また，これ以降 tar.gz ファイルは必要ないので，不要であれば削除しても構わない.
ディレクトリ DD-AVX は，図 1 に示す DD-AVX のサブディレクトリから成る.

2.3 DD-AVX と Lis のマージ

本章では， 展開した lis-1.4.58 と DD-AVX のマージ方法について述べる.
マージには，DD-AVX-merge.sh を用いる．このスクリプトの引数は [lis のディレクトリ], [DD-AVX のディレクトリ], [マージ後の出力ディレクトリ] である.
以下のコマンドを実行し，Lis と DD-AVX をマージする.

```
> sh ./DD-AVX-merge.sh lis-1.4.58/ [output_dir]
```

これにより, lis-1.4.58 と DD-AVX をマージした [output_dir] が得られる.

2.4 Configure

Configure と，そのオプションについて述べる.
“–enable-avx”，“–enable-avx2” が DD-AVX から新たに追加されたオプションである.

ディレクトリ [output_dir] において次のコマンドを実行し, Configure を行う.
これにより，ユーザが定義した環境向けの Makefile が生成される.

```
DD-AVX
 | Configure, Configure.in
 + config
 |  設定ファイル
 + doc
 |  マニュアル
 + include
 |  ヘッダファイル
 + src
 | + system
 | | AVX, AVX2 に対応したアラインメントを考慮したメモリ確保関数
 | + precision
 | | 倍々精度演算カーネル
 | + matvec
 | | 各格納形式に対する疎行列ベクトル積の切り替えインタフェース
 + test
 |  検証プログラム，実行例スクリプト
```

図 1: DD-AVX-($VERSION).tar.gz のファイル構成

Configure は，以下のコマンドによって行う． 導入先を<install-dir>としたとき

```
> ./configure --prefix=<install-dir> [option] とする.
```

例えば，OpenMP と MPI のハイブリッド並列で 4 倍精度反復解法を AVX2 を用いて，mpicc のオプションに-static intel を付けたいとき，以下のようにする．

```
> ./configure --prefix=<install-dir> --enable-omp --enable-mpi
  --enable-quad --enable-avx2 CFLAGS=-static\ intel
```

本ライブラリを使うためのオプションを表 3 に示す．表 4 に TARGET として指定できる主な計算機環境を示す．
なお，SSE2, AVX, AVX2 は倍々精度精度演算 (“–enable quad”) でしか利用できない．SIMD の設定は，Configure 時に設定し，以降は再度 Configure するまで変更できない．

2.5 実行ファイルの生成・導入

本章では，実行ファイルの生成を行い，ライブラリを導入する方法を述べる．

2.5.1 実行ファイルの生成

ディレクトリ [output_dir] において次のコマンドを入力し, 実行ファイルを生成する．

```
> make
```

実行ファイルが正常に生成されたかどうかを確認するには, のコマンドを入力し, ディレクトリ [putput_dir]/test

表 3: Configure 時のオプション一覧 (`./configure --help` からも参照可能)

`--enable-omp`	OpenMP ライブラリを使用
`--enable-mpi`	MPI ライブラリを使用
`--enable-fortran`	FORTRAN 77 互換インタフェースを使用
`--enable-f90`	Fortran 90 互換インタフェースを使用
`--enable-saamg`	SA-AMG 前処理を使用
`--enable-quad`	倍々精度演算を使用
`--enable-longdouble`	long double 型 4 倍精度演算を使用
`--enable-longlong`	64 ビット整数型を使用
`--enable-sse2`	SIMD 拡張命令 SSE2 を使用
`--enable-avx`	SIMD 拡張命令 AVX を使用
`--enable-avx2`	SIMD 拡張命令 AVX2 を使用
`--enable-debug`	デバッグモードを使用
`--enable-shared`	動的リンクを使用
`--enable-gprof`	プロファイラを使用
`--prefix=<install-dir>`	導入先を指定
`TARGET=<target>`	計算機環境を指定
`CC=<c_compiler>`	C コンパイラを指定
`CFLAGS=<c_flags>`	C コンパイラオプションを指定
`F77=<f77_compiler>`	FORTRAN 77 コンパイラを指定
`F77FLAGS=<f77_flags>`	FORTRAN 77 コンパイラオプションを指定
`FC=<f90_compiler>`	Fortran 90 コンパイラを指定
`FCFLAGS=<f90_flags>`	Fortran90 コンパイラオプションを指定
`LDFLAGS=<ld_flags>`	リンクオプションを指定

表 4: TARGET の一覧 (詳細は lis-($VERSION)/configure.in を参照)

<target>	等価なオプション
cray_xt3_cross	./configure CC=cc FC=ftn CFLAGS="-O3 -B -fastsse -tp k8-64" FCFLAGS="-O3 -fastsse -tp k8-64 -Mpreprocess" FCLDFLAGS="-Mnomain" ac_cv_sizeof_void_p=8 cross_compiling=yes ax_f77_mangling="lower case, no underscore, extra underscore"
fujitsu_fx10_cross	./configure CC=fccpx FC=frtpx CFLAGS="-Kfast,ocl,preex" FCFLAGS="-Kfast,ocl,preex -Cpp -fs" FCLDFLAGS="-mlcmain=main" ac_cv_sizeof_void_p=8 cross_compiling=yes ax_f77_mangling="lower case, underscore, no extra underscore"
hitachi_sr16k	./configure CC=cc FC=f90 CFLAGS="-Os -noparallel" FCFLAGS="-Oss -noparallel" FCLDFLAGS="-lf90s" ac_cv_sizeof_void_p=8 ax_f77_mangling="lower case, underscore, no extra underscore"
ibm_bgl_cross	./configure CC=blrts_xlc FC=blrts_xlf90 CFLAGS="-O3 -qarch=440d -qtune=440 -qstrict" FCFLAGS="-O3 -qarch=440d -qtune=440 -qsuffix=cpp=F90" ac_cv_sizeof_void_p=4 cross_compiling=yes ax_f77_mangling="lower case, no underscore, no extra underscore"
nec_sx9_cross	./configure CC=sxmpic++ FC=sxmpif90 AR=sxar RANLIB=true ac_cv_sizeof_void_p=8 ax_vector_machine=yes cross_compiling=yes ax_f77_mangling="lower case, no underscore, extra underscore"
DD-AVX_cross	./configure CC=icc --enable-omp --enable-mpi --enable-avx
DD-AVX2_cross	./configure CC=icc --enable-omp --enable-mpi --enable-avx2
DD-AVX_MPI_cross	./configure CC=icc --enable-omp --enable-mpi --enable-avx --enable-mpi
DD-AVX2_MPI_cross	./configure CC=icc --enable-omp --enable-mpi --enable-avx2 --enable-mpi

に生成された実行ファイルを用いて検証を行う.

```
> make check
```

このコマンドでは, Matrix Market 形式のファイル `test/testmat.mtx` から行列, ベクトルデータを読み込み, BiCG 法を用いて線型方程式 $Ax = b$ の解を求める. 以下に SGI Altix 3700 上での実行結果を示す. なおオプション`--enable-omp` と`--enable-mpi` は組み合わせて使用することができる.

既定

```
matrix size = 100 x 100 (460 nonzero entries)
initial vector x = 0
precision : double
solver    : BiCG 2
precon    : none
conv_cond : ||b-Ax||_2 <= 1.0e-12 * ||b-Ax_0||_2
storage   : CSR
lis_solve : normal end

BiCG: number of iterations = 15 (double = 15, quad = 0)
BiCG: elapsed time         = 5.178690e-03 sec.
BiCG:   preconditioner     = 1.277685e-03 sec.
BiCG:     matrix creation  = 1.254797e-03 sec.
BiCG:   linear solver      = 3.901005e-03 sec.
BiCG: relative residual    = 6.327297e-15
```

--enable-omp

```
max number of threads = 32
number of threads = 2
matrix size = 100 x 100 (460 nonzero entries)
initial vector x = 0
precision : double
solver    : BiCG 2
precon    : none
conv_cond : ||b-Ax||_2 <= 1.0e-12 * ||b-Ax_0||_2
storage   : CSR
lis_solve : normal end

BiCG: number of iterations = 15 (double = 15, quad = 0)
BiCG: elapsed time         = 8.960009e-03 sec.
BiCG:   preconditioner     = 2.297878e-03 sec.
BiCG:     matrix creation  = 2.072096e-03 sec.
BiCG:   linear solver      = 6.662130e-03 sec.
BiCG: relative residual    = 6.221213e-15
```

--enable-mpi

```
number of processes = 2
matrix size = 100 x 100 (460 nonzero entries)
initial vector x = 0
precision : double
solver    : BiCG 2
precon    : none
conv_cond : ||b-Ax||_2 <= 1.0e-12 * ||b-Ax_0||_2
storage   : CSR
lis_solve : normal end

BiCG: number of iterations = 15 (double = 15, quad = 0)
BiCG: elapsed time         = 2.911400e-03 sec.
BiCG:   preconditioner     = 1.560780e-04 sec.
BiCG:     matrix creation  = 1.459997e-04 sec.
BiCG:   linear solver      = 2.755322e-03 sec.
BiCG: relative residual    = 6.221213e-15
```

2.5.2 導入

ディレクトリ [output_dir] において次のコマンドを入力し, 導入先のディレクトリにファイルを複製する.

```
> make install
```

これにより, ディレクトリ<install-dir>に以下のファイルが複製される.

```
($INSTALLDIR)
 +bin
 |   +lsolve esolve hpcg_kernel hpcg_spmvtest spmvtest*
 +include
 |   +lis_config.h lis.h lisf.h
 +lib
 |   +liblis.a
 +share
     +doc/lis examples/lis man
```

lis_config.h はライブラリを生成する際に, また lis.h は C, lisf.h は Fortran でライブラリを使用する際に必要なヘッダファイルである. liblis.a は生成されたライブラリファイルである. ライブラリファイルが正常に導入されたかどうかを確認するには, ディレクトリ lis-($VERSION) において次のコマンドを入力し, ディレクトリ examples/lis に生成された実行ファイルを用いて検証を行う.

```
> make installcheck
```

examples/lis 下の test1, etest5, test3b, spmvtest3b は, lsolve, esolve, hpcg_kernel, hpcg_spmvtest の別名で ($INSTALLDIR)/bin に複製される. examples/lis/spmvtest*も, それぞれ ($INSTALLDIR)/bin に複製される.

2.6 検証

2.6.1 DD-AVX_test1

サンプルプログラムは，[output_dir]/test にある．

DD-AVX_test1.c は，DD-AVX から追加されたサンプルプログラムで，Lis の test1.c を DD-AVX 向けにしたものである．このプログラムは，4 倍精度での利用を前提としている．

実行するには，ディレクトリ [output_dir]/test において

```
> ./DD-AVX_test1 matrix_filename rhs_setting solution_filename rhistory_filename [options]
```

とすることで，`matrix_filename` から行列データを読み込み, 線型方程式 $Ax = b$ を `options` で指定された解法で解く．このとき，格納形式はブロックサイズ 4x1 の BCRS 形式になる．

また，コメントアウトを外せば，OpenMP のスケジューリング方式を自動で選択する機能も使用できる．

解を拡張 Matrix Market 形式で `solution_filename` に, 残差履歴を PLAIN 形式で `rhistory_filename` に書き出す (付録 A を参照)．入力可能な行列データ形式は Matrix Market 形式, 拡張 Matrix Market 形式である．`rhs_setting` には

0	行列データファイルに含まれる右辺ベクトルを用いる
1	$b = (1, \ldots, 1)^T$ を用いる
2	$b = A \times (1, \ldots, 1)^T$ を用いる
rhs_filename	右辺ベクトルのファイル名

のいずれかを指定できる．`rhs_filename` は PLAIN 形式, Matrix Market 形式に対応する．`test1f.F` は `test1.c` の Fortran 版である．

2.6.2 test1

ディレクトリ [output_dir]/test において

```
> ./test1 matrix_filename rhs_setting solution_filename rhistory_filename [options]
```

と入力すると, `matrix_filename` から行列データを読み込み, 線型方程式 $Ax = b$ を `options` で指定された解法で解く．また, 解を拡張 Matrix Market 形式で `solution_filename` に, 残差履歴を PLAIN 形式で `rhistory_filename` に書き出す (付録 A を参照)．入力可能な行列データ形式は Matrix Market 形式, 拡張 Matrix Market 形式である．`rhs_setting` には

0	行列データファイルに含まれる右辺ベクトルを用いる
1	$b = (1, \ldots, 1)^T$ を用いる
2	$b = A \times (1, \ldots, 1)^T$ を用いる
rhs_filename	右辺ベクトルのファイル名

のいずれかを指定できる．`rhs_filename` は PLAIN 形式, Matrix Market 形式に対応する．`test1f.F` は `test1.c` の Fortran 版である．

2.6.3 test2

ディレクトリ [output_dir]/test において

```
> test2 m n matrix_type solution_filename rhistory_filename [options]
```

と入力すると, 2 次元 Poisson 方程式を 5 点中心差分で離散化して得られる次数 mn の 5 重対角行列を係数とする線型方程式 $Ax = b$ を, matrix_type で指定された行列格納形式, options で指定された解法で解く. また, 解を拡張 Matrix Market 形式で solution_filename に, 残差履歴を PLAIN 形式で rhistory_filename に書き出す. ただし, 右辺ベクトル b は線型方程式 $Ax = b$ の解ベクトルの値がすべて 1 となるよう設定される. m, n は各次元の格子点数である. test2f.F90 は test2.c の Fortran90 版である.

2.6.4 test2b

ディレクトリ [output_dir]/test において

```
> test2b m n matrix_type solution_filename rhistory_filename [options]
```

と入力すると, 2 次元 Poisson 方程式を 9 点中心差分で離散化して得られる次数 mn の 9 重対角行列を係数とする線型方程式 $Ax = b$ を, matrix_type で指定された行列格納形式, options で指定された解法で解く. また, 解を拡張 Matrix Market 形式で solution_filename に, 残差履歴を PLAIN 形式で rhistory_filename に書き出す. ただし, 右辺ベクトル b は線型方程式 $Ax = b$ の解ベクトルの値がすべて 1 となるよう設定される. m, n は各次元の格子点数である.

2.6.5 test3

ディレクトリ [output_dir]/test において

```
> test3 l m n matrix_type solution_filename rhistory_filename [options]
```

と入力すると, 3 次元 Poisson 方程式を 7 点中心差分で離散化して得られる次数 lmn の 7 重対角行列を係数とする線型方程式 $Ax = b$ を, matrix_type で指定された行列格納形式, options で指定された解法で解く. また, 解を拡張 Matrix Market 形式で solution_filename に, 残差履歴を PLAIN 形式で rhistory_filename に書き出す. ただし, 右辺ベクトル b は線型方程式 $Ax = b$ の解ベクトルの値がすべて 1 となるよう設定される. l, m, n は各次元の格子点数である.

2.6.6 test3b

ディレクトリ [output_dir]/test において

```
> test3b l m n matrix_type solution_filename rhistory_filename [options]
```

と入力すると, 3 次元 Poisson 方程式を 27 点中心差分で離散化して得られる次数 lmn の 27 重対角行列を係数とする線型方程式 $Ax = b$ を, matrix_type で指定された行列格納形式, options で指定された解法で解く. また, 解を拡張 Matrix Market 形式で solution_filename に, 残差履歴を PLAIN 形式で rhistory_filename に書き出す. ただし, 右辺ベクトル b は線型方程式 $Ax = b$ の解ベクトルの値がすべて 1 となるよう設定される. l, m, n は各次元の格子点数である.

2.6.7 test4

線型方程式 $Ax = b$ を指定された解法で解き, 解を標準出力に書き出す. 行列 A は次数 12 の 3 重対角行列

$$A = \begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & -1 & 2 & -1 & \\ & & & -1 & 2 & \end{pmatrix}$$

である. 右辺ベクトル b は解 x がすべて 1 となるよう設定される. `test4f.F` は `test4.c` の Fortran 版である.

2.6.8 test5

ディレクトリ [output_dir]/test において

```
> test5 n gamma [options]
```

と入力すると, 線型方程式 $Ax = b$ を指定された解法で解く. 行列 A は次数 n の Toepliz 行列

$$A = \begin{pmatrix} 2 & 1 & & & & & \\ 0 & 2 & 1 & & & & \\ \gamma & 0 & 2 & 1 & & & \\ & \ddots & \ddots & \ddots & \ddots & & \\ & & \gamma & 0 & 2 & 1 & \\ & & & \gamma & 0 & 2 & \end{pmatrix}$$

である. 右辺ベクトル b は解 x がすべて 1 となるよう設定される.

2.6.9 etest1

ディレクトリ [output_dir]/test において

```
> etest1 matrix_filename evector_filename rhistory_filename [options]
```

と入力すると, `matrix_filename` から行列データを読み込み, 固有値問題 $Ax = \lambda x$ を `options` で指定された解法で解いて, 指定された固有値を標準出力に書き出す. また, 対応する固有ベクトルを拡張 Matrix Market 形式で `evector_filename` に, 残差履歴を PLAIN 形式で `rhistory_filename` に書き出す. 入力可能な行列データ形式は Matrix Market 形式である. `etest1f.F` は `etest1.c` の Fortran 版である.

2.6.10 etest2

ディレクトリ [output_dir]/test において

```
> etest2 m n matrix_type evector_filename rhistory_filename [options]
```

と入力すると, 2 次元 Helmholtz 方程式を 5 点中心差分で離散化して得られる次数 mn の 5 重対角行列に関する固有値問題 $Ax = \lambda x$ を, `matrix_type` で指定された行列格納形式, `options` で指定された解法で解き, 指定された固有値を標準出力に書き出す. また, 対応する固有ベクトルを `evector_filename` に, 残差履歴を `rhistory_filename` に書き出す. `m`, `n` は各次元の格子点数である.

2.6.11 etest3

ディレクトリ [output_dir]/test において

```
> etest3 l m n matrix_type evector_filename rhistory_filename [options]
```

と入力すると, 3 次元 Helmholtz 方程式を 7 点中心差分で離散化して得られる次数 lmn の 7 重対角行列に関する固有値問題 $Ax = \lambda x$ を, `matrix_type` で指定された行列格納形式, `options` で指定された解法で解き, 指定された固有値を標準出力に書き出す. また, 対応する固有ベクトルを拡張 Matrix Market 形式で `evector_filename` に, 残差履歴を PLAIN 形式で `rhistory_filename` に書き出す. `l`, `m`, `n` は各次元の格子点数である.

2.6.12 etest4

ディレクトリ [output_dir]/test において

```
> etest4 n [options]
```

と入力すると, 固有値問題 $Ax = \lambda x$ を指定された解法で解き, 指定された固有値を標準出力に書き出す. 行列 A は次数 n の 3 重対角行列

$$A = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

である. `etest4f.F` は `etest4.c` の Fortran 版である.

2.6.13 etest5

ディレクトリ [output_dir]/test において

```
> etest5 matrix_filename evalues_filename evectors_filename residuals_filename
 iters_filename [options]
```

と入力すると, `matrix_filename` から行列データを読み込み, 固有値問題 $Ax = \lambda x$ を `options` で指定された解法で解いて, 指定された固有値を標準出力に書き出す. また, 指定された個数の固有値を `evalues_filename` に, 対応する固有ベクトル, 残差ノルム及び反復回数を `evectors_filename`, `residuals_filename` 及び `iters_filename` に拡張 Matrix Market 形式で書き出す. 入力可能な行列データ形式は Matrix Market 形式である.

2.6.14 etest6

ディレクトリ [output_dir]/test において

```
> etest6 l m n matrix_type evalues_filename evectors_filename residuals_filename
 iters_filename [options]
```

と入力すると, 3 次元 Helmholtz 方程式を 7 点中心差分で離散化して得られる次数 lmn の 7 重対角行列に関する固有値問題 $Ax = \lambda x$ を, `matrix_type` で指定された行列格納形式, `options` で指定された解法で解き, 指定された固有値を標準出力に書き出す. また, 指定された個数の固有値を `evalues_filename` に, 対応する固

有ベクトル, 残差ノルム及び反復回数を evectors_filename, residuals_filename 及び iters_filename に拡張 Matrix Market 形式で書き出す. l, m, n は各次元の格子点数である.

2.6.15 spmvtest1

ディレクトリ [output_dir]/test において

```
> spmvtest1 n iter [matrix_type]
```

と入力すると, 1 次元 Laplace 演算子を 3 点中心差分で離散化して得られる次数 n の 3 重対角係数行列

$$A = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix}$$

と $(1, \ldots, 1)^T$ との積を iter で指定された回数実行し, FLOPS 値を算出する. 必要なら matrix_type により,

0	実行可能なすべての行列格納形式について測定する
1-11	行列格納形式の番号

のいずれかを指定する.

2.6.16 spmvtest2

ディレクトリ [output_dir]/test において

```
> spmvtest2 m n iter [matrix_type]
```

と入力すると, 2 次元 Laplace 演算子を 5 点中心差分で離散化して得られる次数 mn の 5 重対角係数行列とベクトル $(1, \ldots, 1)^T$ との積を iter で指定された回数実行し, FLOPS 値を算出する. 必要なら matrix_type により,

0	実行可能なすべての行列格納形式について測定する
1-11	行列格納形式の番号

のいずれかを指定する. m, n は各次元の格子点数である.

2.6.17 spmvtest2b

ディレクトリ [output_dir]/test において

```
> spmvtest2b m n iter [matrix_type]
```

と入力すると, 2 次元 Laplace 演算子を 9 点中心差分で離散化して得られる次数 mn の 9 重対角係数行列とベクトル $(1, \ldots, 1)^T$ との積を iter で指定された回数実行し, FLOPS 値を算出する. 必要なら matrix_type により,

0	実行可能なすべての行列格納形式について測定する

1-11	行列格納形式の番号

のいずれかを指定する. m, n は各次元の格子点数である.

2.6.18 spmvtest3

ディレクトリ [output_dir]/test において

```
> spmvtest3 l m n iter [matrix_type]
```

と入力すると, 3 次元 Laplace 演算子を 7 点中心差分で離散化して得られる次数 lmn の 7 重対角係数行列とベクトル $(1, \ldots, 1)^T$ との積を iter で指定された回数実行し, FLOPS 値を算出する. 必要なら matrix_type により,

0	実行可能なすべての行列格納形式について測定する
1-11	行列格納形式の番号

のいずれかを指定する. l, m, n は各次元の格子点数である.

2.6.19 spmvtest3b

ディレクトリ [output_dir]/test において

```
> spmvtest3b l m n iter [matrix_type]
```

と入力すると, 3 次元 Laplace 演算子を 27 点中心差分で離散化して得られる次数 lmn の 27 重対角係数行列とベクトル $(1, \ldots, 1)^T$ との積を iter で指定された回数実行し, FLOPS 値を算出する. 必要なら matrix_type により,

0	実行可能なすべての行列格納形式について測定する
1-11	行列格納形式の番号

のいずれかを指定する. l, m, n は各次元の格子点数である.

2.6.20 spmvtest4

ディレクトリ [output_dir]/test において

```
> spmvtest4 matrix_filename_list iter [block]
```

と入力すると, matrix_filename_list の示す行列データファイルリストから行列データを読み込み, 各行列とベクトル $(1, \ldots, 1)^T$ との積を実行可能な行列格納形式について iter で指定された回数実行し, FLOPS 値を算出する. 入力可能な行列データ形式は Matrix Market 形式である. 必要なら block により, BSR, BSC 形式のブロックサイズを指定する.

2.6.21 spmvtest5

ディレクトリ [output_dir]/test において

```
> spmvtest5 matrix_filename matrix_type iter [block]
```

と入力すると, `matrix_filename` の示す行列データファイルから行列データを読み込み, 行列とベクトル $(1,\ldots,1)^T$ との積を行列格納形式 `matrix_type` について `iter` で指定された回数実行し, FLOPS 値を算出する. 入力可能な行列データ形式は Matrix Market 形式である. 必要なら `block` により, BSR, BSC 形式のブロックサイズを指定する.

2.7 制限事項

現バージョンには以下の制限がある.

- 行列格納形式
  - VBR 形式はマルチプロセス環境では使用できない.
  - CSR 形式以外の格納形式は SA-AMG 前処理では使用できない.
  - マルチプロセス環境において必要な配列を直接定義する場合は,
- double-double 型 4 倍精度演算
  - 線型方程式解法のうち, Jacobi, Gauss-Seidel, SOR, IDR(s) 法では使用できない.
  - 固有値解法のうち, CG, CR 法では使用できない.
  - Hybrid 前処理での内部反復解法のうち, Jacobi, Gauss-Seidel, SOR 法では使用できない.
  - I+S, SA-AMG 前処理では使用できない.
- long double 型 4 倍精度演算
  - Fortran インタフェースでは使用できない.
  - SA-AMG 前処理では使用できない.
- 前処理
  - Jacobi, SSOR 以外の前処理が選択され, かつ行列 A が CSR 形式でない場合, 前処理作成時に CSR 形式の行列 A が作成される.
  - 非対称線型方程式解法として BiCG 法が選択された場合, SA-AMG 前処理は使用できない.
  - SA-AMG 前処理はマルチスレッド計算には対応していない.
  - SAINV 前処理の前処理行列作成部分は逐次実行される.
- DD-AVX に関する制約
  - 倍精度演算では SIMD 拡張命令は使用できない (自動的に Scalar 命令で実行される).
  - 倍々精度演算では CSR, BSR 以外は使用できない.
  - BCRS 形式ではブロックサイズ 4x1 以外は AVX,AVX2 で使用できない (自動的に Scalar 命令で実行される).
  - OpenMP のスケジューリング方式に関する追加事項は倍精度では使用できない.

3 基本操作

本節では, ライブラリの使用方法について述べる. プログラムでは, 以下の処理を行う必要がある.

- 初期化処理
- 行列の作成
- ベクトルの作成
- ソルバ (解法の情報を格納する構造体) の作成
- 行列, ベクトルへの値の代入
- 解法の設定
- 求解
- 終了処理

また, プログラムの先頭には以下のコンパイラ指示文を記述しなければならない.

- C `#include "lis.h"`
- Fortran `#include "lisf.h"`

`lis.h`, `lisf.h` は, 導入時に `($INSTALLDIR)/include` 下に格納される.

3.1 初期化・終了処理

初期化, 終了処理は以下のように記述する. 初期化処理はプログラムの最初に, 終了処理は最後に実行しなければならない.

C

```
1: #include "lis.h"
2: LIS_INT main(LIS_INT argc, char* argv[])
3: {
4:     lis_initialize(&argc, &argv);
5:     ...
6:     lis_finalize();
7: }
```

Fortran

```
1: #include "lisf.h"
2:      call lis_initialize(ierr)
3:     ...
4:      call lis_finalize(ierr)
```

初期化処理

初期化処理を行うには, 関数

- C `LIS_INT lis_initialize(LIS_INT* argc, char** argv[])`

- Fortran subroutine lis_initialize(LIS_INTEGER ierr)

を用いる. この関数は, MPI の初期化, コマンドライン引数の取得等の初期化処理を行う.

LIS_INT の既定値 int は, プリプロセッサマクロ_LONGLONG が定義された場合には long long int に, また LIS_INTEGER の既定値 integer は, プリプロセッサマクロ LONGLONG が定義された場合には integer*8 に置き換えられる.

終了処理

終了処理を行うには, 関数

- C　　LIS_INT lis_finalize()
- Fortran subroutine lis_finalize(LIS_INTEGER ierr)

を用いる.

3.2　ベクトルの操作

ベクトル v の次数を $global\_n$ とする. ベクトル v を $nprocs$ 個のプロセスで行ブロック分割する場合の各部分ベクトルの行数を $local\_n$ とする. $global\_n$ が $nprocs$ で割り切れる場合は $local\_n = global\_n \ / \ nprocs$ となる. 例えば, ベクトル v を (3.1) 式のように 2 プロセスで行ブロック分割する場合, $global\_n$ と $local\_n$ はそれぞれ 4 と 2 となる.

$$v = \begin{pmatrix} 0 \\ 1 \\ \hline 2 \\ 3 \end{pmatrix} \begin{matrix} \text{PE0} \\ \\ \text{PE1} \\ \\ \end{matrix} \tag{3.1}$$

(3.1) 式のベクトル v を作成する場合, 逐次, マルチスレッド環境ではベクトル v そのものを, マルチプロセス環境では各プロセスにプロセス数で行ブロック分割した部分ベクトルを作成する.

ベクトル v を作成するプログラムは以下のように記述する. ただし, マルチプロセス環境のプロセス数は 2 とする.

C (逐次・マルチスレッド環境)

```
 1: LIS_INT       i,n;
 2: LIS_VECTOR    v;
 3: n = 4;
 4: lis_vector_create(0,&v);
 5: lis_vector_set_size(v,0,n);              /* or lis_vector_set_size(v,n,0); */
 6:
 7: for(i=0;i<n;i++)
 8: {
 9:     lis_vector_set_value(LIS_INS_VALUE,i,(double)i,v);
10: }
```

C (マルチプロセス環境)

```
 1: LIS_INT       i,n,is,ie;                /* or LIS_INT i,ln,is,ie; */
 2: LIS_VECTOR    v;
 3: n = 4;                                  /* ln = 2; */
 4: lis_vector_create(MPI_COMM_WORLD,&v);
 5: lis_vector_set_size(v,0,n);             /* lis_vector_set_size(v,ln,0); */
 6: lis_vector_get_range(v,&is,&ie);
 7: for(i=is;i<ie;i++)
 8: {
 9:     lis_vector_set_value(LIS_INS_VALUE,i,(double)i,v);
10: }
```

Fortran (逐次・マルチスレッド環境)

```
 1: LIS_INTEGER   i,n
 2: LIS_VECTOR    v
 3: n = 4
 4: call lis_vector_create(0,v,ierr)
 5: call lis_vector_set_size(v,0,n,ierr)
 6:
 7: do i=1,n
 9:     call lis_vector_set_value(LIS_INS_VALUE,i,DBLE(i),v,ierr)
10: enddo
```

Fortran (マルチプロセス環境)

```
 1: LIS_INTEGER   i,n,is,ie
 2: LIS_VECTOR    v
 3: n = 4
 4: call lis_vector_create(MPI_COMM_WORLD,v,ierr)
 5: call lis_vector_set_size(v,0,n,ierr)
 6: call lis_vector_get_range(v,is,ie,ierr)
 7: do i=is,ie-1
 8:     call lis_vector_set_value(LIS_INS_VALUE,i,DBLE(i),v,ierr);
 9: enddo
```

変数宣言

第 2 行のように

```
LIS_VECTOR    v;
```

と宣言する.

ベクトルの作成

ベクトル v の作成には, 関数

- C　　`LIS_INT lis_vector_create(LIS_Comm comm, LIS_VECTOR *v)`
- Fortran　`subroutine lis_vector_create(LIS_Comm comm, LIS_VECTOR v, LIS_INTEGER ierr)`

を用いる. comm には MPI コミュニケータを指定する. 逐次, マルチスレッド環境では comm の値は無視される.

次数の設定

次数の設定には, 関数

- C　　LIS_INTEGER lis_vector_set_size(LIS_VECTOR v, LIS_INT local_n, LIS_INT global_n)
- Fortran　subroutine lis_vector_set_size(LIS_VECTIR v, LIS_INTEGER local_n, LIS_INTEGER global_n, LIS_INTEGER ierr)

を用いる. $local\_n$ か $global\_n$ のどちらか一方を与えなければならない.

逐次, マルチスレッド環境では, $local\_n$ は $global\_n$ に等しい. したがって, lis_vector_set_size(v,n,0) と lis_vector_set_size(v,0,n) は, いずれも次数 n のベクトルを作成する.

マルチプロセス環境においては, lis_vector_set_size(v,n,0) は各プロセス上に次数 n の部分ベクトルを作成する. 一方, lis_vector_set_size(v,0,n) は各プロセス p 上に次数 m_p の部分ベクトルを作成する. m_p はライブラリ側で決定される.

値の代入

ベクトル v の第 i 行に値を代入するには, 関数

- C　　LIS_INT lis_vector_set_value(LIS_INT flag, LIS_INT i, LIS_SCALAR value, LIS_VECTOR v)
- Fortran　subroutine lis_vector_set_value(LIS_INTEGER flag, LIS_INTEGER i, LIS_SCALAR value, LIS_VECTOR v, LIS_INTEGER ierr)

を用いる. マルチプロセス環境では, 部分ベクトルの第 i 行ではなく, 全体ベクトルの第 i 行を指定する. flag には

LIS_INS_VALUE 挿入: $v[i] = value$

LIS_ADD_VALUE 加算代入: $v[i] = v[i] + value$

のどちらかを指定する.

ベクトルの複製

既存のベクトルと同じ情報を持つベクトルを作成するには, 関数

- C　　LIS_INT lis_vector_duplicate(LIS_VECTOR vin, LIS_VECTOR *vout)
- Fortran　subroutine lis_vector_duplicate(LIS_VECTOR vin, LIS_VECTOR vout, LIS_INTEGER ierr)

を用いる. 第 1 引数 LIS_VECTOR vin は LIS_MATRIX を指定することも可能である. この関数はベクトルの要素の値は複製しない. 値も複製する場合は, この関数の後に

- C　　LIS_INT lis_vector_copy(LIS_VECTOR vsrc, LIS_VECTOR vdst)
- Fortran　subroutine lis_vector_copy(LIS_VECTOR vsrc, LIS_VECTOR vdst, LIS_INTEGER ierr)

を呼び出す.

ベクトルの破棄

不要になったベクトルをメモリから破棄するには,

- C　　LIS_INT lis_vector_destroy(LIS_VECTOR v)
- Fortran subroutine lis_vector_destroy(LIS_VECTOR v, LIS_INTEGER ierr)

を用いる.

3.3 行列の操作

係数行列 A の次数を $global\_n \times global\_n$ とする. 行列 A を $nprocs$ 個のプロセスで行ブロック分割する場合の各ブロックの行数を $local\_n$ とする. $global\_n$ が $nprocs$ で割り切れる場合は $local\_n = global\_n$ / $nprocs$ となる. 例えば, 行列 A を (3.2) 式のように 2 個のプロセスで行ブロック分割する場合, $global\_n$ と $local\_n$ はそれぞれ 4 と 2 となる.

$$A = \left(\begin{array}{cccc} 2 & 1 & & \\ 1 & 2 & 1 & \\ \hline & 1 & 2 & 1 \\ & & 1 & 2 \end{array}\right) \begin{array}{l} \text{PE0} \\ \\ \text{PE1} \\ \\ \end{array} \tag{3.2}$$

目的の格納形式の行列を作成するには以下の 3 つの方法がある.

方法 1: ライブラリ関数を用いて目的の格納形式の配列を定義する場合

(3.2) 式の行列 A を CSR 形式で作成する場合, 逐次, マルチスレッド環境では行列 A そのものを, マルチプロセス環境では各プロセスにプロセス数で行ブロック分割した部分行列を作成する.

行列 A を CSR 形式で作成するプログラムは以下のように記述する. ただし, マルチプロセス環境のプロセス数は 2 とする.

C (逐次・マルチスレッド環境)

```
 1: LIS_INT       i,n;
 2: LIS_MATRIX    A;
 3: n = 4;
 4: lis_matrix_create(0,&A);
 5: lis_matrix_set_size(A,0,n);              /* or lis_matrix_set_size(A,n,0); */
 6: for(i=0;i<n;i++) {
 7:     if( i>0   ) lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0,A);
 8:     if( i<n-1 ) lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0,A);
 9:     lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0,A);
10: }
11: lis_matrix_set_type(A,LIS_MATRIX_CSR);
12: lis_matrix_assemble(A);
```

C (マルチプロセス環境)

```
 1: LIS_INT       i,n,gn,is,ie;
 2: LIS_MATRIX    A;
 3: gn = 4;                                  /* or n=2 */
 4: lis_matrix_create(MPI_COMM_WORLD,&A);
 5: lis_matrix_set_size(A,0,gn);             /* lis_matrix_set_size(A,n,0); */
 6: lis_matrix_get_size(A,&n,&gn);
 7: lis_matrix_get_range(A,&is,&ie);
 8: for(i=is;i<ie;i++) {
 9:     if( i>0    ) lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0,A);
10:     if( i<gn-1 ) lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0,A);
11:     lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0,A);
12: }
13: lis_matrix_set_type(A,LIS_MATRIX_CSR);
14: lis_matrix_assemble(A);
```

Fortran (逐次・マルチスレッド環境)

```
 1: LIS_INTEGER   i,n
 2: LIS_MATRIX    A
 3: n = 4
 4: call lis_matrix_create(0,A,ierr)
 5: call lis_matrix_set_size(A,0,n,ierr)
 6: do i=1,n
 7:     if( i>1 ) call lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0d0,A,ierr)
 8:     if( i<n ) call lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0d0,A,ierr)
 9:     call lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0d0,A,ierr)
10: enddo
11: call lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
12: call lis_matrix_assemble(A,ierr)
```

Fortran (マルチプロセス環境)

```
 1: LIS_INTEGER   i,n,gn,is,ie
 2: LIS_MATRIX    A
 3: gn = 4
 4: call lis_matrix_create(MPI_COMM_WORLD,A,ierr)
 5: call lis_matrix_set_size(A,0,gn,ierr)
 6: call lis_matrix_get_size(A,n,gn,ierr)
 7: call lis_matrix_get_range(A,is,ie,ierr)
 8: do i=is,ie-1
 9:     if( i>1  ) call lis_matrix_set_value(LIS_INS_VALUE,i,i-1,1.0d0,A,ierr)
10:     if( i<gn ) call lis_matrix_set_value(LIS_INS_VALUE,i,i+1,1.0d0,A,ierr)
11:     call lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0d0,A,ierr)
12: enddo
13: call lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
14: call lis_matrix_assemble(A,ierr)
```

変数宣言

第 2 行のように

```
LIS_MATRIX    A;
```

と宣言する.

行列の作成

行列 A の作成には, 関数

- C　　LIS_INT lis_matrix_create(LIS_Comm comm, LIS_MATRIX *A)
- Fortran subroutine lis_matrix_create(LIS_Comm comm, LIS_MATRIX A, LIS_INTEGER ierr)

を用いる. comm には MPI コミュニケータを指定する. 逐次, マルチスレッド環境では, comm の値は無視される.

次数の設定

次数の設定には, 関数

- C　　LIS_INT lis_matrix_set_size(LIS_MATRIX A, LIS_INT local_n, LIS_INT global_n)
- Fortran subroutine lis_matrix_set_size(LIS_MATRIX A, LIS_INTEGER local_n, LIS_INTEGER global_n, LIS_INTEGER ierr)

を用いる. $local\_n$ か $global\_n$ のどちらか一方を与えなければならない.

逐次, マルチスレッド環境では, $local\_n$ は $global\_n$ に等しい. したがって, lis_matrix_set_size(A,n,0) と lis_matrix_set_size(A,0,n) は, いずれも次数 $n \times n$ の行列を作成する.

マルチプロセス環境においては, lis_matrix_set_size(A,n,0) は各プロセス上に次数 $n \times N$ の部分行列を作成する. N は n の総和である.
一方, lis_matrix_set_size(A,0,n) は各プロセス p 上に次数 $m_p \times n$ の部分行列を作成する. m_p はライブラリ側で決定される.

値の代入

行列 A の第 i 行第 j 列に値を代入するには, 関数

- C　　LIS_INT lis_matrix_set_value(LIS_INT flag, LIS_INT i, LIS_INT j, LIS_SCALAR value, LIS_MATRIX A)
- Fortran subroutine lis_matrix_set_value(LIS_INTEGER flag, LIS_INTEGER i, LIS_INTEGER j, LIS_SCALAR value, LIS_MATRIX A, LIS_INTEGER ierr)

を用いる. マルチプロセス環境では, 全体行列の第 i 行第 j 列を指定する. flag には

LIS_INS_VALUE 挿入: $A[i,j] = value$

LIS_ADD_VALUE 加算代入: $A[i,j] = A[i,j] + value$

のどちらかを指定する.

行列格納形式の設定

行列の格納形式を設定するには, 関数

- C　　LIS_INT lis_matrix_set_type(LIS_MATRIX A, LIS_INT matrix_type)
- Fortran subroutine lis_matrix_set_type(LIS_MATRIX A, LIS_INTEGER matrix_type, LIS_INTEGER ierr)

を用いる. 行列作成時の A の matrix_type は LIS_MATRIX_CSR である. 以下に対応する格納形式を示す.

格納形式		matrix_type
Compressed Sparse Row	(CSR)	{LIS_MATRIX_CSR|1}
Compressed Sparse Column	(CSC)	{LIS_MATRIX_CSC|2}
Modified Compressed Sparse Row	(MSR)	{LIS_MATRIX_MSR|3}
Diagonal	(DIA)	{LIS_MATRIX_DIA|4}
Ellpack-Itpack Generalized Diagonal	(ELL)	{LIS_MATRIX_ELL|5}
Jagged Diagonal	(JAD)	{LIS_MATRIX_JAD|6}
Block Sparse Row	(BSR)	{LIS_MATRIX_BSR|7}
Block Sparse Column	(BSC)	{LIS_MATRIX_BSC|8}
Variable Block Row	(VBR)	{LIS_MATRIX_VBR|9}
Coordinate	(COO)	{LIS_MATRIX_COO|10}
Dense	(DNS)	{LIS_MATRIX_DNS|11}

行列の組み立て

行列の要素と格納形式を設定した後, 関数

- C　　LIS_INT lis_matrix_assemble(LIS_MATRIX A)
- Fortran subroutine lis_matrix_assemble(LIS_MATRIX A, LIS_INTEGER ierr)

を呼び出す. lis_matrix_assemble は lis_matrix_set_type で指定された格納形式に組み立てられる.

行列の破棄

不要になった行列をメモリから破棄するには,

- C　　LIS_INT lis_matrix_destroy(LIS_MATRIX A)
- Fortran subroutine lis_matrix_destroy(LIS_MATRIX A, LIS_INTEGER ierr)

を用いる.

方法 2: 目的の格納形式の配列を直接定義する場合

(3.2) 式の行列 A を CSR 形式で作成する場合, 逐次, マルチスレッド環境では行列 A そのものを, マルチプロセス環境では各プロセスにプロセス数で行ブロック分割した部分行列を作成する.

行列 A を CSR 形式で作成するプログラムは以下のように記述する. ただし, マルチプロセス環境のプロセス数は 2 とする.

C (逐次・マルチスレッド環境)

```
 1: LIS_INT       i,k,n,nnz;
 2: LIS_INT       *ptr,*index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; nnz = 10; k = 0;
 6: lis_matrix_malloc_csr(n,nnz,&ptr,&index,&value);
 7: lis_matrix_create(0,&A);
 8: lis_matrix_set_size(A,0,n);              /* or lis_matrix_set_size(A,n,0); */
 9:
10: for(i=0;i<n;i++)
11: {
12:     if( i>0   ) {index[k] = i-1; value[k] = 1; k++;}
13:     index[k] = i; value[k] = 2; k++;
14:     if( i<n-1 ) {index[k] = i+1; value[k] = 1; k++;}
15:     ptr[i+1] = k;
16: }
17: ptr[0] = 0;
18: lis_matrix_set_csr(nnz,ptr,index,value,A);
19: lis_matrix_assemble(A);
```

C (マルチプロセス環境)

```
 1: LIS_INT       i,k,n,nnz,is,ie;
 2: LIS_INT       *ptr,*index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 2; nnz = 5; k = 0;
 6: lis_matrix_malloc_csr(n,nnz,&ptr,&index,&value);
 7: lis_matrix_create(MPI_COMM_WORLD,&A);
 8: lis_matrix_set_size(A,n,0);
 9: lis_matrix_get_range(A,&is,&ie);
10: for(i=is;i<ie;i++)
11: {
12:     if( i>0   ) {index[k] = i-1; value[k] = 1; k++;}
13:     index[k] = i; value[k] = 2; k++;
14:     if( i<n-1 ) {index[k] = i+1; value[k] = 1; k++;}
15:     ptr[i-is+1] = k;
16: }
17: ptr[0] = 0;
18: lis_matrix_set_csr(nnz,ptr,index,value,A);
19: lis_matrix_assemble(A);
```

配列の関連付け

CSR 形式の配列をライブラリが扱えるよう行列 A に関連付けるには, 関数

- C　　LIS_INT lis_matrix_set_csr(LIS_INT nnz, LIS_INT row[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)
- Fortran subroutine lis_matrix_set_csr(LIS_INTEGER nnz, LIS_INTEGER row(), LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

を用いる. その他の格納形式については 5 節を参照せよ.

方法 3: 外部ファイルから行列, ベクトルデータを読み込む場合

外部ファイルから (3.2) 式の行列 A を CSR 形式で読み込む場合, プログラムは以下のように記述する.

C (逐次・マルチスレッド・マルチプロセス環境)

```
1: LIS_MATRIX    A;
3: lis_matrix_create(LIS_COMM_WORLD,&A);
6: lis_matrix_set_type(A,LIS_MATRIX_CSR);
7: lis_input_matrix(A,"matvec.mtx");
```

Fortran (逐次・マルチスレッド・マルチプロセス環境)

```
1: LIS_MATRIX    A
3: call lis_matrix_create(LIS_COMM_WORLD,A,ierr)
6: call lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
7: call lis_input_matrix(A,'matvec.mtx',ierr)
```

Matrix Market 形式による外部ファイル matvec.mtx の記述例を以下に示す.

```
%%MatrixMarket matrix coordinate real general
4 4 10 1 0
1 2  1.0e+00
1 1  2.0e+00
2 3  1.0e+00
2 1  1.0e+00
2 2  2.0e+00
3 4  1.0e+00
3 2  1.0e+00
3 3  2.0e+00
4 4  2.0e+00
4 3  1.0e+00
```

外部ファイルから (3.2) 式の行列 A を CSR 形式で, また (3.1) 式のベクトル b を読み込む場合のプログラムは以下のように記述する.

C (逐次・マルチスレッド・マルチプロセス環境)

```
1: LIS_MATRIX    A;
2: LIS_VECTOR    b,x;
3: lis_matrix_create(LIS_COMM_WORLD,&A);
4: lis_vector_create(LIS_COMM_WORLD,&b);
5: lis_vector_create(LIS_COMM_WORLD,&x);
6: lis_matrix_set_type(A,LIS_MATRIX_CSR);
7: lis_input(A,b,x,"matvec.mtx");
```

Fortran (逐次・マルチスレッド・マルチプロセス環境)

```
1: LIS_MATRIX    A
2: LIS_VECTOR    b,x
3: call lis_matrix_create(LIS_COMM_WORLD,A,ierr)
4: call lis_vector_create(LIS_COMM_WORLD,b,ierr)
5: call lis_vector_create(LIS_COMM_WORLD,x,ierr)
6: call lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
7: call lis_input(A,b,x,'matvec.mtx',ierr)
```

拡張 Matrix Market 形式による外部ファイル matvec.mtx の記述例を以下に示す (付録 A を参照).

```
%%MatrixMarket matrix coordinate real general
4 4 10 1 0
1 2  1.0e+00
1 1  2.0e+00
2 3  1.0e+00
2 1  1.0e+00
2 2  2.0e+00
3 4  1.0e+00
3 2  1.0e+00
3 3  2.0e+00
4 4  2.0e+00
4 3  1.0e+00
1  0.0e+00
2  1.0e+00
3  2.0e+00
4  3.0e+00
```

外部ファイルからの読み込み

外部ファイルから行列 A のデータを読み込むには, 関数

- `C       LIS_INT lis_input_matrix(LIS_MATRIX A, char *filename)`
- `Fortran subroutine lis_input_matrix(LIS_MATRIX A,`
  `character filename, LIS_INTEGER ierr)`

を用いる. `filename` にはファイルパスを指定する. 対応するファイル形式は以下の通りである (ファイル形式については付録 A を参照).

- Matrix Market 形式
- Harwell-Boeing 形式

外部ファイルから行列 A とベクトル b, x のデータを読み込むには, 関数

- `C       LIS_INT lis_input(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x, char *filename)`
- `Fortran subroutine lis_input(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,`
  `character filename, LIS_INTEGER ierr)`

を用いる. `filename` にはファイルパスを指定する. 対応するファイル形式は以下の通りである (ファイル形式については付録 A を参照).

- 拡張 Matrix Market 形式
- Harwell-Boeing 形式

3.4 線型方程式の求解

線型方程式 $Ax = b$ を指定された解法で解く場合, プログラムは以下のように記述する.

C (逐次・マルチスレッド・マルチプロセス環境)

```
 1: LIS_MATRIX A;
 2: LIS_VECTOR b,x;
 3: LIS_SOLVER solver;
 4:
 5: /* 行列とベクトルの作成 */
 6:
 7: lis_solver_create(&solver);
 8: lis_solver_set_option("-i bicg -p none",solver);
 9: lis_solver_set_option("-tol 1.0e-12",solver);
10: lis_solve(A,b,x,solver);
```

Fortran (逐次・マルチスレッド・マルチプロセス環境)

```
 1: LIS_MATRIX A
 2: LIS_VECTOR b,x
 3: LIS_SOLVER solver
 4:
 5: /* 行列とベクトルの作成 */
 6:
 7: call lis_solver_create(solver,ierr)
 8: call lis_solver_set_option('-i bicg -p none',solver,ierr)
 9: call lis_solver_set_option('-tol 1.0e-12',solver,ierr)
10: call lis_solve(A,b,x,solver,ierr)
```

ソルバの作成

ソルバ (線型方程式解法の情報を格納する構造体) を作成するには, 関数

- C　　LIS_INT lis_solver_create(LIS_SOLVER *solver)
- Fortran subroutine lis_solver_create(LIS_SOLVER solver, LIS_INTEGER ierr)

を用いる.

オプションの設定

線型方程式解法をソルバに設定するには, 関数

- C　　LIS_INT lis_solver_set_option(char *text, LIS_SOLVER solver)
- Fortran subroutine lis_solver_set_option(character text, LIS_SOLVER solver, LIS_INTEGER ierr)

または

- C　　LIS_INT lis_solver_set_optionC(LIS_SOLVER solver)
- Fortran subroutine lis_solver_set_optionC(LIS_SOLVER solver, LIS_INTEGER ierr)

を用いる. lis_solver_set_optionC は, ユーザプログラム実行時にコマンドラインで指定されたオプションをソルバに設定する関数である.

以下に指定可能なコマンドラインオプションを示す. -i {cg|1}は-i cg または-i 1 を意味する. -maxiter [1000] は, -maxiter の既定値が 1000 であることを意味する.

線型方程式解法に関するオプション (既定値: `-i bicg`)

線型方程式解法	オプション	補助オプション	
CG	`-i {cg|1}`		
BiCG	`-i {bicg|2}`		
CGS	`-i {cgs|3}`		
BiCGSTAB	`-i {bicgstab|4}`		
BiCGSTAB(l)	`-i {bicgstabl|5}`	`-ell [2]`	次数 l
GPBiCG	`-i {gpbicg|6}`		
TFQMR	`-i {tfqmr|7}`		
Orthomin(m)	`-i {orthomin|8}`	`-restart [40]`	リスタート値 m
GMRES(m)	`-i {gmres|9}`	`-restart [40]`	リスタート値 m
Jacobi	`-i {jacobi|10}`		
Gauss-Seidel	`-i {gs|11}`		
SOR	`-i {sor|12}`	`-omega [1.9]`	緩和係数 ω $(0 < \omega < 2)$
BiCGSafe	`-i {bicgsafe|13}`		
CR	`-i {cr|14}`		
BiCR	`-i {bicr|15}`		
CRS	`-i {crs|16}`		
BiCRSTAB	`-i {bicrstab|17}`		
GPBiCR	`-i {gpbicr|18}`		
BiCRSafe	`-i {bicrsafe|19}`		
FGMRES(m)	`-i {fgmres|20}`	`-restart [40]`	リスタート値 m
IDR(s)	`-i {idrs|21}`	`-irestart [2]`	リスタート値 s
MINRES	`-i {minres|22}`		

前処理に関するオプション (既定値: `-p none`)

前処理	オプション	補助オプション	
なし	`-p {none|0}`		
Jacobi	`-p {jacobi|1}`		
ILU(k)	`-p {ilu|2}`	`-ilu_fill [0]`	フィルインレベル k
SSOR	`-p {ssor|3}`	`-ssor_w [1.0]`	緩和係数 ω $(0 < \omega < 2)$
Hybrid	`-p {hybrid|4}`	`-hybrid_i [sor]`	線型方程式解法
		`-hybrid_maxiter [25]`	最大反復回数
		`-hybrid_tol [1.0e-3]`	収束判定基準
		`-hybrid_w [1.5]`	SOR の緩和係数 ω $(0 < \omega < 2)$
		`-hybrid_ell [2]`	BiCGSTAB(l) の次数 l
		`-hybrid_restart [40]`	GMRES(m), Orthomin(m) の リスタート値 m
I+S	`-p {is|5}`	`-is_alpha [1.0]`	$I + \alpha S^{(m)}$ のパラメータ α
		`-is_m [3]`	$I + \alpha S^{(m)}$ のパラメータ m
SAINV	`-p {sainv|6}`	`-sainv_drop [0.05]`	ドロップ基準
SA-AMG	`-p {saamg|7}`	`-saamg_unsym [false]`	非対称版の選択 (行列構造は対称とする)
		`-saamg_theta [0.05|0.12]`	ドロップ基準 $a_{ij}^2 \leq \theta^2 \lvert a_{ii}\rvert \lvert a_{jj}\rvert$ (対称|非対称)
Crout ILU	`-p {iluc|8}`	`-iluc_drop [0.05]`	ドロップ基準
		`-iluc_rate [5.0]`	最大フィルイン数の倍率
ILUT	`-p {ilut|9}`	`-ilut_drop [0.05]`	ドロップ基準
		`-ilut_rate [5.0]`	最大フィルイン数の倍率
Additive Schwarz	`-adds true`	`-adds_iter [1]`	繰り返し回数

その他のオプション

オプション	
-maxiter [1000]	最大反復回数
-tol [1.0e-12]	収束判定基準 tol
-tol_w [1.0]	収束判定基準 tol_w
-print [0]	残差履歴の出力
	-print {none|0} 残差履歴を出力しない
	-print {mem|1} 残差履歴をメモリに保存する
	-print {out|2} 残差履歴を標準出力に書き出す
	-print {all|3} 残差履歴をメモリに保存し, 標準出力に書き出す
-scale [0]	スケーリングの選択. 結果は元の行列, ベクトルに上書きされる
	-scale {none|0} スケーリングなし
	-scale {jacobi|1} Jacobi スケーリング $D^{-1}Ax = D^{-1}b$
	(D は $A = (a_{ij})$ の対角部分)
	-scale {symm_diag|2} 対角スケーリング $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$
	($D^{-1/2}$ は対角要素の値が $1/\sqrt{a_{ii}}$ である対角行列)
-initx_zeros [1]	初期ベクトル x_0
	-initx_zeros {false|0} 与えられた値を使用
	-initx_zeros {true|1} すべての要素の値を 0 にする
-conv_cond [0]	収束条件
	-conv_cond {nrm2_r|0} $\|\|b - Ax\|\|_2 \leq tol * \|\|b\|\|_2$
	-conv_cond {nrm2_b|1} $\|\|b - Ax\|\|_2 \leq tol * \|\|b - Ax_0\|\|_2$
	-conv_cond {nrm1_b|2} $\|\|b - Ax\|\|_1 \leq tol_w * \|\|b\|\|_1 + tol$
-omp_num_threads [t]	実行スレッド数
	(t は最大スレッド数)
-storage [0]	行列格納形式
-storage_block [2]	BSR, BSC 形式のブロックサイズ
-f [0]	線型方程式解法の精度
	-f {double|0} 倍精度
	-f {quad|1} 4 倍精度

求解

線型方程式 $Ax = b$ を解くには, 関数

- C `LIS_INT lis_solve(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x, LIS_SOLVER solver)`
- Fortran `subroutine lis_solve(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x, LIS_SOLVER solver, LIS_INTEGER ierr)`

を用いる.

3.5 固有値問題の求解

固有値問題 $Ax = \lambda x$ を指定の解法で解く場合, プログラムは以下のように記述する.

C (逐次・マルチスレッド・マルチプロセス環境)

```
 1: LIS_MATRIX A;
 2: LIS_VECTOR x;
 3: LIS_REAL evalue;
 4: LIS_ESOLVER esolver;
 5:
 6: /* 行列とベクトルの作成 */
 7:
 8: lis_esolver_create(&esolver);
 9: lis_esolver_set_option("-e ii -i bicg -p none",esolver);
10: lis_esolver_set_option("-etol 1.0e-12 -tol 1.0e-12",esolver);
11: lis_esolve(A,x,evalue,esolver);
```

Fortran (逐次・マルチスレッド・マルチプロセス環境)

```
 1: LIS_MATRIX A
 2: LIS_VECTOR x
 3: LIS_REAL evalue
 4: LIS_ESOLVER esolver
 5:
 6: /* 行列とベクトルの作成 */
 7:
 8: call lis_esolver_create(esolver,ierr)
 9: call lis_esolver_set_option('-e ii -i bicg -p none',esolver,ierr)
10: call lis_esolver_set_option('-etol 1.0e-12 -tol 1.0e-12',esolver,ierr)
11: call lis_esolve(A,x,evalue,esolver,ierr)
```

ソルバの作成

ソルバ (固有値解法の情報を格納する構造体) を作成するには, 関数

- C　　LIS_INT lis_esolver_create(LIS_ESOLVER *esolver)
- Fortran subroutine lis_esolver_create(LIS_ESOLVER esolver, LIS_INTEGER ierr)

を用いる.

オプションの設定

固有値解法をソルバに設定するには, 関数

- C　　LIS_INT lis_esolver_set_option(char *text, LIS_ESOLVER esolver)
- Fortran subroutine lis_esolver_set_option(character text, LIS_ESOLVER esolver, LIS_INTEGER ierr)

または

- C　　LIS_INT lis_esolver_set_optionC(LIS_ESOLVER esolver)
- Fortran subroutine lis_esolver_set_optionC(LIS_ESOLVER esolver, LIS_INTEGER ierr)

を用いる. lis_esolver_set_optionC は, ユーザプログラム実行時にコマンドラインで指定されたオプションをソルバに設定する関数である.

以下に指定可能なコマンドラインオプションを示す. -e {pi|1}は-e pi または-e 1 を意味する. -emaxiter [1000] は, -emaxiter の既定値が 1000 であることを意味する.

固有値解法に関するオプション (既定値: -e pi)

固有値解法	オプション	補助オプション	
Power	-e {pi|1}		
Inverse	-e {ii|2}	-i [cg]	線型方程式解法
Approximate Inverse	-e {aii|3}	-i [cg]	線型方程式解法
Rayleigh Quotient	-e {rqi|4}	-i [cg]	線型方程式解法
Subspace	-e {si|5}	-ss [2]	部分空間の大きさ
		-m [0]	モード番号
Lanczos	-e {li|6}	-ss [2]	部分空間の大きさ
		-m [0]	モード番号
CG	-e {cg|7}		
CR	-e {cr|8}		

前処理に関するオプション (既定値: -p ilu)

前処理	オプション	補助オプション	
なし	-p {none|0}		
Jacobi	-p {jacobi|1}		
ILU(k)	-p {ilu|2}	-ilu_fill [0]	フィルインレベル k
SSOR	-p {ssor|3}	-ssor_w [1.0]	緩和係数 ω $(0 < \omega < 2)$
Hybrid	-p {hybrid|4}	-hybrid_i [sor]	線型方程式解法
		-hybrid_maxiter [25]	最大反復回数
		-hybrid_tol [1.0e-3]	収束判定基準
		-hybrid_w [1.5]	SOR の緩和係数 ω $(0 < \omega < 2)$
		-hybrid_ell [2]	BiCGSTAB(l) の次数 l
		-hybrid_restart [40]	GMRES(m), Orthomin(m) のリスタート値 m
I+S	-p {is|5}	-is_alpha [1.0]	$I + \alpha S^{(m)}$ のパラメータ α
		-is_m [3]	$I + \alpha S^{(m)}$ のパラメータ m
SAINV	-p {sainv|6}	-sainv_drop [0.05]	ドロップ基準
SA-AMG	-p {saamg|7}	-saamg_unsym [false]	非対称版の選択 (行列構造は対称とする)
		-saamg_theta [0.05|0.12]	ドロップ基準 $a_{ij}^2 \leq \theta^2 \lvert a_{ii}\rvert\lvert a_{jj}\rvert$ (対称|非対称)
crout ILU	-p {iluc|8}	-iluc_drop [0.05]	ドロップ基準
		-iluc_rate [5.0]	最大フィルイン数の倍率
ILUT	-p {ilut|9}	-ilut_drop [0.05]	ドロップ基準
		-ilut_rate [5.0]	最大フィルイン数の倍率
Additive Schwarz	-adds true	-adds_iter [1]	繰り返し回数

その他のオプション

オプション	
-emaxiter [1000]	最大反復回数
-etol [1.0e-12]	収束判定基準
-eprint [0]	残差履歴の出力
	-eprint {none|0} 残差履歴を出力しない
	-eprint {mem|1} 残差履歴をメモリに保存する
	-eprint {out|2} 残差履歴を標準出力に書き出す
	-eprint {all|3} 残差履歴をメモリに保存し, 標準出力に書き出す
-ie [ii]	Subspace, Lanczos 法の内部で使用する固有値解法の指定
	-ie {pi|1} Power (Subspace のみ)
	-ie {ii|2} Inverse
	-ie {aii|3} Approximate Inverse
	-ie {rqi|4} Rayleigh Quotient
-shift [0.0]	固有値のシフト量
-initx_ones [1]	初期ベクトル x_0
	-initx_ones {false|0} 与えられた値を使用
	-initx_ones {true|1} すべての要素の値を 1 にする
-omp_num_threads [t]	実行スレッド数
	t は最大スレッド数
-estorage [0]	行列格納形式
-estorage_block [2]	BSR, BSC 形式のブロックサイズ
-ef [0]	固有値解法の精度
	-ef {double|0} 倍精度
	-ef {quad|1} 4 倍精度

求解

固有値問題 $Ax = \lambda x$ を解くには, 関数

- C
```
LIS_INT lis_esolve(LIS_MATRIX A, LIS_VECTOR x,
 LIS_REAL evalue, LIS_ESOLVER esolver)
```
- Fortran
```
subroutine lis_esolve(LIS_MATRIX A, LIS_VECTOR x,
 LIS_ESOLVER esolver, LIS_INTEGER ierr)
```

を用いる.

3.6 プログラムの作成

線型方程式 $Ax = b$ を指定された解法で解き, その解を標準出力に書き出すプログラムを以下に示す.

行列 A は次数 12 の 3 重対角行列

$$A = \begin{pmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & -1 & 2 & -1 & \\ & & & -1 & 2 & \end{pmatrix}$$

である. 右辺ベクトル b は解 x がすべて 1 となるよう設定される.

このプログラムはディレクトリ `lis-($VERSION)/test` にある.

検証プログラム: test4.c

```
 1: #include <stdio.h>
 2: #include "lis.h"
 3: main(LIS_INT argc, char *argv[])
 4: {
 5:     LIS_INT i,n,gn,is,ie,iter;
 6:     LIS_MATRIX A;
 7:     LIS_VECTOR b,x,u;
 8:     LIS_SOLVER solver;
 9:     n = 12;
10:     lis_initialize(&argc,&argv);
11:     lis_matrix_create(LIS_COMM_WORLD,&A);
12:     lis_matrix_set_size(A,0,n);
13:     lis_matrix_get_size(A,&n,&gn)
14:     lis_matrix_get_range(A,&is,&ie)
15:     for(i=is;i<ie;i++)
16:     {
17:         if( i>0 ) lis_matrix_set_value(LIS_INS_VALUE,i,i-1,-1.0,A);
18:         if( i<gn-1 ) lis_matrix_set_value(LIS_INS_VALUE,i,i+1,-1.0,A);
19:         lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0,A);
20:     }
21:     lis_matrix_set_type(A,LIS_MATRIX_CSR);
22:     lis_matrix_assemble(A);
23:
24:     lis_vector_duplicate(A,&u);
25:     lis_vector_duplicate(A,&b);
26:     lis_vector_duplicate(A,&x);
27:     lis_vector_set_all(1.0,u);
28:     lis_matvec(A,u,b);
29:
30:     lis_solver_create(&solver);
31:     lis_solver_set_optionC(solver);
32:     lis_solve(A,b,x,solver);
33:     lis_solver_get_iter(solver,&iter);
34:     printf("number of iterations = %d\n",iter);
35:     lis_vector_print(x);
36:     lis_matrix_destroy(A);
37:     lis_vector_destroy(u);
38:     lis_vector_destroy(b);
39:     lis_vector_destroy(x);
40:     lis_solver_destroy(solver);
41:     lis_finalize();
42:     return 0;
43: }
}
```

検証プログラム: test4f.F

```
 1:      implicit none
 2:
 3:#include "lisf.h"
 4:
 5:      LIS_INTEGER       i,n,gn,is,ie,iter,ierr
 6:      LIS_MATRIX        A
 7:      LIS_VECTOR        b,x,u
 8:      LIS_SOLVER        solver
 9:      n  = 12
10:      call lis_initialize(ierr)
11:      call lis_matrix_create(LIS_COMM_WORLD,A,ierr)
12:      call lis_matrix_set_size(A,0,n,ierr)
13:      call lis_matrix_get_size(A,n,gn,ierr)
14:      call lis_matrix_get_range(A,is,ie,ierr)
15:      do i=is,ie-1
16:        if( i>1  ) call lis_matrix_set_value(LIS_INS_VALUE,i,i-1,-1.0d0,
17:     .                                       A,ierr)
18:        if( i<gn ) call lis_matrix_set_value(LIS_INS_VALUE,i,i+1,-1.0d0,
19:     .                                       A,ierr)
20:        call lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0d0,A,ierr)
21:      enddo
22:      call lis_matrix_set_type(A,LIS_MATRIX_CSR,ierr)
23:      call lis_matrix_assemble(A,ierr)
24:
25:      call lis_vector_duplicate(A,u,ierr)
26:      call lis_vector_duplicate(A,b,ierr)
27:      call lis_vector_duplicate(A,x,ierr)
28:      call lis_vector_set_all(1.0d0,u,ierr)
29:      call lis_matvec(A,u,b,ierr)
30:
31:      call lis_solver_create(solver,ierr)
32:      call lis_solver_set_optionC(solver,ierr)
33:      call lis_solve(A,b,x,solver,ierr)
34:      call lis_solver_get_iter(solver,iter,ierr)
35:      write(*,*) 'number of iterations = ',iter
36:      call lis_vector_print(x,ierr)
37:      call lis_matrix_destroy(A,ierr)
38:      call lis_vector_destroy(b,ierr)
39:      call lis_vector_destroy(x,ierr)
40:      call lis_vector_destroy(u,ierr)
41:      call lis_solver_destroy(solver,ierr)
42:      call lis_finalize(ierr)
43:
44:      stop
45:      end
```

3.7 実行ファイルの生成

test4.c から実行ファイルを生成する方法について述べる．ディレクトリ lis-($VERSION)/test にある検証プログラム test4.c を SGI Altix 3700 上の Intel C/C++ Compiler (icc), Intel Fortran Compiler (ifort) でコンパイルする場合の例を以下に示す．SA-AMG 前処理には Fortran90 で記述されたコードが含まれるため, SA-AMG 前処理を使用する場合には Fortran90 コンパイラでリンクしなければならない．また, マルチプロセス環境ではプリプロセッサマクロ USE_MPI が定義されなければならない．64bit 整数型を

使用する場合, C プログラムについてはプリプロセッサマクロ_LONGLONG が, また Fortran プログラムについてはプリプロセッサマクロ LONGLONG が定義されなければならない.

逐次環境

コンパイル
```
> icc -c -I($INSTALLDIR)/include test4.c
```
リンク
```
> icc -o test4 test4.o -llis
```
リンク (--enable-saamg)
```
> ifort -nofor_main -o test4 test4.o -llis
```

マルチスレッド環境

コンパイル
```
> icc -c -openmp -I($INSTALLDIR)/include test4.c
```
リンク
```
> icc -openmp -o test4 test4.o -llis
```
リンク (--enable-saamg)
```
> ifort -nofor_main -openmp -o test4 test4.o -llis
```

マルチプロセス環境

コンパイル
```
> icc -c -DUSE_MPI -I($INSTALLDIR)/include test4.c
```
リンク
```
> icc -o test4 test4.o -llis -lmpi
```
リンク (--enable-saamg)
```
> ifort -nofor_main -o test4 test4.o -llis -lmpi
```

マルチスレッド・マルチプロセス環境

コンパイル
```
> icc -c -openmp -DUSE_MPI -I($INSTALLDIR)/include test4.c
```
リンク
```
> icc -openmp -o test4 test4.o -llis -lmpi
```
リンク (--enable-saamg)
```
> ifort -nofor_main -openmp -o test4 test4.o -llis -lmpi
```

次に, test4f.F から実行ファイルを生成する方法について述べる. ディレクトリ lis-($VERSION)/test にある検証プログラム test4f.F を SGI Altix 3700 上の Intel Fortran Compiler (ifort) でコンパイルする場合の例を以下に示す. Fortran のユーザプログラムにはコンパイラ指示文が含まれるため, プリプロセッサを使用するようコンパイラに指示しなければならない. ifort の場合のオプションは-fpp である.

逐次環境

コンパイル
```
> ifort -c -fpp -I($INSTALLDIR)/include test4f.F
```
リンク
```
> ifort -o test4f test4f.o -llis
```

マルチスレッド環境

コンパイル

```
> ifort -c -fpp -openmp -I($INSTALLDIR)/include test4f.F
```

リンク

```
> ifort -openmp -o test4f test4f.o -llis
```

マルチプロセス環境

コンパイル

```
> ifort -c -fpp -DUSE_MPI -I($INSTALLDIR)/include test4f.F
```

リンク

```
> ifort -o test4f test4f.o -llis -lmpi
```

マルチスレッド・マルチプロセス環境

コンパイル

```
> ifort -c -fpp -openmp -DUSE_MPI -I($INSTALLDIR)/include test4f.F
```

リンク

```
> ifort -openmp -o test4f test4f.o -llis -lmpi
```

3.8 実行

ディレクトリ lis-($VERSION)/test にある検証プログラム test4 または test4f を SGI Altix 3700 上のそれぞれの環境で

逐次環境

```
> ./test4 -i bicgstab
```

マルチスレッド環境

```
> env OMP_NUM_THREADS=2 ./test4 -i bicgstab
```

マルチプロセス環境

```
> mpirun -np 2 ./test4 -i bicgstab
```

マルチスレッド・マルチプロセス環境

```
> mpirun -np 2 env OMP_NUM_THREADS=2 ./test4 -i bicgstab
```

と入力して実行すると, 以下のように解が標準出力に書き出される.

```
initial vector x = 0
precision : double
solver    : BiCGSTAB 4
precon    : none
conv_cond : ||b-Ax||_2 <= 1.0e-12 * ||b-Ax_0||_2
storage   : CSR
lis_solve : normal end

     0  1.000000e-00
     1  1.000000e+00
     2  1.000000e-00
```

```
 3  1.000000e+00
 4  1.000000e-00
 5  1.000000e+00
 6  1.000000e+00
 7  1.000000e-00
 8  1.000000e+00
 9  1.000000e-00
10  1.000000e+00
11  1.000000e-00
```

4　4倍精度演算

反復法の計算では, 丸め誤差の影響によって収束が停滞することがある. 本ライブラリでは, long double 型及び倍精度浮動小数点数を 2 個用いた”double-double”[45, 46] 型の 4 倍精度演算を用いることにより, 収束を改善することが可能である. double-double 型演算では, 浮動小数 a を $a = a.hi + a.lo$, $\frac{1}{2}\mathrm{ulp}(a.hi) \geq |a.lo|$ (上位 $a.hi$ と下位 $a.lo$ は倍精度浮動小数) により定義し, Dekker[47] と Knuth[48] のアルゴリズムに基づいて倍精度の四則演算の組み合わせにより 4 倍精度演算を実現する. double-double 型の演算は一般に Fortran の 4 倍精度演算より高速である [49] が, Fortran の表現形式 [50] では仮数部が 112 ビットであるのに対して, 倍精度浮動小数を 2 個使用するため, 仮数部が 104 ビットとなり, 8 ビット少ない. また, 指数部は倍精度浮動小数と同じ 11 ビットである.

double-double 型演算では, 入力として与えられる行列, ベクトル, 及び出力の解は倍精度である. ユーザプログラムは 4 倍精度変数を直接扱うことはなく, オプションとして 4 倍精度演算を使用するかどうかを指定するだけでよい. なお, Intel 系のアーキテクチャに対しては Streaming SIMD Extensions (SSE) 命令, AVX 命令，AVX2 命令を用いて高速化を行う [51][86].

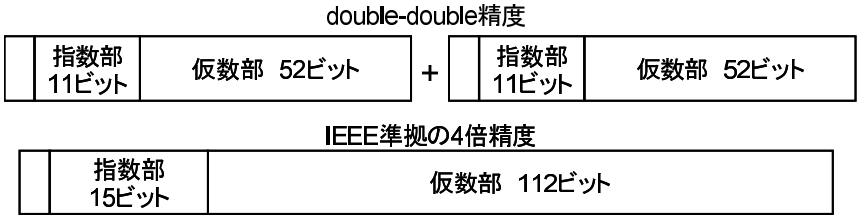

図 2: double-double 型のビット数

4.1　4倍精度演算の使用

Toepliz 行列

$$A = \begin{pmatrix} 2 & 1 & & & & \\ 0 & 2 & 1 & & & \\ \gamma & 0 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & \gamma & 0 & 2 & 1 \\ & & & \gamma & 0 & 2 \end{pmatrix}$$

に対する線型方程式 $Ax = b$ を指定された解法で解き, 解を標準出力に書き出す検証プログラムが `test5.c` である. 右辺ベクトル b は解 x がすべて 1 となるよう設定される. n は行列 A の次数である. `test5` において,

倍精度の場合

```
> ./test5 200 2.0 -f double
```

または

```
> ./test5 200 2.0
```

と入力して実行すると, 以下の結果が得られる.

```
n = 200, gamma = 2.000000
initial vector x = 0
precision : double
solver    : BiCG 2
precon    : none
conv_cond : ||b-Ax||_2 <= 1.0e-12 * ||b-Ax_0||_2
storage   : CSR
lis_solve : LIS_MAXITER(code=4)

BiCG: number of iterations = 1001 (double = 1001, quad = 0)
BiCG: elapsed time         = 2.044368e-02 sec.
BiCG:   preconditioner     = 4.768372e-06 sec.
BiCG:     matrix creation  = 4.768372e-06 sec.
BiCG:   linear solver      = 2.043891e-02 sec.
BiCG: relative residual    = 8.917591e+01
```

4 倍精度の場合

```
> ./test5 200 2.0 -f quad
```

と入力して実行すると, 以下の結果が得られる.

```
n = 200, gamma = 2.000000
initial vector x = 0
precision : quad
solver    : BiCG 2
precon    : none
conv_cond : ||b-Ax||_2 <= 1.0e-12 * ||b-Ax_0||_2
storage   : CSR
lis_solve : normal end

BiCG: number of iterations = 230 (double = 230, quad = 0)
BiCG: elapsed time         = 2.267408e-02 sec.
BiCG:   preconditioner     = 4.549026e-04 sec.
BiCG:     matrix creation  = 5.006790e-06 sec.
BiCG:   linear solver      = 2.221918e-02 sec.
BiCG: relative residual    = 6.499145e-11
```

4.2 DD-AVX の追加機能

4.2.1 BSR 形式を用いた高速な疎行列ベクトル積

AVX，AVX2 を用いた倍精度疎行列と倍々精度ベクトル積は，SIMD 化による端数の処理や非連続なアクセスにより性能が劣化する．これらは，BSR 形式を用いることで解決できる [87]．また，BSR 形式のブロックサイズ $row \times col$ は，4×1 が最適であることがわかっている．

DD-AVX では，AVX,AVX2 有効時，倍精度疎行列と倍々精度ベクトル積を BSR4x1 を実装した．

なお，BSR 形式の生成方法は，“Lis” の倍精度 BSR 形式の生成方法と同じであり，lis_omp_set_blocksize() を用いる．

DD-AVX_test1.c は，test1,c を改良し，4x1 の BCRS 形式を作成できるようにしたものである．

4.2.2 疎行列ベクトル積における OpenMP のスケジューリング方式自動チューニング

倍々精度演算は，演算量が倍精度と比べ多いため，OpenMP を用いたスレッド並列時に負荷分散を行うことの効果が高いことがわかっている [85]．

一般的には OpenMP が用意しているスケジューリング方式 “guided” を用いれば平均的に性能が高いため，DD-AVX では，デフォルトを “guided” に設定した．

しかし，問題行列が極端に負荷分散しにくい構造のとき，事前に行列を分析して，適切な分割方法を用いることが望ましい．

DD-AVX では，事前に行列を分析して負荷分散を行う，

- lis_omp_balanced()

を実装した．

また，不要な行列の分析を回避するために，負荷分散を行うか，“guided” を行うかを自動で判別する関数として，

- lis_omp_auto_select()

を実装した．

以下に，testmatrix.mtx(L.6) を読み込み，Bicg 法で求解する (L.23) コードを示す．このとき，格納形式は BSR4x1 形式 (L.10-14)，OpenMP のスケジューリング方式は自動選択している．(L.20)

4 倍精度向け BCRS 形式・OpenMP スケジューリング方式の自動選択の利用

```
1: lis_initialize(&argc, &argv);
2: LIS_MATRIX A;
3: LIS_INT block;
4: LIS_VECTOR b,x;
5: LIS_SOLVER solver;
7: lis_matrix_create(LIS_COMM_WORLD,&A);
8:
9: /* BSR4x1 の場合*/;
10: lis_matrix_set_type(A,LIS_MATRIX_BSR);
11: A->bnr = 4;  //BSR のブロック行サイズ
12: A->bnc = 1;  //BSR のブロック列サイズ
13: /* BSR4x1 の場合*/;
14:
15: lis_input_matrix(A,"testmatrix.mtx"); /*行列 A に testmatrix.mtx を BSR 形式で読み込む*/
16:
17: lis_solver_create(&solver);
18: /* lis_omp_balanced(solver); OpenMP のスレッド負荷分散を有効にする*/
19: lis_omp_auto_select(solver);/*OpenMP スケジューリング方式の自動選択*/
20: lis_solver_set_option("-tol 1.0e-12",solver);/*収束条件‘‘1.0e-12’’*/
21: lis_solver_set_option("-i bicg -p none",solver); /*‘‘bicg 法’’, ‘‘前処理なし’’*/
22: lis_solve(A,b,x,solver);
23: /*printf() による結果出力*/
24: lis_finalize();
```

5 行列格納形式

本節では, ライブラリで使用できる行列の格納形式について述べる. 行列の行 (列) 番号は 0 から始まるものとする. 次数 $n \times n$ の行列 $A = (a_{ij})$ の非零要素数を nnz とする.

5.1 Compressed Sparse Row (CSR)

CSR 形式では, データを 3 つの配列 (ptr,index,value) に格納する.

- 長さ nnz の倍精度配列 value は, 行列 A の非零要素の値を行方向に沿って格納する.
- 長さ nnz の整数配列 index は, 配列 value に格納された非零要素の列番号を格納する.
- 長さ $n+1$ の整数配列 ptr は, 配列 value と index の各行の開始位置を格納する.

5.1.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の CSR 形式での格納方法を図 3 に示す. この行列を CSR 形式で作成する場合, プログラムは以下のように記述する.

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

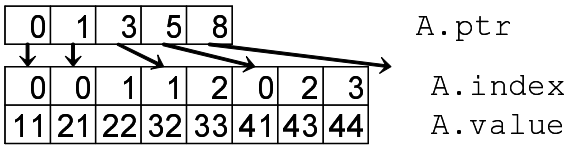

図 3: CSR 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

```
 1: LIS_INT       n,nnz;
 2: LIS_INT       *ptr,*index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; nnz = 8;
 6: ptr   = (LIS_INT *)malloc( (n+1)*sizeof(int) );
 7: index = (LIS_INT *)malloc( nnz*sizeof(int) );
 8: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
 9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11:
12: ptr[0] = 0; ptr[1] = 1; ptr[2] = 3; ptr[3] = 5; ptr[4] = 8;
13: index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  1;
14: index[4] =  2; index[5] =  0; index[6] =  2; index[7] =  3;
15: value[0] = 11; value[1] = 21; value[2] = 22; value[3] = 32;
16: value[4] = 33; value[5] = 41; value[6] = 43; value[7] = 44;
17:
18: lis_matrix_set_csr(nnz,ptr,index,value,A);
19: lis_matrix_assemble(A);
```

5.1.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の CSR 形式での格納方法を図 4 に示す. 2 プロセス上にこの行列を CSR 形式で作成する場合, プログラムは以下のように記述する.

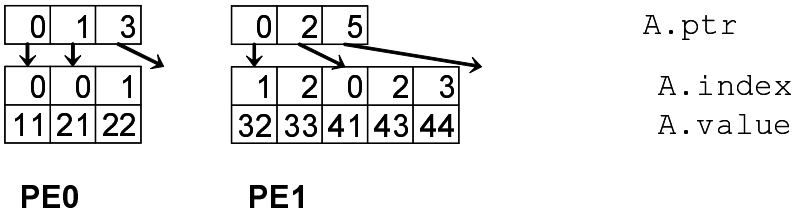

図 4: CSR 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

```
 1: LIS_INT       i,k,n,nnz,my_rank;
 2: LIS_INT       *ptr,*index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; nnz = 3;}
 7: else             {n = 2; nnz = 5;}
 8: ptr   = (LIS_INT *)malloc( (n+1)*sizeof(int) );
 9: index = (LIS_INT *)malloc( nnz*sizeof(int) );
10: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
11: lis_matrix_create(MPI_COMM_WORLD,&A);
12: lis_matrix_set_size(A,n,0);
13: if( my_rank==0 ) {
14:     ptr[0] = 0; ptr[1] = 1; ptr[2] = 3;
15:     index[0] =  0; index[1] =  0; index[2] =  1;
16:     value[0] = 11; value[1] = 21; value[2] = 22;}
17: else {
18:     ptr[0] = 0; ptr[1] = 2; ptr[2] = 5;
19:     index[0] =  1; index[1] =  2; index[2] =  0; index[3] =  2; index[4] =  3;
20:     value[0] = 32; value[1] = 33; value[2] = 41; value[3] = 43; value[4] = 44;}
21: lis_matrix_set_csr(nnz,ptr,index,value,A);
22: lis_matrix_assemble(A);
```

5.1.3 関連する関数

配列の関連付け

CSR 形式の配列を行列 A に関連付けるには, 関数

- C　　LIS_INT lis_matrix_set_csr(LIS_INT nnz, LIS_INT row[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)
- Fortran subroutine lis_matrix_set_csr(LIS_INTEGER nnz, LIS_INTEGER row(), LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

を用いる.

5.2 Compressed Sparse Column (CSC)

CSC 形式では, データを 3 つの配列 (ptr,index,value) に格納する.

- 長さ nnz の倍精度配列 value は, 行列 A の非零要素の値を列方向に沿って格納する.
- 長さ nnz の整数配列 index は, 配列 value に格納された非零要素の行番号を格納する.
- 長さ $n+1$ の整数配列 ptr は, 配列 value と index の各列の開始位置を格納する.

5.2.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の CSC 形式での格納方法を図 5 に示す. この行列を CSC 形式で作成する場合, プログラムは以下のように記述する.

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

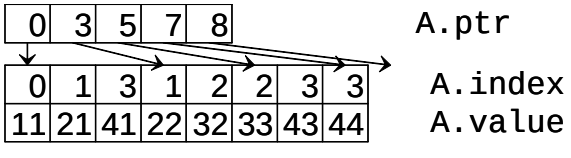

図 5: CSC 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

```
 1: LIS_INT       n,nnz;
 2: LIS_INT       *ptr,*index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; nnz = 8;
 6: ptr   = (LIS_INT *)malloc( (n+1)*sizeof(int) );
 7: index = (LIS_INT *)malloc( nnz*sizeof(int) );
 8: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
 9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11:
12: ptr[0] = 0; ptr[1] = 3; ptr[2] = 5; ptr[3] = 7; ptr[4] = 8;
13: index[0] =  0; index[1] =  1; index[2] =  3; index[3] =  1;
14: index[4] =  2; index[5] =  2; index[6] =  3; index[7] =  3;
15: value[0] = 11; value[1] = 21; value[2] = 41; value[3] = 22;
16: value[4] = 32; value[5] = 33; value[6] = 43; value[7] = 44;
17:
18: lis_matrix_set_csc(nnz,ptr,index,value,A);
19: lis_matrix_assemble(A);
```

5.2.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の CSC 形式での格納方法を図 6 に示す. 2 プロセス上にこの行列を CSC 形式で作成する場合, プログラムは以下のように記述する.

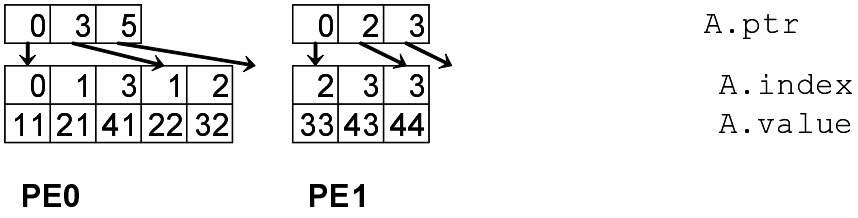

図 6: CSC 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

```
 1: LIS_INT       i,k,n,nnz,my_rank;
 2: LIS_INT       *ptr,*index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; nnz = 3;}
 7: else             {n = 2; nnz = 5;}
 8: ptr   = (LIS_INT *)malloc( (n+1)*sizeof(int) );
 9: index = (LIS_INT *)malloc( nnz*sizeof(int) );
10: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
11: lis_matrix_create(MPI_COMM_WORLD,&A);
12: lis_matrix_set_size(A,n,0);
13: if( my_rank==0 ) {
14:     ptr[0] = 0; ptr[1] = 3; ptr[2] = 5;
15:     index[0] =  0; index[1] =  1; index[2] =  3; index[3] =  1; index[4] =  2;
16:     value[0] = 11; value[1] = 21; value[2] = 41; value[3] = 22; value[4] = 32}
17: else {
18:     ptr[0] = 0; ptr[1] = 2; ptr[2] = 3;
19:     index[0] =  2; index[1] =  3; index[2] =  3;
20:     value[0] = 33; value[1] = 43; value[2] = 44;}
21: lis_matrix_set_csc(nnz,ptr,index,value,A);
22: lis_matrix_assemble(A);
```

5.2.3 関連する関数

配列の関連付け

CSC 形式の配列を行列 A に関連付けるには, 関数

- C　　LIS_INT lis_matrix_set_csc(LIS_INT nnz, LIS_INT row[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)
- Fortran subroutine lis_matrix_set_csc(LIS_INTEGER nnz, LIS_INTEGER row(), LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

を用いる.

5.3 Modified Compressed Sparse Row (MSR)

MSR 形式では, データを 2 つの配列 (index,value) に格納する. ndz を対角部分の零要素数とする.

- 長さ $nnz + ndz + 1$ の倍精度配列 value は, 第 n 要素までは行列 A の対角部分を格納する. 第 $n+1$ 要素は使用しない. 第 $n+2$ 要素からは行列 A の対角以外の非零要素の値を行方向に沿って格納する.
- 長さ $nnz + ndz + 1$ の整数配列 index は, 第 $n+1$ 要素までは行列 A の非対角部分の各行の開始位置を格納する. 第 $n+2$ 要素からは行列 A の非対角部分の配列 value に格納された非零要素の列番号を格納する.

5.3.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の MSR 形式での格納方法を図 7 に示す. この行列を MSR 形式で作成する場合, プログラムは以下のように記述する.

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

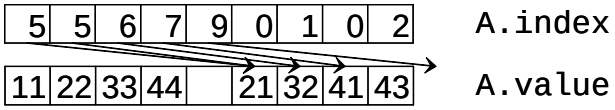

図 7: MSR 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

```
 1: LIS_INT       n,nnz,ndz;
 2: LIS_INT       *index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; nnz = 8; ndz = 0;
 6: index = (LIS_INT *)malloc( (nnz+ndz+1)*sizeof(int) );
 7: value = (LIS_SCALAR *)malloc( (nnz+ndz+1)*sizeof(LIS_SCALAR) );
 8: lis_matrix_create(0,&A);
 9: lis_matrix_set_size(A,0,n);
10:
11: index[0] =  5; index[1] =  5; index[2] =  6; index[3] =  7;
12: index[4] =  9; index[5] =  0; index[6] =  1; index[7] =  0; index[8] =  2;
13: value[0] = 11; value[1] = 22; value[2] = 33; value[3] = 44;
14: value[4] =  0; value[5] = 21; value[6] = 32; value[7] = 41; value[8] = 43;
15:
16: lis_matrix_set_msr(nnz,ndz,index,value,A);
17: lis_matrix_assemble(A);
```

5.3.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の MSR 形式での格納方法を図 8 に示す. 2 プロセス上にこの行列を MSR 形式で作成する場合, プログラムは以下のように記述する.

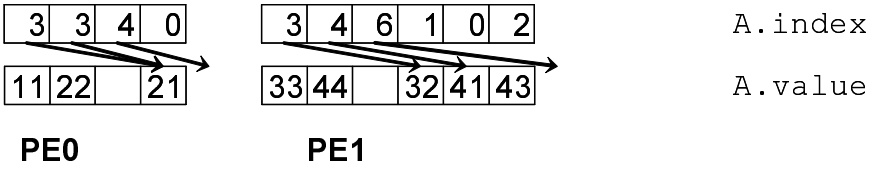

図 8: MSR 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

```
 1: LIS_INT       i,k,n,nnz,ndz,my_rank;
 2: LIS_INT       *index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; nnz = 3; ndz = 0;}
 7: else             {n = 2; nnz = 5; ndz = 0;}
 8: index = (LIS_INT *)malloc( (nnz+ndz+1)*sizeof(int) );
 9: value = (LIS_SCALAR *)malloc( (nnz+ndz+1)*sizeof(LIS_SCALAR) );
10: lis_matrix_create(MPI_COMM_WORLD,&A);
11: lis_matrix_set_size(A,n,0);
12: if( my_rank==0 ) {
13:     index[0] =  3; index[1] =  3; index[2] =  4; index[3] =  0;
14:     value[0] = 11; value[1] = 22; value[2] =  0; value[3] = 21;}
15: else {
16:     index[0] =  3; index[1] =  4; index[2] =  6; index[3] =  1;
17:     index[4] =  0; index[5] =  2;
18:     value[0] = 33; value[1] = 44; value[2] =  0; value[3] = 32;
19:     value[4] = 41; value[5] = 43;}
20: lis_matrix_set_msr(nnz,ndz,index,value,A);
21: lis_matrix_assemble(A);
```

5.3.3 関連する関数

配列の関連付け

MSR 形式の配列を行列 A に関連付けるには, 関数

- C　　LIS_INT lis_matrix_set_msr(LIS_INT nnz, LIS_INT ndz, LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)
- Fortran subroutine lis_matrix_set_msr(LIS_INTEGER nnz, LIS_INTEGER ndz, LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

を用いる.

5.4 Diagonal (DIA)

DIA 形式では, データを 2 つの配列 (index, value) に格納する. nnd を行列 A の非零な対角要素の本数とする.

- 長さ $nnd \times n$ の倍精度配列 value は, 行列 A の非零な対角要素の値を格納する.
- 長さ nnd の整数配列 index は, 主対角要素から各対角要素へのオフセットを格納する.

マルチスレッド環境では以下のように格納する.
データを 2 つの配列 (index, value) に格納する. $nprocs$ をスレッド数とする. nnd_p を行列 A を行ブロック分割した部分行列の非零な対角の本数とする. $maxnnd$ を nnd_p の値の最大値とする.

- 長さ $maxnnd \times n$ の倍精度配列 value は, 行列 A を行ブロック分割した部分行列の非零な対角要素の値を格納する.
- 長さ $nprocs \times maxnnd$ の整数配列 index は, 主対角要素から各対角要素へのオフセットを格納する.

5.4.1 行列の作成 (逐次環境)

行列 A の DIA 形式での格納方法を図 9 に示す. この行列を DIA 形式で作成する場合, プログラムは以下のように記述する.

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

-3	-1	0										A.index
0	0	0	41	0	21	32	43	11	22	33	44	A.value

図 9: DIA 形式のデータ構造 (逐次環境)

逐次環境

```
 1: LIS_INT       n,nnd;
 2: LIS_INT       *index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; nnd = 3;
 6: index = (LIS_INT *)malloc( nnd*sizeof(int) );
 7: value = (LIS_SCALAR *)malloc( n*nnd*sizeof(LIS_SCALAR) );
 8: lis_matrix_create(0,&A);
 9: lis_matrix_set_size(A,0,n);
10:
11: index[0] = -3; index[1] = -1; index[2] =  0;
12: value[0] =  0; value[1] =  0; value[2] =  0; value[3] = 41;
13: value[4] =  0; value[5] = 21; value[6] = 32; value[7] = 43;
14: value[8] = 11; value[9] = 22; value[10]= 33; value[11]= 44;
15:
16: lis_matrix_set_dia(nnd,index,value,A);
17: lis_matrix_assemble(A);
```

5.4.2 行列の作成 (マルチスレッド環境)

2 スレッド上への行列 A の DIA 形式での格納方法を図 10 に示す. 2 スレッド上にこの行列を DIA 形式で作成する場合, プログラムは以下のように記述する.

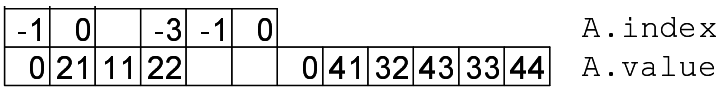

図 10: DIA 形式のデータ構造 (マルチスレッド環境)

マルチスレッド環境

```
 1: LIS_INT       n,maxnnd,nprocs;
 2: LIS_INT       *index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; maxnnd = 3; nprocs = 2;
 6: index = (LIS_INT *)malloc( maxnnd*sizeof(int) );
 7: value = (LIS_SCALAR *)malloc( n*maxnnd*sizeof(LIS_SCALAR) );
 8: lis_matrix_create(0,&A);
 9: lis_matrix_set_size(A,0,n);
10:
11: index[0] = -1; index[1] =  0; index[2] =  0; index[3] = -3; index[4] = -1; index[5] =  0;
12: value[0] =  0; value[1] = 21; value[2] = 11; value[3] = 22; value[4] =  0; value[5] =  0;
13: value[6] =  0; value[7] = 41; value[8] = 32; value[9] = 43; value[10]= 33; value[11]= 44;
14:
15: lis_matrix_set_dia(maxnnd,index,value,A);
16: lis_matrix_assemble(A);
```

5.4.3 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の DIA 形式での格納方法を図 11 に示す. 2 プロセス上にこの行列を DIA 形式で作成する場合, プログラムは以下のように記述する.

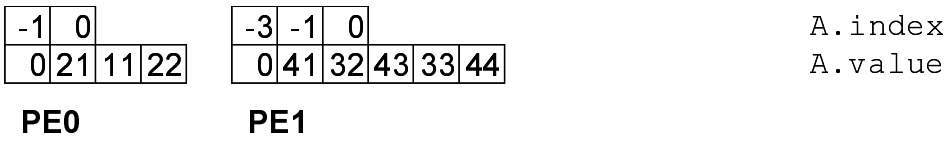

図 11: DIA 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

```
 1: LIS_INT       i,n,nnd,my_rank;
 2: LIS_INT       *index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; nnd = 2;}
 7: else             {n = 2; nnd = 3;}
 8: index = (LIS_INT *)malloc( nnd*sizeof(int) );
 9: value = (LIS_SCALAR *)malloc( n*nnd*sizeof(LIS_SCALAR) );
10: lis_matrix_create(MPI_COMM_WORLD,&A);
11: lis_matrix_set_size(A,n,0);
12: if( my_rank==0 ) {
13:     index[0] = -1; index[1] =  0;
14:     value[0] =  0; value[1] = 21; value[2] = 11; value[3] = 22;}
15: else {
16:     index[0] = -3; index[1] = -1; index[2] =  0;
17:     value[0] =  0; value[1] = 41; value[2] = 32; value[3] = 43; value[4] = 33;
18:     value[5] = 44;}
19: lis_matrix_set_dia(nnd,index,value,A);
20: lis_matrix_assemble(A);
```

5.4.4 関連する関数

配列の関連付け

DIA 形式の配列を行列 A に関連付けるには, 関数

- C　　LIS_INT lis_matrix_set_dia(LIS_INT nnd, LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)
- Fortran subroutine lis_matrix_set_dia(LIS_INTEGER nnd, LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

を用いる.

5.5 Ellpack-Itpack Generalized Diagonal (ELL)

ELL 形式では, データを 2 つの配列 (index, value) に格納する. $maxnzr$ を行列 A の各行での非零要素数の最大値とする.

- 長さ $maxnzr \times n$ の倍精度配列 value は, 行列 A の各行の非零要素の値を列方向に沿って格納する. 最初の列は各行の最初の非零要素からなる. ただし, 格納する非零要素がない場合は 0 を格納する.
- 長さ $maxnzr \times n$ の整数配列 index は, 配列 value に格納された非零要素の列番号を格納する. ただし, 第 i 行の非零要素数を nnz とすると index[$nnz \times n + i$] にはその行番号 i を格納する.

5.5.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の ELL 形式での格納方法を図 12 に示す. この行列を ELL 形式で作成する場合, プログラムは以下のように記述する.

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

0	0	1	0	0	1	2	2	0	1	2	3	A.index
11	21	32	41	0	22	33	43	0	0	0	44	A.value

図 12: ELL 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

```
 1: LIS_INT       n,maxnzr;
 2: LIS_INT       *index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; maxnzr = 3;
 6: index = (LIS_INT *)malloc( n*maxnzr*sizeof(int) );
 7: value = (LIS_SCALAR *)malloc( n*maxnzr*sizeof(LIS_SCALAR) );
 8: lis_matrix_create(0,&A);
 9: lis_matrix_set_size(A,0,n);
10:
11: index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  0; index[4] =  0; index[5] =  1;
12: index[6] =  2; index[7] =  2; index[8] =  0; index[9] =  1; index[10]=  2; index[11]=  3;
13: value[0] = 11; value[1] = 21; value[2] = 32; value[3] = 41; value[4] =  0; value[5] = 22;
14: value[6] = 33; value[7] = 43; value[8] =  0; value[9] =  0; value[10]=  0; value[11]= 44;
15:
16: lis_matrix_set_ell(maxnzr,index,value,A);
17: lis_matrix_assemble(A);
```

5.5.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の ELL 形式での格納方法を図 13 に示す. 2 プロセス上にこの行列を ELL 形式で作成する場合, プログラムは以下のように記述する.

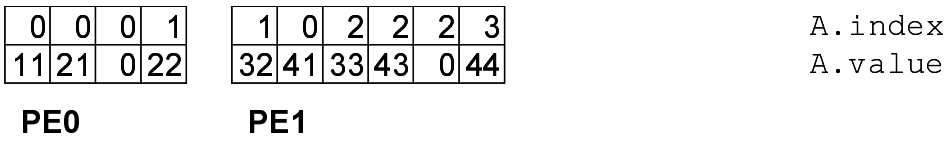

図 13: ELL 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

```
 1: LIS_INT       i,n,maxnzr,my_rank;
 2: LIS_INT       *index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; maxnzr = 2;}
 7: else             {n = 2; maxnzr = 3;}
 8: index = (LIS_INT *)malloc( n*maxnzr*sizeof(int) );
 9: value = (LIS_SCALAR *)malloc( n*maxnzr*sizeof(LIS_SCALAR) );
10: lis_matrix_create(MPI_COMM_WORLD,&A);
11: lis_matrix_set_size(A,n,0);
12: if( my_rank==0 ) {
13:     index[0] =  0; index[1] =  0; index[2] =  0; index[3] =  1;
14:     value[0] = 11; value[1] = 21; value[2] =  0; value[3] = 22;}
15: else {
16:     index[0] =  1; index[1] =  0; index[2] =  2; index[3] =  2; index[4] =  2;
17:     index[5] =  3;
18:     value[0] = 32; value[1] = 41; value[2] = 33; value[3] = 43; value[4] =  0;
19:     value[5] = 44;}
20: lis_matrix_set_ell(maxnzr,index,value,A);
21: lis_matrix_assemble(A);
```

5.5.3 関連する関数

配列の関連付け

ELL 形式の配列を行列 A に関連付けるには, 関数

- C　　LIS_INT lis_matrix_set_ell(LIS_INT maxnzr, LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)
- Fortran subroutine lis_matrix_set_ell(LIS_INTEGER maxnzr, LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

を用いる.

5.6 Jagged Diagonal (JAD)

JAD 形式では, 最初に各行の非零要素数の大きい順に行の並び替えを行い, 各行の非零要素を列方向に沿って格納する. JAD 形式では, データを 4 つの配列 (perm, ptr, index, value) に格納する. $maxnzr$ を行列 A の各行での非零要素数の最大値とする.

- 長さ n の整数配列 perm は, 並び替えた行番号を格納する.
- 長さ nnz の倍精度配列 value は, 並び替えられた行列 A の鋸歯状対角要素の値を格納する. 最初の鋸歯状対角要素は各行の第 1 非零要素からなる. 次の鋸歯状対角要素は各行の第 2 非零要素からなる. これを順次繰り返していく.
- 長さ nnz の整数配列 index は, 配列 value に格納された非零要素の列番号を格納する.
- 長さ $maxnzr + 1$ の整数配列 ptr は, 各鋸歯状対角要素の開始位置を格納する.

マルチスレッド環境では以下のように格納する.
データを 4 つの配列 (perm, ptr, index, value) に格納する. $nprocs$ をスレッド数とする. $maxnzr_p$ を行列 A を行ブロック分割した部分行列の各行での非零要素数の最大値とする. $maxmaxnzr$ は配列 $maxnzr_p$ の値の最大値である.

- 長さ n の整数配列 perm は, 行列 A を行ブロック分割した部分行列を並び替えた行番号を格納する.
- 長さ nnz の倍精度配列 value は, 並び替えられた行列 A の鋸歯状対角要素の値を格納する. 最初の鋸歯状対角要素は各行の第 1 非零要素からなる. 次の鋸歯状対角要素は各行の第 2 非零要素からなる. これを順次繰り返していく.
- 長さ nnz の整数配列 index は, 配列 value に格納された非零要素の列番号を格納する.
- 長さ $nprocs \times (maxmaxnzr + 1)$ の整数配列 ptr は, 行列 A を行ブロック分割した部分行列の各鋸歯状対角要素の開始位置を格納する.

5.6.1 行列の作成 (逐次環境)

行列 A の JAD 形式での格納方法を図 14 に示す. この行列を JAD 形式で作成する場合, プログラムは以下のように記述する.

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

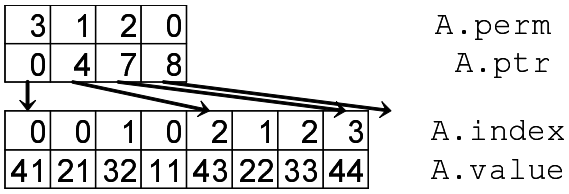

図 14: JAD 形式のデータ構造 (逐次環境)

逐次環境

```
 1: LIS_INT       n,nnz,maxnzr;
 2: LIS_INT       *perm,*ptr,*index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; nnz = 8; maxnzr = 3;
 6: perm  = (LIS_INT *)malloc( n*sizeof(int) );
 7: ptr   = (LIS_INT *)malloc( (maxnzr+1)*sizeof(int) );
 8: index = (LIS_INT *)malloc( nnz*sizeof(int) );
 9: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
10: lis_matrix_create(0,&A);
11: lis_matrix_set_size(A,0,n);
12:
13: perm[0] = 3; perm[1] = 1; perm[2] = 2; perm[3] = 0;
14: ptr[0]  = 0; ptr[1]  = 4; ptr[2]  = 7; ptr[3]  = 8;
15: index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  0;
16: index[4] =  2; index[5] =  1; index[6] =  2; index[7] =  3;
17: value[0] = 41; value[1] = 21; value[2] = 32; value[3] = 11;
18: value[4] = 43; value[5] = 22; value[6] = 33; value[7] = 44;
19:
20: lis_matrix_set_jad(nnz,maxnzr,perm,ptr,index,value,A);
21: lis_matrix_assemble(A);
```

5.6.2 行列の作成 (マルチスレッド環境)

2 スレッド上への行列 A の JAD 形式での格納方法を図 15 に示す. 2 スレッド上にこの行列を JAD 形式で作成する場合, プログラムは以下のように記述する.

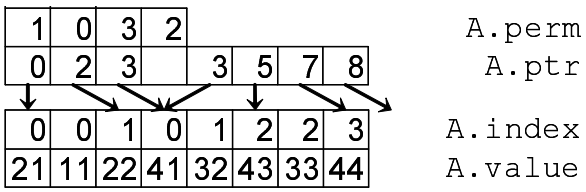

図 15: JAD 形式のデータ構造 (マルチスレッド環境)

マルチスレッド環境

```
 1: LIS_INT       n,nnz,maxmaxnzr,nprocs;
 2: LIS_INT       *perm,*ptr,*index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; nnz = 8; maxmaxnzr = 3; nprocs = 2;
 6: perm  = (LIS_INT *)malloc( n*sizeof(int) );
 7: ptr   = (LIS_INT *)malloc( nprocs*(maxmaxnzr+1)*sizeof(int) );
 8: index = (LIS_INT *)malloc( nnz*sizeof(int) );
 9: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
10: lis_matrix_create(0,&A);
11: lis_matrix_set_size(A,0,n);
12:
13: perm[0] = 1; perm[1] = 0; perm[2] = 3; perm[3] = 2;
14: ptr[0]  = 0; ptr[1]  = 2; ptr[2]  = 3; ptr[3]  = 0;
15: ptr[4]  = 3; ptr[5]  = 5; ptr[6]  = 7; ptr[7]  = 8;
16: index[0] =  0; index[1] =  0; index[2] =  1; index[3] =  0;
17: index[4] =  1; index[5] =  2; index[6] =  2; index[7] =  3;
18: value[0] = 21; value[1] = 11; value[2] = 22; value[3] = 41;
19: value[4] = 32; value[5] = 43; value[6] = 33; value[7] = 44;
20:
21: lis_matrix_set_jad(nnz,maxmaxnzr,perm,ptr,index,value,A);
22: lis_matrix_assemble(A);
```

5.6.3 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の JAD 形式での格納方法を図 16 に示す. 2 プロセス上にこの行列を JAD 形式で作成する場合, プログラムは以下のように記述する.

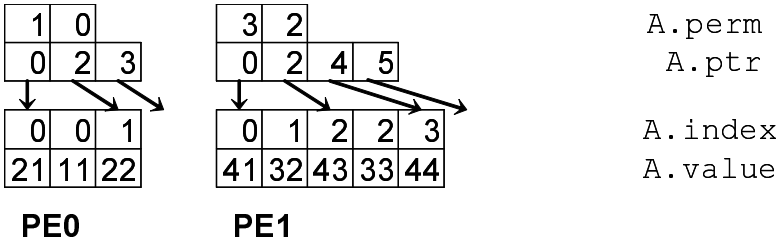

図 16: JAD 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

```
 1: LIS_INT       i,n,nnz,maxnzr,my_rank;
 2: LIS_INT       *perm,*ptr,*index;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; nnz = 3; maxnzr = 2;}
 7: else             {n = 2; nnz = 5; maxnzr = 3;}
 8: perm  = (LIS_INT *)malloc( n*sizeof(int) );
 9: ptr   = (LIS_INT *)malloc( (maxnzr+1)*sizeof(int) );
10: index = (LIS_INT *)malloc( nnz*sizeof(int) );
11: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
12: lis_matrix_create(MPI_COMM_WORLD,&A);
13: lis_matrix_set_size(A,n,0);
14: if( my_rank==0 ) {
15:     perm[0] = 1; perm[1] = 0;
16:     ptr[0]  = 0; ptr[1]  = 2; ptr[2]  = 3;
17:     index[0] =  0; index[1] =  0; index[2] =  1;
18:     value[0] = 21; value[1] = 11; value[2] = 22;}
19: else {
20:     perm[0] = 3; perm[1] = 2;
21:     ptr[0]  = 0; ptr[1]  = 2; ptr[2]  = 4; ptr[3]  = 5;
22:     index[0] =  0; index[1] =  1; index[2] =  2; index[3] =  2; index[4] =  3;
23:     value[0] = 41; value[1] = 32; value[2] = 43; value[3] = 33; value[4] = 44;}
24: lis_matrix_set_jad(nnz,maxnzr,perm,ptr,index,value,A);
25: lis_matrix_assemble(A);
```

5.6.4 関連する関数

配列の関連付け

JAD 形式の配列を行列 A に関連付けるには, 関数

- C　　LIS_INT lis_matrix_set_jad(LIS_INT nnz, LIS_INT maxnzr, LIS_INT perm[], LIS_INT ptr[], LIS_INT index[], LIS_SCALAR value[], LIS_MATRIX A)

- Fortran subroutine lis_matrix_set_jad(LIS_INTEGER nnz, LIS_INTEGER maxnzr, LIS_INTEGER ptr(), LIS_INTEGER index(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

を用いる.

5.7 Block Sparse Row (BSR)

BSR 形式では, 行列を $r \times c$ の大きさの部分行列 (ブロックと呼ぶ) に分解する. BSR 形式では, CSR 形式と同様の手順で非零ブロック (少なくとも 1 つの非零要素が存在する) を格納する. $nr = n/r$, $nnzb$ を A の非零ブロック数とする. BSR 形式では, データを 3 つの配列 (bptr, bindex, value) に格納する.

- 長さ $nnzb \times r \times c$ の倍精度配列 value は, 非零ブロックの全要素の値を格納する.
- 長さ $nnzb$ の整数配列 bindex は, 非零ブロックのブロック列番号を格納する.
- 長さ $nr + 1$ の整数配列 bptr は, 配列 bindex のブロック行の開始位置を格納する.

5.7.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の BSR 形式での格納方法を図 17 に示す. この行列を BSR 形式で作成する場合, プログラムは以下のように記述する.

$$A = \left(\begin{array}{cc|cc} 11 & & & \\ 21 & 22 & & \\ \hline & 32 & 33 & \\ 41 & & 43 & 44 \end{array}\right)$$

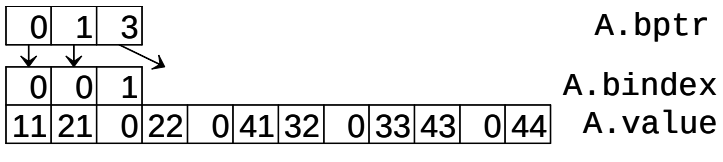

図 17: BSR 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

```
 1: LIS_INT       n,bnr,bnc,nr,nc,bnnz;
 2: LIS_INT       *bptr,*bindex;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; bnr = 2; bnc = 2; bnnz = 3; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;
 6: bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(int) );
 7: bindex = (LIS_INT *)malloc( bnnz*sizeof(int) );
 8: value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
 9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11:
12: bptr[0] = 0; bptr[1] = 1; bptr[2] = 3;
13: bindex[0] =  0; bindex[1] =  0; bindex[2] =  1;
14: value[0]  = 11; value[1] = 21; value[2] =  0; value[3] = 22;
15: value[4]  =  0; value[5] = 41; value[6] = 32; value[7] =  0;
16: value[8]  = 33; value[9] = 43; value[10]=  0; value[11]= 44;
17:
18: lis_matrix_set_bsr(bnr,bnc,bnnz,bptr,bindex,value,A);
19: lis_matrix_assemble(A);
```

5.7.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の BSR 形式での格納方法を図 18 に示す. 2 プロセス上にこの行列を BSR 形式で作成する場合, プログラムは以下のように記述する.

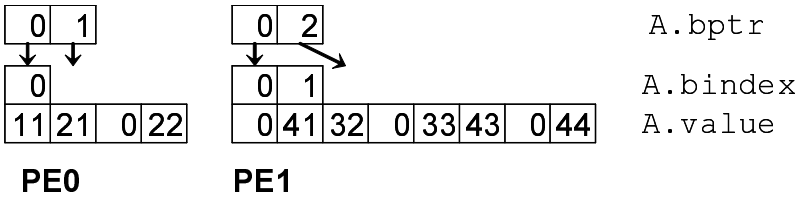

図 18: BSR 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

```
 1: LIS_INT       n,bnr,bnc,nr,nc,bnnz,my_rank;
 2: LIS_INT       *bptr,*bindex;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; bnr = 2; bnc = 2; bnnz = 1; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
 7: else             {n = 2; bnr = 2; bnc = 2; bnnz = 2; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
 8: bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(int) );
 9: bindex = (LIS_INT *)malloc( bnnz*sizeof(int) );
10: value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
11: lis_matrix_create(MPI_COMM_WORLD,&A);
12: lis_matrix_set_size(A,n,0);
13: if( my_rank==0 ) {
14:     bptr[0] = 0; bptr[1] = 1;
15:     bindex[0] =  0;
16:     value[0]  = 11; value[1] = 21; value[2] =  0; value[3] = 22;}
17: else {
18:     bptr[0] = 0; bptr[1] = 2;
19:     bindex[0] =  0; bindex[1] =  1;
20:     value[0]  =  0; value[1]  = 41; value[2] = 32; value[3] =  0;
21:     value[4]  = 33; value[5]  = 43; value[6] =  0; value[7] = 44;}
22: lis_matrix_set_bsr(bnr,bnc,bnnz,bptr,bindex,value,A);
23: lis_matrix_assemble(A);
```

5.7.3 関連する関数

配列の関連付け

BSR 形式の配列を行列 A に関連付けるには, 関数

- C　　LIS_INT lis_matrix_set_bsr(LIS_INT bnr, LIS_INT bnc, LIS_INT bnnz, LIS_INT bptr[], LIS_INT bindex[], LIS_SCALAR value[], LIS_MATRIX A)
- Fortran subroutine lis_matrix_set_bsr(LIS_INTEGER bnr, LIS_INTEGER bnc, LIS_INTEGER bnnz, LIS_INTEGER bptr(), LIS_INTEGER bindex(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

を用いる.

5.8 Block Sparse Column (BSC)

BSC 形式では, 行列を $r \times c$ の大きさの部分行列 (ブロックと呼ぶ) に分解する. BSC 形式では, CSC 形式と同様の手順で非零ブロック (少なくとも 1 つの非零要素が存在する) を格納する. $nc = n/c$, $nnzb$ を A の非零ブロック数とする. BSC 形式では, データを 3 つの配列 (bptr, bindex, value) に格納する.

- 長さ $nnzb \times r \times c$ の倍精度配列 value は, 非零ブロックの全要素の値を格納する.
- 長さ $nnzb$ の整数配列 bindex は, 非零ブロックのブロック行番号を格納する.
- 長さ $nc + 1$ の整数配列 bptr は, 配列 bindex のブロック列の開始位置を格納する.

5.8.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の BSC 形式での格納方法を図 19 に示す. この行列を BSC 形式で作成する場合, プログラムは以下のように記述する.

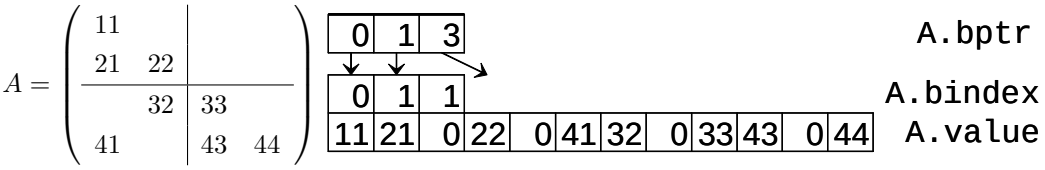

図 19: BSC 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

```
 1: LIS_INT       n,bnr,bnc,nr,nc,bnnz;
 2: LIS_INT       *bptr,*bindex;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; bnr = 2; bnc = 2; bnnz = 3; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;
 6: bptr   = (LIS_INT *)malloc( (nc+1)*sizeof(int) );
 7: bindex = (LIS_INT *)malloc( bnnz*sizeof(int) );
 8: value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
 9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11:
12: bptr[0] = 0; bptr[1] = 1; bptr[2] = 3;
13: bindex[0] =  0; bindex[1] =  1; bindex[2] =  1;
14: value[0]  = 11; value[1] = 21; value[2] =  0; value[3] = 22;
15: value[4]  =  0; value[5] = 41; value[6] = 32; value[7] =  0;
16: value[8]  = 33; value[9] = 43; value[10]=  0; value[11]= 44;
17:
18: lis_matrix_set_bsc(bnr,bnc,bnnz,bptr,bindex,value,A);
19: lis_matrix_assemble(A);
```

5.8.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の BSC 形式での格納方法を図 20 に示す. 2 プロセス上にこの行列を BSC 形式で作成する場合, プログラムは以下のように記述する.

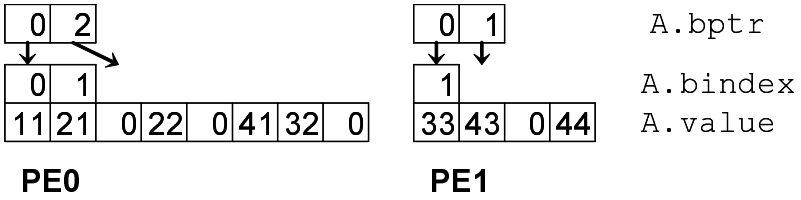

図 20: BSC 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

```
 1: LIS_INT       n,bnr,bnc,nr,nc,bnnz,my_rank;
 2: LIS_INT       *bptr,*bindex;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; bnr = 2; bnc = 2; bnnz = 2; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
 7: else             {n = 2; bnr = 2; bnc = 2; bnnz = 1; nr = (n-1)/bnr+1; nc = (n-1)/bnc+1;}
 8: bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(int) );
 9: bindex = (LIS_INT *)malloc( bnnz*sizeof(int) );
10: value  = (LIS_SCALAR *)malloc( bnr*bnc*bnnz*sizeof(LIS_SCALAR) );
11: lis_matrix_create(MPI_COMM_WORLD,&A);
12: lis_matrix_set_size(A,n,0);
13: if( my_rank==0 ) {
14:     bptr[0] = 0; bptr[1] = 2;
15:     bindex[0] =  0; bindex[1] =  1;
16:     value[0]  = 11; value[1]  = 21; value[2] =  0; value[3] = 22;
17:     value[4]  =  0; value[5]  = 41; value[6] = 32; value[7] =  0;}
18: else {
19:     bptr[0] = 0; bptr[1] = 1;
20:     bindex[0] =  1;
21:     value[0]  = 33; value[1]  = 43; value[2] =  0; value[3] = 44;}
22: lis_matrix_set_bsc(bnr,bnc,bnnz,bptr,bindex,value,A);
23: lis_matrix_assemble(A);
```

5.8.3 関連する関数

配列の関連付け

BSC 形式の配列を行列 A に関連付けるには, 関数

- C　　LIS_INT lis_matrix_set_bsc(LIS_INT bnr, LIS_INT bnc, LIS_INT bnnz, LIS_INT bptr[], LIS_INT bindex[], LIS_SCALAR value[], LIS_MATRIX A)
- Fortran subroutine lis_matrix_set_bsc(LIS_INTEGER bnr, LIS_INTEGER bnc, LIS_INTEGER bnnz, LIS_INTEGER bptr(), LIS_INTEGER bindex(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

を用いる.

5.9 Variable Block Row (VBR)

VBR 形式は BSR 形式を一般化したものである. 行と列の分割位置は配列 (`row`, `col`) で与えられる. VBR 形式では, CSR 形式と同様の手順で非零ブロック (少なくとも 1 つの非零要素が存在する) を格納する. nr, nc をそれぞれ行分割数, 列分割数とする. $nnzb$ を A の非零ブロック数, nnz を非零ブロックの全要素数とする. VBR 形式では, データを 6 つの配列 (`bptr`, `bindex`, `row`, `col`, `ptr`, `value`) に格納する.

- 長さ $nr+1$ の整数配列 `row` は, ブロック行の開始行番号を格納する.
- 長さ $nc+1$ の整数配列 `col` は, ブロック列の開始列番号を格納する.
- 長さ $nnzb$ の整数配列 `bindex` は, 非零ブロックのブロック列番号を格納する.
- 長さ $nr+1$ の整数配列 `bptr` は, 配列 `bindex` のブロック行の開始位置を格納する.
- 長さ nnz の倍精度配列 `value` は, 非零ブロックの全要素の値を格納する.
- 長さ $nnzb+1$ の整数配列 `ptr` は, 配列 `value` の非零ブロックの開始位置を格納する.

5.9.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の VBR 形式での格納方法を図 21 に示す. この行列を VBR 形式で作成する場合, プログラムは以下のように記述する.

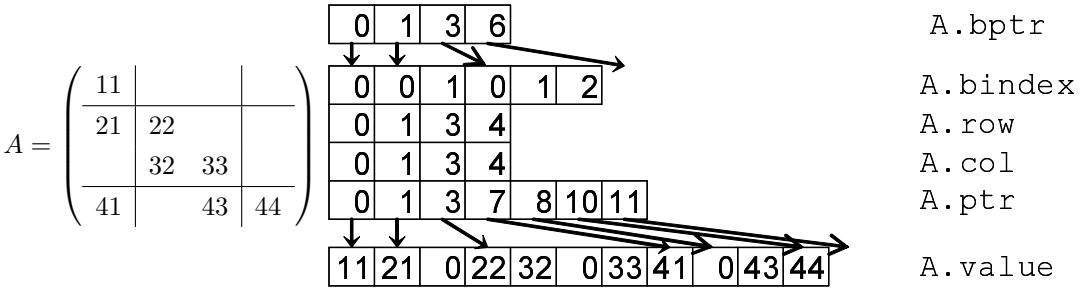

図 21: VBR 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

```
 1: LIS_INT       n,nnz,nr,nc,bnnz;
 2: LIS_INT       *row,*col,*ptr,*bptr,*bindex;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; nnz = 11; bnnz = 6; nr = 3; nc = 3;
 6: bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(int) );
 7: row    = (LIS_INT *)malloc( (nr+1)*sizeof(int) );
 8: col    = (LIS_INT *)malloc( (nc+1)*sizeof(int) );
 9: ptr    = (LIS_INT *)malloc( (bnnz+1)*sizeof(int) );
10: bindex = (LIS_INT *)malloc( bnnz*sizeof(int) );
11: value  = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
12: lis_matrix_create(0,&A);
13: lis_matrix_set_size(A,0,n);
14:
15: bptr[0] = 0; bptr[1] = 1; bptr[2] = 3; bptr[3] = 6;
16: row[0]  = 0; row[1]  = 1; row[2]  = 3; row[3] = 4;
17: col[0]  = 0; col[1]  = 1; col[2]  = 3; col[3] = 4;
18: bindex[0] =  0; bindex[1] =  0; bindex[2] =  1; bindex[3] =  0;
19: bindex[4] =  1; bindex[5] =  2;
20: ptr[0]    =  0; ptr[1]    =  1; ptr[2]    =  3; ptr[3]    =  7;
21: ptr[4]    =  8; ptr[5]    = 10; ptr[6]    = 11;
22: value[0]  = 11; value[1]  = 21; value[2]  =  0; value[3]  = 22;
23: value[4]  = 32; value[5]  =  0; value[6]  = 33; value[7]  = 41;
24: value[8]  =  0; value[9]  = 43; value[10] = 44;
25:
26: lis_matrix_set_vbr(nnz,nr,nc,bnnz,row,col,ptr,bptr,bindex,value,A);
27: lis_matrix_assemble(A);
```

5.9.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の VBR 形式での格納方法を図 22 に示す. 2 プロセス上にこの行列を VBR 形式で作成する場合, プログラムは以下のように記述する.

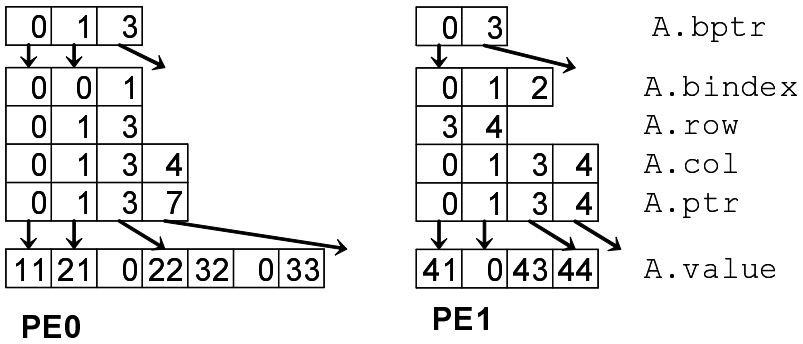

図 22: VBR 形式のデータ構造 (逐次・マルチスレッド環境)

マルチプロセス環境

```
 1: LIS_INT       n,nnz,nr,nc,bnnz,my_rank;
 2: LIS_INT       *row,*col,*ptr,*bptr,*bindex;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; nnz = 7; bnnz = 3; nr = 2; nc = 3;}
 7: else             {n = 2; nnz = 4; bnnz = 3; nr = 1; nc = 3;}
 8: bptr   = (LIS_INT *)malloc( (nr+1)*sizeof(int) );
 9: row    = (LIS_INT *)malloc( (nr+1)*sizeof(int) );
10: col    = (LIS_INT *)malloc( (nc+1)*sizeof(int) );
11: ptr    = (LIS_INT *)malloc( (bnnz+1)*sizeof(int) );
12: bindex = (LIS_INT *)malloc( bnnz*sizeof(int) );
13: value  = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
14: lis_matrix_create(MPI_COMM_WORLD,&A);
15: lis_matrix_set_size(A,n,0);
16: if( my_rank==0 ) {
17:     bptr[0] = 0; bptr[1] = 1; bptr[2] = 3;
18:     row[0]  = 0; row[1]  = 1; row[2]  = 3;
19:     col[0]  = 0; col[1]  = 1; col[2]  = 3; col[3] = 4;
20:     bindex[0] =  0; bindex[1] =  0; bindex[2] =  1;
21:     ptr[0]    =  0; ptr[1]    =  1; ptr[2]    =  3; ptr[3]    =  7;
22:     value[0]  = 11; value[1] = 21; value[2] =  0; value[3] = 22;
23:     value[4]  = 32; value[5] =  0; value[6] = 33;}
24: else {
25:     bptr[0] = 0; bptr[1] = 3;
26:     row[0]  = 3; row[1]  = 4;
27:     col[0]  = 0; col[1]  = 1; col[2]  = 3; col[3] = 4;
28:     bindex[0] =  0; bindex[1] =  1; bindex[2] =  2;
29:     ptr[0]    =  0; ptr[1]    =  1; ptr[2]    =  3; ptr[3]    =  4;
30:     value[0]  = 41; value[1]  =  0; value[2]  = 43; value[3]  = 44;}
31: lis_matrix_set_vbr(nnz,nr,nc,bnnz,row,col,ptr,bptr,bindex,value,A);
32: lis_matrix_assemble(A);
```

5.9.3 関連する関数

配列の関連付け

VBR 形式の配列を行列 A に関連付けるには, 関数

- C　　LIS_INT lis_matrix_set_vbr(LIS_INT nnz, LIS_INT nr, LIS_INT nc, LIS_INT bnnz, LIS_INT row[], LIS_INT col[], LIS_INT ptr[], LIS_INT bptr[], LIS_INT bindex[], LIS_SCALAR value[], LIS_MATRIX A)
- Fortran subroutine lis_matrix_set_vbr(LIS_INTEGER nnz, LIS_INTEGER nr, LIS_INTEGER nc, LIS_INTEGER bnnz, LIS_INTEGER row(), LIS_INTEGER col(), LIS_INTEGER ptr(), LIS_INTEGER bptr(), LIS_INTEGER bindex(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

を用いる.

5.10 Coordinate (COO)

COO 形式では, データを 3 つの配列 (row, col, value) に格納する.

- 長さ nnz の倍精度配列 value は, 非零要素の値を格納する.
- 長さ nnz の整数配列 row は, 非零要素の行番号を格納する.
- 長さ nnz の整数配列 col は, 非零要素の列番号を格納する.

5.10.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の COO 形式での格納方法を図 23 に示す. この行列を COO 形式で作成する場合, プログラムは以下のように記述する.

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

0	1	3	1	2	2	3	3	A.row
0	0	0	1	1	2	2	3	A.col
11	21	41	22	32	33	43	44	A.value

図 23: COO 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

```
 1: LIS_INT       n,nnz;
 2: LIS_INT       *row,*col;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: n = 4; nnz = 8;
 6: row   = (LIS_INT *)malloc( nnz*sizeof(int) );
 7: col   = (LIS_INT *)malloc( nnz*sizeof(int) );
 8: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
 9: lis_matrix_create(0,&A);
10: lis_matrix_set_size(A,0,n);
11:
12: row[0] = 0; row[1] = 1; row[2] = 3; row[3] = 1;
13: row[4] = 2; row[5] = 2; row[6] = 3; row[7] = 3;
14: col[0] = 0; col[1] = 0; col[2] = 0; col[3] = 1;
15: col[4] = 1; col[5] = 2; col[6] = 2; col[7] = 3;
16: value[0] = 11; value[1] = 21; value[2] = 41; value[3] = 22;
17: value[4] = 32; value[5] = 33; value[6] = 43; value[7] = 44;
18:
19: lis_matrix_set_coo(nnz,row,col,value,A);
20: lis_matrix_assemble(A);
```

5.10.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の COO 形式での格納方法を図 24 に示す. 2 プロセス上にこの行列を COO 形式で作成する場合, プログラムは以下のように記述する.

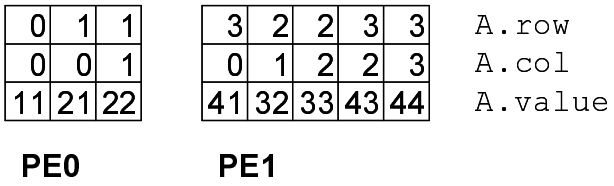

図 24: COO 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

```
 1: LIS_INT       n,nnz,my_rank;
 2: LIS_INT       *row,*col;
 3: LIS_SCALAR    *value;
 4: LIS_MATRIX    A;
 5: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 6: if( my_rank==0 ) {n = 2; nnz = 3;}
 7: else             {n = 2; nnz = 5;}
 8: row   = (LIS_INT *)malloc( nnz*sizeof(int) );
 9: col   = (LIS_INT *)malloc( nnz*sizeof(int) );
10: value = (LIS_SCALAR *)malloc( nnz*sizeof(LIS_SCALAR) );
11: lis_matrix_create(MPI_COMM_WORLD,&A);
12: lis_matrix_set_size(A,n,0);
13: if( my_rank==0 ) {
14:     row[0] = 0; row[1] = 1; row[2] = 1;
15:     col[0] = 0; col[1] = 0; col[2] = 1;
16:     value[0] = 11; value[1] = 21; value[2] = 22;}
17: else {
18:     row[0] = 3; row[1] = 2; row[2] = 2; row[3] = 3; row[4] = 3;
19:     col[0] = 0; col[1] = 1; col[2] = 2; col[3] = 2; col[4] = 3;
20:     value[0] = 41; value[1] = 32; value[2] = 33; value[3] = 43; value[4] = 44;}
21: lis_matrix_set_coo(nnz,row,col,value,A);
22: lis_matrix_assemble(A);
```

5.10.3 関連する関数

配列の関連付け

COO 形式の配列を行列 A に関連付けるには, 関数

- C　　LIS_INT lis_matrix_set_coo(LIS_INT nnz, LIS_INT row[], LIS_INT col[], LIS_SCALAR value[], LIS_MATRIX A)
- Fortran subroutine lis_matrix_set_coo(LIS_INTEGER nnz, LIS_INTEGER row(), LIS_INTEGER col(), LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)

を用いる.

5.11 Dense (DNS)

DNS 形式では, データを 1 つの配列 (value) に格納する.

- 長さ $n \times n$ の倍精度配列 value は, 列優先で要素の値を格納する.

5.11.1 行列の作成 (逐次・マルチスレッド環境)

行列 A の DNS 形式での格納方法を図 25 に示す. この行列を DNS 形式で作成する場合, プログラムは以下のように記述する.

$$A = \begin{pmatrix} 11 & & & \\ 21 & 22 & & \\ & 32 & 33 & \\ 41 & & 43 & 44 \end{pmatrix}$$

11	21	0	41	0	22	32	0
0	0	33	43	0	0	0	44

A.Value

図 25: DNS 形式のデータ構造 (逐次・マルチスレッド環境)

逐次・マルチスレッド環境

```
 1: LIS_INT       n;
 2: LIS_SCALAR    *value;
 3: LIS_MATRIX    A;
 4: n = 4;
 5: value = (LIS_SCALAR *)malloc( n*n*sizeof(LIS_SCALAR) );
 6: lis_matrix_create(0,&A);
 7: lis_matrix_set_size(A,0,n);
 8:
 9: value[0] = 11; value[1] = 21; value[2] =  0; value[3] = 41;
10: value[4] =  0; value[5] = 22; value[6] = 32; value[7] =  0;
11: value[8] =  0; value[9] =  0; value[10]= 33; value[11]= 43;
12: value[12]=  0; value[13]=  0; value[14]=  0; value[15]= 44;
13:
14: lis_matrix_set_dns(value,A);
15: lis_matrix_assemble(A);
```

5.11.2 行列の作成 (マルチプロセス環境)

2 プロセス上への行列 A の DNS 形式での格納方法を図 26 に示す. 2 プロセス上にこの行列を DNS 形式で作成する場合, プログラムは以下のように記述する.

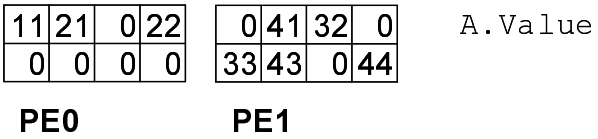

図 26: DNS 形式のデータ構造 (マルチプロセス環境)

マルチプロセス環境

```
 1: LIS_INT       n,my_rank;
 2: LIS_SCALAR    *value;
 3: LIS_MATRIX    A;
 4: MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
 5: if( my_rank==0 ) {n = 2;}
 6: else             {n = 2;}
 7: value = (LIS_SCALAR *)malloc( n*n*sizeof(LIS_SCALAR) );
 8: lis_matrix_create(MPI_COMM_WORLD,&A);
 9: lis_matrix_set_size(A,n,0);
10: if( my_rank==0 ) {
11:     value[0] = 11; value[1] = 21; value[2] =  0; value[3] = 22;
12:     value[4] =  0; value[5] =  0; value[6] =  0; value[7] =  0;}
13: else {
14:     value[0] =  0; value[1] = 41; value[2] = 32; value[3] =  0;
15:     value[4] = 33; value[5] = 43; value[6] =  0; value[7] = 44;}
16: lis_matrix_set_dns(value,A);
17: lis_matrix_assemble(A);
```

5.11.3 関連する関数

配列の関連付け

DNS 形式の配列を行列 A に関連付けるには, 関数

- C `LIS_INT lis_matrix_set_dns(LIS_SCALAR value[], LIS_MATRIX A)`
- Fortran `subroutine lis_matrix_set_dns(LIS_SCALAR value(), LIS_MATRIX A, LIS_INTEGER ierr)`

を用いる.

参考文献

[1] Lis (Library of Iterative solvers for liner systems Lis), http://www.ssisc.org/. . Experience in Developing an Open Source Scalable Software Infrastructure in Japan. Lecture Notes in Computer Science 6017, pp. 87-98, Springer, 2010.

[2] A. Nishida. Experience in Developing an Open Source Scalable Software Infrastructure in Japan. Lecture Notes in Computer Science 6017, pp. 87-98, Springer, 2010.

[3] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. Journal of Research of the National Bureau of Standards, Vol. 49, No. 6, pp. 409–436, 1952.

[4] C. Lanczos. Solution of Linear Equations by Minimized Iterations. Journal of Research of the National Bureau of Standards, Vol. 49, No. 1, pp. 33–53, 1952.

[5] R. Fletcher. Conjugate Gradient Methods for Indefinite Systems. Lecture Notes in Mathematics 506, pp. 73–89, Springer, 1976.

[6] T. Sogabe, M. Sugihara, and S. Zhang. An Extension of the Conjugate Residual Method to Nonsymmetric Linear Systems. Journal of Computational and Applied Mathematics, Vol. 226, No. 1, pp. 103–113, 2009.

[7] P. Sonneveld. CGS, A Fast Lanczos-Type Solver for Nonsymmetric Linear Systems. SIAM Journal on Scientific and Statistical Computing, Vol. 10, No. 1, pp. 36–52, 1989.

[8] K. Abe, T. Sogabe, S. Fujino, and S. Zhang. A Product-Type Krylov Subspace Method Based on Conjugate Residual Method for Nonsymmetric Coefficient Matrices (in Japanese). IPSJ Transactions on Advanced Computing Systems, Vol. 48, No. SIG8(ACS18), pp. 11–21, 2007.

[9] H. van der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. SIAM Journal on Scientific and Statistical Computing, Vol. 13, No. 2, pp. 631–644, 1992.

[10] S. Zhang. Generalized Product-Type Methods Preconditionings Based on Bi-CG for Solving Nonsymmetric Linear Systems. SIAM Journal on Scientific Computing, Vol. 18, No. 2, pp. 537–551, 1997.

[11] S. Fujino, M. Fujiwara, and M. Yoshida. A Proposal of Preconditioned BiCGSafe Method with Safe Convergence. Proceedings of The 17th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation, CD-ROM, 2005.

[12] S. Fujino and Y. Onoue. Estimation of BiCRSafe Method Based on Residual of BiCR Method (in Japanese). IPSJ SIG Technical Report, 2007-HPC-111, pp. 25–30, 2007.

[13] G. L. G. Sleijpen, H. A. van der Vorst, and D. R. Fokkema. BiCGstab(l) and Other Hybrid Bi-CG Methods. Numerical Algorithms, Vol. 7, No. 1, pp. 75–109, 1994.

[14] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. SIAM Journal on Scientific Computing, Vol. 14, No. 2, pp. 470–482, 1993.

[15] K. R. Biermann. Eine unveröffentlichte Jugendarbeit C. G. J. Jacobi über wiederholte Funktionen. Journal für die reine und angewandte Mathematik, Vol. 207, pp. 996-112, 1961.

[16] S. C. Eisenstat, H. C. Elman, and M. H. Schultz. Variational Iterative Methods for Nonsymmetric Systems of Linear Equations. SIAM Journal on Numerical Analysis, Vol. 20, No. 2, pp. 345–357, 1983.

[17] C. F. Gauss. Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem. Perthes et Besser, 1809.

[18] L. Seidel. Über ein Verfahren, die Gleichungen, auf welche die Methode der kleinsten Quadrate führt, sowie lineäre Gleichungen überhaupt, durch successive Annäherung aufzulösen. Abhandlungen der Bayerischen Akademie, Vol. 11, pp. 81–108, 1873.

[19] Y. Saad and M. H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. SIAM Journal on Scientific and Statistical Computing, Vol. 7, No. 3, pp. 856–869, 1986.

[20] D. M. Young. Iterative Methods for Solving Partial Difference Equations of Elliptic Type. Doctoral Thesis, Harvard University, 1950.

[21] S. P. Frankel. Convergence Rates of Iterative Treatments of Partial Differential Equations. Mathematical Tables and Other Aids to Computation, Vol. 4, No. 30, pp. 65–75, 1950.

[22] Y. Saad. A Flexible Inner-outer Preconditioned GMRES Algorithm. SIAM Journal on Scientific and Statistical Computing, Vol. 14, No. 2, pp. 461–469, 1993.

[23] P. Sonnerveld and M. B. van Gijzen. IDR(s): A Family of Simple and Fast Algorithms for Solving Large Nonsymmetric Systems of Linear Equations. SIAM Journal on Scientific Computing, Vol. 31, No. 2, pp. 1035–1062, 2008.

[24] C. C. Paige and M. A. Saunders. Solution of Sparse Indefinite Systems of Linear Equations. SIAM Journal on Numerical Analysis, Vol. 12, No. 4, pp. 617–629, 1975.

[25] R. von Mises and H. Pollaczek-Geiringer. Praktische Verfahren der Gleichungsauflösung. Zeitschrift für Angewandte Mathematik und Mechanik, Vol. 9, No. 2, pp. 152–164, 1929.

[26] H. Wielandt. Beiträge zur mathematischen Behandlung komplexer Eigenwertprobleme, Teil V: Bestimmung höherer Eigenwerte durch gebrochene Iteration. Bericht B 44/J/37, Aerodynamische Versuchsanstalt Göttingen, 1944.

[27] J. W. S. Rayleigh. Some General Theorems relating to Vibrations. Proceedings of the London Mathematical Society, Vol. 4, No. 1, pp. 357–368, 1873.

[28] H. R. Rutishauser. Computational Aspects of F. L. Bauser's Simultaneous Iteration Method. Numerische Mathematik, Vol. 13, No. 1, pp. 4–13, 1969.

[29] C. Lanczos. An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators. Journal of Research of the National Bureau of Standards, Vol. 45, No. 4, pp. 255–282, 1950.

[30] A. V. Knyazev. Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. SIAM Journal on Scientific Computing, Vol. 23, No. 2, pp. 517-541, 2001.

[31] E. Suetomi and H. Sekimoto. Conjugate Gradient Like Methods and Their Application to Eigenvalue Problems for Neutron Diffusion Equation. Annals of Nuclear Energy, Vol. 18, No. 4, pp. 205-227, 1991.

[32] O. Axelsson. A Survey of Preconditioned Iterative Methods for Linear Systems of Equations. BIT, Vol. 25, No. 1, pp. 166–187, 1985.

[33] I. Gustafsson. A Class of First Order Factorization Methods. BIT, Vol. 18, No. 2, pp. 142–156, 1978.

[34] Y. Saad. ILUT: A Dual Threshold Incomplete LU Factorization. Numerical Linear Algebra with Applications, Vol. 1, No. 4, pp. 387–402, 1994.

[35] Y. Saad, et al. ITSOL: ITERATIVE SOLVERS Package. http://www-users.cs.umn.edu/~saad/software/ITSOL/.

[36] N. Li, Y. Saad, and E. Chow. Crout Version of ILU for General Sparse Matrices. SIAM Journal on Scientific Computing, Vol. 25, No. 2, pp. 716–728, 2003.

[37] T. Kohno, H. Kotakemori, and H. Niki. Improving the Modified Gauss-Seidel Method for Z-matrices. Linear Algebra and its Applications, Vol. 267, pp. 113–123, 1997.

[38] A. Fujii, A. Nishida, and Y. Oyanagi. Evaluation of Parallel Aggregate Creation Orders : Smoothed Aggregation Algebraic Multigrid Method. High Performance Computational Science and Engineering, pp. 99–122, Springer, 2005.

[39] K. Abe, S. Zhang, H. Hasegawa, and R. Himeno. A SOR-base Variable Preconditioned CGR Method (in Japanese). Transactions of the JSIAM, Vol. 11, No. 4, pp. 157–170, 2001.

[40] R. Bridson and W. P. Tang. Refining an Approximate Inverse. Journal of Computational and Applied Mathematics, Vol. 123, No. 1-2, pp. 293–306, 2000.

[41] T. Chan and T. Mathew. Domain Decomposition Algorithms. Acta Numerica, Vol. 3, pp. 61–143, 1994.

[42] M. Dryja and O. B. Widlund. Domain Decomposition Algorithms with Small Overlap. SIAM Journal on Scientific Computing, Vol. 15, No. 3, pp. 604–620, 1994.

[43] H. Kotakemori, H. Hasegawa, and A. Nishida. Performance Evaluation of a Parallel Iterative Method Library using OpenMP. Proceedings of the 8th International Conference on High Performance Computing in Asia Pacific Region, pp. 432–436, IEEE, 2005.

[44] H. Kotakemori, H. Hasegawa, T. Kajiyama, A. Nukada, R. Suda, and A. Nishida. Performance Evaluation of Parallel Sparse Matrix-Vector Products on SGI Altix 3700. Lecture Notes in Computer Science 4315, pp. 153–163, Springer, 2008.

[45] D. H. Bailey. A Fortran-90 Double-Double Library. http://crd-legacy.lbl.gov/~dhbailey/mpdist/.

[46] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for Quad-Double Precision Floating Point Arithmetic. Proceedings of the 15th Symposium on Computer Arithmetic, pp. 155–162, 2001.

[47] T. Dekker. A Floating-Point Technique for Extending the Available Precision. Numerische Mathematik, Vol. 18, No. 3, pp. 224–242, 1971.

[48] D. E. Knuth. The Art of Computer Programming: Seminumerical Algorithms, Vol. 2. Addison-Wesley, 1969.

[49] D. H. Bailey. High-Precision Floating-Point Arithmetic in Scientific Computation. Computing in Science and Engineering, Vol. 7, No. 3, pp. 54–61, IEEE, 2005.

[50] Intel Fortran Compiler for Linux Systems User's Guide, Vol I. Intel Corporation, 2004.

[51] H. Kotakemori, A. Fujii, H. Hasegawa, and A. Nishida. Implementation of Fast Quad Precision Operation and Acceleration with SSE2 for Iterative Solver Library (in Japanese). IPSJ Transactions on Advanced Computing Systems, Vol. 1, No. 1, pp. 73–84, 2008.

[52] R. Courant and D. Hilbert. Methods of Mathematical Physics. Wiley-VCH, 1989.

[53] C. Lanczos. The Variational Principles of Mechanics, 4th Edition. University of Toronto Press, 1970.

[54] J. H. Wilkinson. The Algebraic Eigenvalue Problem. Oxford University Press, 1988.

[55] D. M. Young. Iterative Solution of Large Linear Systems. Academic Press, 1971.

[56] G. H. Golub and C. F. Van Loan. Matrix Computations, 3rd Edition. The Johns Hopkins University Press, 1996.

[57] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. Solving Linear Systems on Vector and Shared Memory Computers. SIAM, 1991.

[58] Y. Saad. Numerical Methods for Large Eigenvalue Problems. Halsted Press, 1992.

[59] R. Barrett, et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, 1994.

[60] Y. Saad. Iterative Methods for Sparse Linear Systems. Second Edition. SIAM, 2003.

[61] A. Greenbaum. Iterative Methods for Solving Linear Systems. SIAM, 1997.

[62] Z. Bai, et al. Templates for the Solution of Algebraic Eigenvalue Problems. SIAM, 2000.

[63] J. H. Wilkinson and C. Reinsch. Handbook for Automatic Computation, Vol. 2: Linear Algebra. Grundlehren Der Mathematischen Wissenschaften, Vol. 186, Springer, 1971.

[64] B. T. Smith, J. M. Boyle, Y. Ikebe, V. C. Klema, and C. B. Moler. Matrix Eigensystem Routines: EISPACK Guide, 2nd ed. Lecture Notes in Computer Science 6, Springer, 1970.

[65] B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler. Matrix Eigensystem Routines: EISPACK Guide Extension. Lecture Notes in Computer Science 51, Springer, 1972.

[66] J. J. Dongarra, J. R. Bunch, G. B. Moler, and G. M. Stewart. LINPACK Users' Guide. SIAM, 1979.

[67] J. R. Rice and R. F. Boisvert. Solving Elliptic Problems Using ELLPACK. Springer, 1985.

[68] E. Anderson, et al. LAPACK Users' Guide. 3rd ed. SIAM, 1987.

[69] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A Sparse Matrix Library in C++ for High Performance Architectures. Proceedings of the Second Object Oriented Numerics Conference, pp. 214–218, 1992.

[70] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' Guide for the Harwell-Boeing Sparse Matrix Collection (Release I). Technical Report TR/PA/92/86, CERFACS, 1992.

[71] Y. Saad. SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations, Version 2, 1994. http://www-users.cs.umn.edu/~saad/software/SPARSKIT/.

[72] A. Geist, et al. PVM: Parallel Virtual Machine. MIT Press, 1994.

[73] R. Bramley and X. Wang. SPLIB: A library of Iterative Methods for Sparse Linear System. Technical Report, Department of Computer Science, Indiana University, 1995.

[74] R. F. Boisvert, et al. The Matrix Market Exchange Formats: Initial Design. Technical Report NISTIR 5935, National Institute of Standards and Technology, 1996.

[75] L. S. Blackford, et al. ScaLAPACK Users' Guide. SIAM, 1997.

[76] R. B. Lehoucq, D. C. Sorensen, and C. Yang. ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly-Restarted Arnoldi Methods. SIAM, 1998.

[77] R. S. Tuminaro, et al. Official Aztec User's Guide, Version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, 1999.

[78] W. Gropp, E. Lusk, and A. Skjellum. Using MPI, 2nd Edition: Portable Parallel Programming with the Message-Passing Interface. MIT Press, 1999.

[79] S. Balay, et al. PETSc Users Manual. Technical Report ANL-95/11, Argonne National Laboratory, 2004.

[80] V. Hernandez, J. E. Roman, and V. Vidal. SLEPc: A Scalable and Flexible Toolkit for the Solution of Eigenvalue Problems. ACM Transactions on Mathematical Software, Vol. 31, No. 3, pp. 351–362, 2005.

[81] M. A. Heroux, et al. An Overview of the Trilinos Project. ACM Transactions on Mathematical Software, Vol. 31, No. 3, pp. 397–423, 2005.

[82] R. D. Falgout, J. E. Jones, and U. M. Yang. The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners. Lecture Notes in Computational Science and Engineering 51, pp. 209–236, Springer, 2006.

[83] B. Chapman, G. Jost, and R. van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, 2007.

[84] J. Dongarra and M. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report SAND2013-4744, Sandia National Laboratories, 2013.

[85] B. Chapman, G. Jost, and R. van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, 2007.

[86] Hishinuma, T., Fujii, A., Tanaka, T., and Hasegawa, H.: AVX acceleration of DD arithmetic between a sparse matrix and vector, the Tenth International Conference on Parallel Processing and Applied Mathematics, Warsaw, Poland (2013).

[87] 菱沼利彰，藤井昭宏，田中輝雄，長谷川秀彦: AVX2 を用いた倍精度 BCRS 形式疎行列と倍々精度ベクトル積の高速化，情報処理学会論文誌 コンピューティングシステム，Vol.7, No.4, pp.1-10 (2014).

[88] 菱沼利彰，田中輝雄，長谷川秀彦: 疎行列ベクトル積に対する OpenMP スケジューリング方式の分析 2014 年ハイパフォーマンスコンピューティングと計算科学シンポジウム，p.31 (2014).

A ファイル形式

本節では, 本ライブラリで使用できるファイル形式について述べる.

A.1 拡張 Matrix Market 形式

Matrix Market 形式はベクトルデータの格納に対応していないため, 拡張 Matrix Market 形式では行列とベクトルを合わせて格納できるよう仕様を拡張する. $M \times N$ の行列 $A = (a_{ij})$ の非零要素数を L とする. $a_{ij} = A(I, J)$ とする. ファイル形式を以下に示す.

```
%%MatrixMarket matrix coordinate real general  <-- ヘッダ
%                                              <-+
%                                                | 0 行以上のコメント行
%                                              <-+
M N L B X                                      <-- 行数 列数 非零数 (0 or 1) (0 or 1)
I1 J1 A(I1,J1)                                 <-+
I2 J2 A(I2,J2)                                   | 行番号 列番号 値
 . . .                                           | インデックスは 1-base
IL JL A(IL,JL)                                 <-+
I1 B(I1)                                       <-+
I2 B(I2)                                         | B=1 の場合のみ存在する
 . . .                                           | 行番号 値
IM B(IM)                                       <-+
I1 X(I1)                                       <-+
I2 X(I2)                                         | X=1 の場合のみ存在する
 . . .                                           | 行番号 値
IM X(IM)                                       <-+
```

(A.1) 式の行列 A とベクトル b に対するファイル形式を以下に示す.

$$A = \begin{pmatrix} 2 & 1 & & \\ 1 & 2 & 1 & \\ & 1 & 2 & 1 \\ & & 1 & 2 \end{pmatrix} \qquad b = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} \tag{A.1}$$

```
%%MatrixMarket matrix coordinate real general
4 4 10 1 0
1 2  1.00e+00
1 1  2.00e+00
2 3  1.00e+00
2 1  1.00e+00
2 2  2.00e+00
3 4  1.00e+00
3 2  1.00e+00
3 3  2.00e+00
4 4  2.00e+00
4 3  1.00e+00
1  0.00e+00
2  1.00e+00
3  2.00e+00
4  3.00e+00
```

A.2 Harwell-Boeing 形式

Harwell-Boeing 形式では, CSC 形式で行列を格納する. value を行列 A の非零要素の値, index を非零要素の行番号, ptr を value と index の各列の開始位置を格納する配列とする. ファイル形式を以下に示す.

```
第 1 行 (A72,A8)
   1 - 72 Title
  73 - 80 Key
第 2 行 (5I14)
   1 - 14 ヘッダを除く総行数
  15 - 28 ptr の行数
  29 - 42 index の行数
  43 - 56 value の行数
  57 - 70 右辺の行数
第 3 行 (A3,11X,4I14)
   1 -  3 行列の種類
            第 1 列: R Real matrix
                   C Complex matrix (非対応)
                   P Pattern only (非対応)
            第 2 列: S Symmetric
                   U Unsymmetric
                   H Hermitian (非対応)
                   Z Skew symmetric (非対応)
                   R Rectangular (非対応)
            第 3 列: A Assembled
                   E Elemental matrices (非対応)
   4 - 14 空白
  15 - 28 行数
  29 - 42 列数
  43 - 56 非零要素数
  57 - 70 0
第 4 行 (2A16,2A20)
   1 - 16 ptr の形式
  17 - 32 index の形式
  33 - 52 value の形式
  53 - 72 右辺の形式
第 5 行 (A3,11X,2I14) 右辺が存在する場合
   1      右辺の種類
            F フルベクトル
            M 行列と同じ形式 (非対応)
   2      初期値が与えられるならば G
   3      解が与えられるならば X
   4 - 14 空白
  15 - 28 右辺の数
  29 - 42 非零要素数
```

(A.1) 式の行列 A とベクトル b に対するファイル形式を以下に示す.

```
1--------10--------20--------30--------40--------50--------60--------70--------80
Harwell-Boeing format sample                                           Lis
             8             1             1             4             2
RUA                        4             4            10             4
(11i7)          (13i6)          (3e26.18)           (3e26.18)
F                          1             0
      1      3      6      9
     1     2     1     2     3     2     3     4     3     4
  2.00000000000000000E+00  1.00000000000000000E+00  1.00000000000000000E+00
  2.00000000000000000E+00  1.00000000000000000E+00  1.00000000000000000E+00
```

```
  2.00000000000000000E+00  1.00000000000000000E+00  1.00000000000000000E+00
  2.00000000000000000E+00
  0.00000000000000000E+00  1.00000000000000000E+00  2.00000000000000000E+00
  3.00000000000000000E+00
```

A.3 ベクトル用拡張 Matrix Market 形式

ベクトル用拡張 Matrix Market 形式では, ベクトルデータを格納できるよう Matrix Market 形式の仕様を拡張する. 次数 N のベクトル $b = (b_i)$ に対して $b_i = B(I)$ とする. ファイル形式を以下に示す.

```
%%MatrixMarket vector coordinate real general  <-- ヘッダ
%                                              <-+
%                                                | 0 行以上のコメント行
%                                              <-+
N                                              <-- 行数
I1 B(I1)                                       <-+
I2 B(I2)                                         | 行番号 値
 . . .                                           | インデックスは 1-base
IN B(IN)                                       <-+
```

(A.1) 式のベクトル b に対するファイル形式を以下に示す.

```
%%MatrixMarket vector coordinate real general
4
1  0.00e+00
2  1.00e+00
3  2.00e+00
4  3.00e+00
```

A.4 ベクトル用 PLAIN 形式

ベクトル用 PLAIN 形式は, ベクトルの値を第 1 要素から順に書き出したものである. 次数 N のベクトル $b = (b_i)$ に対して $b_i = B(I)$ とする. ファイル形式を以下に示す.

```
B(1)                                           <-+
B(2)                                             | N 個
. . .                                            |
B(N)                                           <-+
```

(A.1) 式のベクトル b に対するファイル形式を以下に示す.

```
0.00e+00
1.00e+00
2.00e+00
3.00e+00
```