

Lis 1.1.2 User's Manual

Copyright (C) 2002-2007 The Scalable Software Infrastructure Project,
supported by “Development of Software Infrastructure for Large Scale Scientific Simula-
tion” Team, CREST, JST. <http://www.ssisc.org/>

Last update : December 24, 2007

Contents

0	Additions and changes from Lis 1.0	1
0.1	Additions	1
0.2	Changes	1
1	Introduction	2
2	Installation	3
2.1	Required Systems	3
2.2	Decompressing the Files	3
2.3	Executing <code>configure</code> script	4
2.4	Executing <code>make</code>	4
2.5	Testing	4
2.6	Installation	6
2.7	Test Programs	8
2.7.1	Test 1	8
2.7.2	Test 2	8
2.7.3	Test 3	8
2.7.4	Test 4	9
2.8	Restrictions	10
3	Basic Operation	11
3.1	Initialization and Finalization	12
3.2	Vector	12
3.3	Matrix	15
3.4	Solution	21
3.5	Sample Program	23
3.6	Compiling and Linking	25
3.7	Execution	27
4	Quadruple precision operation	28
4.1	Executing quadruple precision operation	28
5	Matrix Storage Format	29
5.1	Compressed Row Storage (CRS)	29
5.1.1	How to Create a Matrix (Sequential and OpenMP)	29
5.1.2	How to Create a Matrix (MPI)	30
5.1.3	Pertinent Function	30
5.2	Compressed Column Storage (CCS)	31
5.2.1	How to Create a Matrix (Sequential and OpenMP)	31
5.2.2	How to Create a Matrix (MPI)	32
5.2.3	Pertinent Function	32
5.3	Modified Compressed Sparse Row (MSR)	33
5.3.1	How to Create a Matrix (Sequential and OpenMP)	33
5.3.2	How to Create a Matrix (MPI)	34
5.3.3	Pertinent Function	34
5.4	Diagonal (DIA)	35
5.4.1	How to Create a Matrix (Sequential)	35
5.4.2	How to Create a Matrix (OpenMP)	36
5.4.3	How to Create a Matrix (MPI)	37
5.4.4	Pertinent Function	37
5.5	Ellpack-Itpack generalized diagonal (ELL)	38
5.5.1	How to Create a Matrix (Sequential and OpenMP)	38

5.5.2	How to Create a Matrix (MPI)	39
5.5.3	Pertinent Function	39
5.6	Jagged Diagonal (JDS)	40
5.6.1	How to Create a Matrix (Sequential)	41
5.6.2	How to Create a Matrix (OpenMP)	42
5.6.3	How to Create a Matrix (MPI)	43
5.6.4	Pertinent Function	43
5.7	Block Sparse Row (BSR)	44
5.7.1	How to Create a Matrix (Sequential and OpenMP)	44
5.7.2	How to Create a Matrix (MPI)	45
5.7.3	Pertinent Function	45
5.8	Block Sparse Column (BSC)	46
5.8.1	How to Create a Matrix (Sequential and OpenMP)	46
5.8.2	How to Create a Matrix (MPI)	47
5.8.3	Pertinent Function	47
5.9	Variable Block Row (VBR)	48
5.9.1	How to Create a Matrix (Sequential and OpenMP)	48
5.9.2	How to Create a Matrix (MPI)	49
5.9.3	Pertinent Function	50
5.10	Coordinate (COO)	51
5.10.1	How to Create a Matrix (Sequential and OpenMP)	51
5.10.2	How to Create a Matrix (MPI)	52
5.10.3	Pertinent Function	52
5.11	Dense (DNS)	53
5.11.1	How to Create a Matrix (Sequential and OpenMP)	53
5.11.2	How to Create a Matrix (MPI)	54
5.11.3	Pertinent Function	54
6	Functions	55
6.1	Vector Operations	55
6.1.1	lis_vector_create	55
6.1.2	lis_vector_destroy	56
6.1.3	lis_vector_duplicate	56
6.1.4	lis_vector_set_size	57
6.1.5	lis_vector_get_size	57
6.1.6	lis_vector_get_range	58
6.1.7	lis_vector_set_value	58
6.1.8	lis_vector_get_value	59
6.1.9	lis_vector_copy	59
6.1.10	lis_vector_set_all	60
6.2	Matrix Operations	61
6.2.1	lis_matrix_create	61
6.2.2	lis_matrix_destroy	61
6.2.3	lis_matrix_duplicate	62
6.2.4	lis_matrix_malloc	62
6.2.5	lis_matrix_set_value	63
6.2.6	lis_matrix_assemble	63
6.2.7	lis_matrix_set_size	64
6.2.8	lis_matrix_get_size	64
6.2.9	lis_matrix_get_range	65
6.2.10	lis_matrix_set_type	66
6.2.11	lis_matrix_get_type	66
6.2.12	lis_matrix_set_blocksize	67

6.2.13	lis_matrix_convert	67
6.2.14	lis_matrix_copy	68
6.2.15	lis_matrix_get_diagonal	68
6.2.16	lis_matrix_set_crs	68
6.2.17	lis_matrix_set_ccs	69
6.2.18	lis_matrix_set_msr	69
6.2.19	lis_matrix_set_dia	70
6.2.20	lis_matrix_set_ell	70
6.2.21	lis_matrix_set_jds	71
6.2.22	lis_matrix_set_bsr	71
6.2.23	lis_matrix_set_bsc	72
6.2.24	lis_matrix_set_vbr	72
6.2.25	lis_matrix_set_coo	73
6.2.26	lis_matrix_set_dns	73
6.3	Vector and Matrix Calculations	74
6.3.1	lis_vector_scale	74
6.3.2	lis_vector_dot	74
6.3.3	lis_vector_nrm2	74
6.3.4	lis_vector_nrm1	75
6.3.5	lis_vector_axpy	75
6.3.6	lis_vector_xpay	75
6.3.7	lis_vector_axpyz	76
6.3.8	lis_matrix_scaling	76
6.3.9	lis_matvec	77
6.3.10	lis_matvect	77
6.4	Solve	78
6.4.1	lis_solver_create	78
6.4.2	lis_solver_destroy	78
6.4.3	lis_solver_set_option	79
6.4.4	lis_solver_set_optionC	81
6.4.5	lis_solve	81
6.4.6	lis_solver_get_iters	82
6.4.7	lis_solver_get_itersex	82
6.4.8	lis_solver_get_time	83
6.4.9	lis_solver_get_timeex	83
6.4.10	lis_solver_get_residualnorm	84
6.4.11	lis_get_residual_history	84
6.4.12	lis_solver_get_solver	85
6.4.13	lis_solver_get_solvername	85
6.5	File Input and Output	86
6.5.1	lis_input	86
6.5.2	lis_input_vector	86
6.5.3	lis_output	87
6.5.4	lis_output_vector	87
6.5.5	lis_output_residual_history	88
6.6	Other Functions	89
6.6.1	lis_initialize	89
6.6.2	lis_finalize	89
6.6.3	lis_wtime	89

References

90

A	Appendix A: File Formats	92
A.1	MatrixMarket Format	92
A.2	Harwell-Boeing Format	92
A.3	MatrixMarketFormat (for Vectors)	94
A.4	PLAIN Format (for Vectors)	94

0 Additions and changes from Lis 1.0

0.1 Additions

1. Added support of quadruple precision operations in iterative methods
2. Added support of Fortran user interface
3. Added support of the nonsymmetric SA-AMG preconditioner (symmetric structure only)
4. Added support of the ILUT and Crout ILU preconditioner
5. Added support of the additive Schwarz preconditioner
6. Added support of user-defined preconditioners
7. Added support of the Harwell-Boeing format
8. Added support of the BiCGSafe, CR, BiCR, CRS, BiCRSTAB, GPBiCR, BiCRSafe, FGMRES, and IDR(s) iterative methods

0.2 Changes

1. Integrated Lis-AMG into Lis
2. Introduced autoconf
3. Changed a number of specifications
 - (a) Introduced the LIS_SOLVER structure (integrated type parameters)
 - (b) Changed the declaration of matrices and vectors
 - (c) Changed the notation of command line options

1 Introduction

Library of Iterative Solvers for Linear Systems (Lis) is a library written in the C and Fortran 90 languages for solving the linear equation

$$Ax = b$$

using an iterative method. Installation of Lis requires a computing environment in which a C compiler can be used. In addition, the use of AMG preconditioner routines requires a compiler that supports Fortran 90. In a parallel computing environment, OpenMP or MPI-1 is used.

Lis has the following features:

- 20 iterative methods and 10 preconditioners can be combined and used
- 11 sparse matrix storage formats can be used
- The sequential and the parallel environment can be used by a common interface
- Double precision and quadruple precision can be used by a common interface

The iterative methods adopted here are intended for linear equations with large coefficient matrixes as real matrixes, which covers approximately 13 types of methods, including stationary (e.g., SOR) and non-stationary (e.g., GPBiCG) methods. The preconditioners adopted include a $I + S$ type[9], which is effective for stationary iterative methods; algebraic multi-grid SA-AMG[10] based on smoothed aggregation; a hybrid method[11] based on iterative methods such as SOR; the approximate inverse matrix SAINV[12], which approximates the inverse matrix A^{-1} itself based on A-Orthogonalization; and the Crout ILU preconditioner, which can decompose more stably than traditional ILU[8], in addition to scaling and incomplete LU decomposition, which are well-known methods, as shown in Table 2. Eleven types of data storage methods, including CRS, are used, as shown in Table 3.

Table 1: Iterative Methods	Table 2: Preconditioner	Table 3: Matrix Storage
CG	CR	Compressed Row Storage (CRS)
BiCG	BiCR[2]	Compressed Column Storage (CCS)
CGS	CRS[3]	Modified Compressed Sparse Row (MSR)
BiCGSTAB	BiCRSTAB[3]	Diagonal (DIA)
GPBiCG	GPBiCR[3]	Ellpack-Itpack generalized diagonal (ELL)
BiCGSafe[1]	BiCRSafe[4]	Jagged Diagonal (JDS)
BiCGSTAB(1)	TFQMR	Block Sparse Row (BSR)
Jacobi	Orthomin(m)	Block Sparse Column (BSC)
Gauss-Seidel	GMRES(m)	Variable Block Row (VBR)
SOR	FGMRES(m)[5]	Dense (DNS)
	IDR(s) [13]	Coordinate (COO)
	additive Schwarz	
	User defined	

2 Installation

This section describes the steps for installing and testing Lis. The explanation is predicated on Lis being installed on the SGI Altix 3700.

2.1 Required Systems

Installation of Lis requires a computing environment in which a C compiler can be used. In addition, the use of the AMG preconditioner routine requires a compiler that supports Fortran 90. In a parallel computing environment, OpenMP or MPI-1 is used. Lis has been verified to run in the environments shown in Table 4.

Table 4: Tested Platforms

C Compiler (Required)	OS
Intel C/C++ Compiler 7.0, 8.0, 9.0, 9.1	Linux
IBM XL C/C++ V7.0	Linux
Sun WorkShop 6 update 2 ONE Studio 7, 11	Solaris 9
GCC 3.3	Linux
FORTRAN Compiler (Option)	
OS	
Intel FORTRAN Compiler 8.1, 9.0, 9.1	Linux
IBM XL FORTRAN V9.1	Linux
Sun WorkShop 6 update 2 ONE Studio 7, 11	Solaris 9
g77 3.3 gfortran 4.1 g95 0.91	Linux

2.2 Decompressing the Files

Enter the following command to decompress the files. (`$VERSION`) represents the version.

```
>gunzip -c lis-($VERSION).tar.gz | tar xvf -
```

This command creates the `lis-($VERSION)` directory along with its subfolders, as shown in Figure 1.

```
lis-($VERSION)
+ config
| Configure files
+ include
| Include files
+ src
| Source files
+ test
  Test programs
```

Figure 1: Files contained in `lis-($VERSION).tar.gz`

2.3 Executing configure script

Enter the following command to execute the script.

- default settings: `>./configure`
- specifying the installation destination: `>./configure --prefix=<install-dir>`

Table 5 shows the option that can be specified for configuration. Table 6 shows the computer environment that can be specified by TARGET.

Table 5: Configuration Options

<code>--enable-omp</code>	Use OpenMP
<code>--enable-mpi</code>	Use MPI
<code>--enable-fortran</code>	Use FORTRAN API
<code>--enable-saamg</code>	Use SA-AMG preconditioner
<code>--enable-quad</code>	Use Quadruple precision operation
<code>--prefix=<install-dir></code>	Name of the installation target directory
<code>TARGET=<target></code>	Computing Environments
<code>CC=<c_compiler></code>	C compiler
<code>CFLAGS=<c_flags></code>	Compilation options for C compilers
<code>FC=<fortran90_compiler></code>	Fortran 90 compiler
<code>FCFLAGS=<fc_flags></code>	Compilation options for the Fortran 90 compiler
<code>LDFLAGS=<ld_flags></code>	Link options

2.4 Executing make

In the `lis-($VERSION)` directory, execute make by entering

```
>make
```

compilation is then performed.

2.5 Testing

Here, ensure that `make` (library compilation) has been successfully executed.

To do so, in the `lis-($VERSION)` directory, enter the following:

```
>make check
```

This command performs a test using the executable files created in the `lis-($VERSION)/test` directory. The test reads matrix and vector data from the MatrixMarket file `lis-($VERSION)/test/testmat.mtx` and writes the approximate solution of the linear equation $Ax = b$ obtained with the BiCG method into `lis-($VERSION)/test/sol.txt`, and the residual history into `lis-($VERSION)/test/res.txt`. If all approximate solutions are 1, then the result is correct. The execution result on SGI Altix 3700 is shown below. The last two digits shown vary depending on the execution environment.

Table 6: TARGET options

<target>	Executing configure script
cray_xt3	./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" --enable-mpi ax_f77_mangling="lower case, no underscore, extra underscore"
nec_sx6i_cross	./configure CC=svc++ FC=svf90 AR=svar RANLIB=true STRIP=svstrip ac_cv_sizeof_void_p=8 ax_vector_machine=yes cross_compiling=yes ax_f77_mangling="lower case, no underscore, extra underscore"
nec_es	./configure CC=esmpic++ FC=esmpif90 AR=esar RANLIB=true ac_cv_sizeof_void_p=8 ax_vector_machine=yes cross_compiling=yes --enable-mpi --enable-omp ax_f77_mangling="lower case, no underscore, extra underscore"
fujitsu_pq	./configure CC="fcc" F77="frt" FC="frt" MPICC="mpifcc" MPIF77="mpifrt" MPIFC="mpifrt" MPIRUN="mpiexec" CFLAGS="-O3" FFLAGS="-O3 -Cpp" FCFLAGS="-O3 -Cpp -Am" FCLDFLAGS="" ac_cv_sizeof_void_p=8 ax_f77_mangling="lower case, underscore, no extra underscore" ax_cv_c_compiler_vendor="fujitsu" ax_cv_f77_compiler_vendor="fujitsu" MPIRUN="mpiexec" MPINP="-n" OMPFLAG="-KOMP"
fujitsu_pg	./configure CC="fcc" F77="frt" FC="frt" MPICC="mpifcc" MPIF77="mpifrt" MPIFC="mpifrt" MPIRUN="mpiexec" CFLAGS="-O3" FFLAGS="-O3 -Cpp" FCFLAGS="-O3 -Cpp -Am" FCLDFLAGS="" ac_cv_sizeof_void_p=8 ax_f77_mangling="lower case, underscore, no extra underscore" ax_cv_c_compiler_vendor="fujitsu" ax_cv_f77_compiler_vendor="fujitsu" cross_compiling="yes" MPIRUN="mpiexec" MPINP="-n" OMPFLAG="-KOMP" AMDEFS="-pg"
ibm_bg	./configure CC=blrts_xlc FC=blrts_xlf90 CFLAGS="-O3 -qarch=440d -qtune=440 -qstrict -I/bgl/BlueLight/ppcfloor/bglsys/include" FFLAGS="-O3 -qarch=440d -qtune=440 -qsuffix=cpp=F -qfixed=72 -w -I/bgl/BlueLight/ppcfloor/bglsys/include" FCFLAGS="-O3 -qarch=440d -qtune=440 -qsuffix=cpp=F90 -w -I/bgl/BlueLight/ppcfloor/bglsys/include" ac_cv_sizeof_void_p=4 cross_compiling="yes" --enable-mpi ax_f77_mangling="lower case, no underscore, no extra underscore"

default

```
100 x 100 matrix 460 entries
Initial vector x = 0
PRECISION : DOUBLE
SOLVER    : BiCG 2
PRECON    : None
CONV_COND : ||r||_2 <= 1.0e-12*||r_0||_2
STORAGE   : CRS
lis_solve is normal end
BiCG: iter      = 15 iter_double = 15 iter_quad = 0
BiCG: times     = 5.178690e-03
BiCG: p_times   = 1.277685e-03 (p_c = 1.254797e-03 p_i = 2.288818e-05 )
BiCG: i_times   = 3.901005e-03
BiCG: Residual  = 6.327297e-15
```

--enable-omp

```
Max Procs   = 32
Max Threads = 2
100 x 100 matrix 460 entries
Initial vector x = 0
PRECISION : DOUBLE
SOLVER    : BiCG 2
PRECON    : None
CONV_COND : ||r||_2 <= 1.0e-12*||r_0||_2
STORAGE   : CRS
lis_solve is normal end
BiCG: iter      = 15 iter_double = 15 iter_quad = 0
BiCG: times     = 8.960009e-03
BiCG: p_times   = 2.297878e-03 (p_c = 2.072096e-03 p_i = 2.257824e-04 )
BiCG: i_times   = 6.662130e-03
BiCG: Residual  = 6.221213e-15
```

--enable-mpi

```
100 x 100 matrix 460 entries
Initial vector x = 0
PRECISION : DOUBLE
SOLVER    : BiCG 2
PRECON    : None
CONV_COND : ||r||_2 <= 1.0e-12*||r_0||_2
STORAGE   : CRS
lis_solve is normal end
BiCG: iter      = 15 iter_double = 15 iter_quad = 0
BiCG: times     = 2.911400e-03
BiCG: p_times   = 1.560780e-04 (p_c = 1.459997e-04 p_i = 1.007831e-05 )
BiCG: i_times   = 2.755322e-03
BiCG: Residual  = 6.221213e-15
```

2.6 Installation

In the `lis-($VERSION)` directory, enter the following:

```
>make install
```

This copies files, as follows:

```
$(INSTALLDIR)
+include
| +lis.h lisf.h
+lib
  +liblis.a
```

lis.h and lisf.h are the header files required when the library is used for C and FORTRAN, respectively. liblis.a is a library file.

2.8 Restrictions

The current version has the following restrictions:

- Restrictions for the preconditioner
 - When a preconditioner other than the Jacobi or SSOR preconditioner is selected and matrix A is not in the CRS format, it is created in the CRS format at the time of preconditioning.
- Restrictions for quadruple precision operation
 - Jacobi, Gauss-Seidel, SOR, and IDR(s) methods are unsupported,
 - Jacobi, Gauss-Seidel and SOR are unsupported in HYBRID preconditioner,
 - I+S and SA-AMG preconditioners are unsupported.
- Restrictions for the matrix storage format
 - In the MPI environment, when the user prepares a necessary array for a target storage format, only CRS is accepted.

3 Basic Operation

This section describes how to use the library. Writing programs to solve the linear equation

$$Ax = b$$

requires the following:

- Initialization
- Matrix creation
- Vector creation
- Solver creation
- Value assignment for matrix and vector
- Iterative method and preconditioner assignment for solver
- Solve
- Finalization

In addition, each program must include the following `include` statement:

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

When Lis is installed in `$(INSTALLDIR)`, `lis.h` and `lisf.h` are located in `$(INSTALLDIR)/include`.

3.1 Initialization and Finalization

The initialization and finalization must be written as follows. The initialization must be executed at the beginning of the program, and the finalization must be executed at the end of the program.

```
C
1: #include "lis.h"
2: int main(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)
```

Initialization

To perform initialization, the following functions are used:

- C `lis_initialize(int* argc, char** argv[])`
- FORTRAN subroutine `lis_initialize(integer ierr)`

This function performs initialization of MPI, and specifies options through the command line.

Finalization

To perform finalization, the following functions are used:

- C `int lis_finalize()`
- FORTRAN subroutine `lis_finalize(integer ierr)`

3.2 Vector

Assume that the order of vector v is `global_n` and that the number of rows of each partial vector when the row block of vector v has been divided with `nprocs` units of the processor is `local_n`. If `global_n` can be divided into an integer answer, then `local_n = global_n / nprocs`. For example, when the row block of vector v is divided with two processors, as shown by Equation (3.1), `global_n` and `local_n` are 4 and 2, respectively.

$$v = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} \begin{matrix} \text{PE0} \\ \text{PE1} \end{matrix} \quad (3.1)$$

In the case of creating vector v in Equation (3.1), Sequential and OpenMP processing create vector v itself in order and MPI creates, at each processor, a partial vector to which the row block was divided with a given number of processors.

Programs to create vector v look like the following, where the number of MPI version processors is assumed to be two.

C(Sequential, OpenMP)

```
1: 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(MPI)

```
1: int      i,n,is,ie;          /*or 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: }
```

FORTTRAN(Sequential, OpenMP)

```
1: 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
```

FORTTRAN(MPI)

```
1: 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
```

Variable declaration

As the second line shows, the declaration is stated as follows:

```
LIS_VECTOR    v;
```

Creating vectors

To create a vector *v*, the following functions are used:

- C `int lis_vector_create(LIS_Comm comm, LIS_VECTOR *vec)`
- FORTRAN subroutine `lis_vector_create(LIS_Comm comm, LIS_VECTOR vec, integer ierr)`

For the example program above, `comm` must be replaced with the MPI communicator. For Sequential and OpenMP processing, the value for `comm` is ignored.

Assigning vector size

To assign the size of a vector *v*, the following functions are used:

- C `int lis_vector_set_size(LIS_VECTOR vec, int local_n, int global_n)`
- FORTRAN subroutine `lis_vector_create(integer local_n, integer global_n, LIS_Comm comm, LIS_VECTOR vec, integer ierr)`

Either `local_n` or `global_n` must be provided. This function can create a vector in one of the following ways: Creating partial vectors of order `local_n` from `local_n`, or creating from `global_n` partial vectors into which the row block of the vector of order `global_n` has been divided with a given number of processors.

In the case of Sequential and OpenMP processing, `local_n = global_n`. This means that both `lis_vector_set_size(v,n,0)` and `lis_vector_set_size(v,0,n)` create a vector of order *n*.

For MPI, `lis_vector_set_size(v,n,0)` creates a partial vector of order n_p at each processor *p*. On the other hand, `lis_vector_set_size(v,0,n)` creates a partial vector of order m_p at each processor *p*. The value for m_p is determined by the library.

Assigning elements

To assign an element to the *i*-th row of vector *v*, the following functions are used:

- C `int lis_vector_set(int flag, int i, LIS_SCALAR value, LIS_VECTOR v)`
- FORTRAN subroutine `lis_vector_set_value(int flag, int i, LIS_SCALAR value, LIS_VECTOR v, integer ierr)`

For MPI, the *i*-th row of the whole vector must be specified, rather than the *i*-th row of the partial vector. Either

LIS_INS_VALUE Assignment `:v[i] = value`, or

LIS_ADD_VALUE Assignment `add:v[i] = v[i] + value`

must be provided for `flag`.

Duplicating vectors

To create a vector that has the same information as for an existing vector, the following functions are used:

- C `int lis_vector_duplicate(LIS_VECTOR vin, LIS_VECTOR *vout)`
- FORTRAN subroutine `lis_vector_duplicate(LIS_VECTOR vin, LIS_VECTOR vout, integer ierr)`

C(MPI)

```
1: 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_CRS);
14: lis_matrix_assemble(A);
```

FORTRAN(Sequential, OpenMP)

```
1: 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_CRS,ierr)
12: call lis_matrix_assemble(A,ierr)
```

FORTRAN(MPI)

```
1: 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_CRS,ierr)
14: call lis_matrix_assemble(A,ierr)
```

Variable declaration

As the second line shows, the declaration is stated as the following:

```
LIS_MATRIX   A;
```

Creating a matrix

To create matrix A, the following functions are used:

- C `int lis_matrix_create(LIS_Comm comm, LIS_MATRIX *A)`
- FORTRAN subroutine `lis_matrix_create(LIS_Comm comm, LIS_MATRIX A, integer ierr)`

`comm` must be replaced with the MPI communicator. For sequential and OpenMP processing, the value for `comm` is ignored.

Assigning a matrix size

To assign a size to matrix `A`, the following functions are used:

- C `int lis_matrix_set_size(LIS_MATRIX A, int local_n, int global_n)`
- FORTRAN subroutine `lis_matrix_set_size(LIS_MATRIX A, integer local_n, integer global_n, integer ierr)`

Either `local_n` or `global_n` must be provided. This function can create matrices in one of the following two ways: Creating partial matrices of order `local_n` x `N` from `local_n`, or creating partial matrices into which the row block of order `global_n` x `global_n` from `global_n` with a given number of processors. `N` represents the total sum of `local_n` of each processor.

In the case of Sequential and OpenMP processing, `local_n = global_n`. This means that both `lis_matrix_set_size(A,n,0)` and `lis_matrix_set_size(A,0,n)` create a matrix of `n` x `n`.

For MPI, `lis_matrix_set_size(A,n,0)` creates at each processor p a partial matrix of $n_p \times N$, where N is the total sum of n_p of each processor. On the other hand, `lis_matrix_set_size(A,0,n)` creates at each processor p a partial matrix of $m_p \times n$, where m_p is the number of the partial matrix, which is determined by the library.

Assigning elements

To assign an element to the cell at the i -th row and j -th column of matrix `A`, the following functions are used:

- C `int lis_matrix_set_value(int flag, int i, int j, LIS_SCALAR value, LIS_MATRIX A)`
- FORTRAN subroutine `lis_matrix_set_value(integer flag, integer i, integer j, LIS_SCALAR value, LIS_MATRIX A, integer ierr)`

For MPI, the i -th row and j -th column of the whole matrix must be specified, rather than the i -th row and j -th column of the partial matrix. Either

LIS_INS_VALUE Assignment `:A(i,j) = value`, or

LIS_ADD_VALUE Assignment `add:A(i,j) = A(i,j) + value`

must be provided for `flag`.

Assigning a storage format

To assign a storage format to matrix `A`, the following functions are used:

- C `int lis_matrix_set_type(LIS_MATRIX A, int matrix_type)`
- FORTRAN subroutine `lis_matrix_set_type(LIS_MATRIX A, int matrix_type, integer ierr)`

`matrix_type` of `A` is `LIS_MATRIX_CRS` when the matrix is created. The following storage formats are accepted.

Storage formats		matrix_type
Compressed Row Storage	(CRS)	{LIS_MATRIX_CRS 1}
Compressed Column Storage	(CCS)	{LIS_MATRIX_CCS 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	(JDS)	{LIS_MATRIX_JDS 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}
Dense	(DNS)	{LIS_MATRIX_DNS 10}
Coordinate	(COO)	{LIS_MATRIX_COO 11}

Assembling a matrix

After assigning elements, the following function must be used:

- C `int lis_matrix_assemble(LIS_MATRIX A)`
- FORTRAN subroutine `lis_matrix_assemble(LIS_MATRIX A, integer ierr)`

`lis_matrix_assemble` is assembled to the storage format specified with `lis_matrix_set_type`.

Removing a matrix

To remove an unwanted matrix, the following functions are used:

- C `int lis_matrix_destroy(LIS_MATRIX A)`
- FORTRAN subroutine `lis_matrix_destroy(LIS_MATRIX A, integer ierr)`

Approach 2: The user prepares the arrays required by the desired storage type

In the case of creating matrix A in Equation (3.2) in the CRS format, Sequential and OpenMP processing creates matrix A itself, and MPI creates at each processor a partial matrix into which the row block has been divided with a given number of processors.

Programs to create matrix A in the CRS format resemble the following, where the number of MPI version processors is assumed to be two:

C(Sequential, OpenMP)

```
1: int          i,k,n,nnz;
2: int          *ptr,*index;
3: LIS_SCALAR   *value;
4: LIS_MATRIX   A;
5: n = 4; nnz = 10; k = 0;
6: lis_matrix_malloc_crs(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_crs(nnz,ptr,index,value,A);
19: lis_matrix_assemble(A);
```

C(MPI)

```
1: int          i,k,n,nnz,is,ie;
2: int          *ptr,*index;
3: LIS_SCALAR   *value;
4: LIS_MATRIX   A;
5: n = 2; nnz = 5; k = 0;
6: lis_matrix_malloc_crs(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_crs(nnz,ptr,index,value,A);
19: lis_matrix_assemble(A);
```

Associating arrays

To associate the arrays required by the CRS format created by the user with matrix A, the following functions are used:

- C `int lis_matrix_set_crs(int nnz, int *row, int *index, LIS_SCALAR *value, LIS_MATRIX A)`
- FORTRAN unsupported

Approach 3: Reading matrix and vector data from a file

Programs to read matrix A in Equation (3.2) in CRS format and vector b in Equation (3.1) from a file resemble the following:

C(Sequential, OpenMP, MPI)

```

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_CRS);
7: lis_input(A,b,x,"matvec.mtx");

```

FORTRAN(Sequential, OpenMP, MPI)

```

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_CRS,ierr)
7: call lis_input(A,b,x,'matvec.mtx',ierr)

```

The content of the target file `matvec.mtx` resembles the following:

```

%%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

```

Reading data from a file

To read the data for matrix A and vectors b and x from a file, the following functions are used:

- C `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, integer ierr)`

`filename` must be replaced with the path to the target file. The following file formats are supported.

- MatrixMarket format (extended to allow vector data to be read)
- Harwell-Boeing format
- LIS format (original format)

3.4 Solution

The program to solve the linear equation $Ax = b$ with a specified iterative method and preconditioner resembles the following:

```
C(Sequential, OpenMP, MPI)
1: LIS_MATRIX A;
2: LIS_VECTOR b,x;
3: LIS_SOLVER solver;
4:
5: /* Create matrix and vector */
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_solver(A,b,x,solver);
```

```
FORTRAN(Sequential, OpenMP, MPI)
1: LIS_MATRIX A
2: LIS_VECTOR b,x
3: LIS_SOLVER solver
4:
5: /* Create matrix and vector */
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_solver(A,b,x,solver,ierr)
```

Creating a SOLVER

To create a SOLVER, the following functions are used:

- C `int lis_solver_create(LIS_SOLVER *solver)`
- FORTRAN subroutine `lis_solver_create(LIS_SOLVER solver, integer ierr)`

Specifying options

To specify options such as an iterative method and preconditioner, the following functions are used:

- C `int lis_solver_set_option(char *text, LIS_SOLVER solver)`
- FORTRAN subroutine `lis_solver_set_option(character text, LIS_SOLVER solver, integer ierr)`

or,

- C `int lis_solver_set_optionC(LIS_SOLVER solver)`
- FORTRAN subroutine `lis_solver_set_optionC(LIS_SOLVER solver, integer ierr)`

ine opt `lis_solver_set_optionC` is a function that sets the option specified in the command line to SOLVER, when the user's program is executed. When `text` is replaced with a desired command line option, the information will be stored in options and parameters.

The table below shows the allowable command lines, where `-i {cgl1}` means `\verb-i cg=` or `-i 1` and `-maxiter [1000]` indicates that `-maxiter` defaults to 1,000.

Specifying an Iterative Method Default: -i bicg

Method	Option	Auxiliary Option	
CG	-i {cg 1}		
BiCG	-i {bicg 2}		
CGS	-i {cgs 3}		
BiCGSTAB	-i {bicgstab 4}		
BiCGSTAB(1)	-i {bicgstab1 5}	-ell [2]	Value for l
GPBiCG	-i {gpbicg 6}		
TFQMR	-i {tfqmr 7}		
Orthomin(m)	-i {orthomin 8}	-restart [40]	Value for Restart m
GMRES(m)	-i {gmres 9}	-restart [40]	Value for Restart m
Jacobi	-i {jacobi 10}		
Gauss-Seidel	-i {gs 11}		
SOR	-i {sor 12}	-omega [1.9]	Value for Relaxation Coefficient 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]	Value for Restart m
IDR(s)	-i {idrs 21}	-restart [40]	Value for Restart s

Specifying a Preconditioner Default: -p none

Preconditioner	Option	Auxiliary Option	
None	-p {none 0}		
Jacobi	-p {jacobi 1}		
ILU(k)	-p {ilu 2}	-ilu_fill [0]	Fill level k
SSOR	-p {ssor 3}	-ssor_w [1.0]	Relaxation Coefficient ω ($0 < \omega < 2$)
Hybrid	-p {hybrid 4}	-hybrid_i {sor}	Iterative method
		-hybrid_maxiter [25]	Maximum number of iterations
		-hybrid_tol [1.0e-3]	Convergence criteria
		-hybrid_w [1.5]	Relaxation Coefficient ω for the SOR method ($0 < \omega < 2$)
		-hybrid_ell [2]	Value for l of the BiCGSTAB(1) method
		-hybrid_restart [40]	Restart values for GMRES and Orthomin
I+S	-p {is 5}	-is_alpha [1.0]	Parameter α for preconditioner of a $I + \alpha S^{(m)}$ type
		-is_m [3]	Parameter m for preconditioner of a $I + \alpha S^{(m)}$ type
SAINV	-p {sainv 6}	-sainv_drop [0.05]	Drop criteria
SA-AMG	-p {saamg 7}	-saamg_unsym [false]	Selection of asymmetric version
Crout ILU	-p {iluc 8}	-iluc_drop [0.05]	Drop criteria
		-iluc_rate [5.0]	Ratio of Maximum fill-in
ILUT	-p {ilut 9}	-ilut_drop [0.05]	Drop criteria
		-ilut_rate [5.0]	Ratio of Maximum fill-in
additive Schwarz	-adds true	-adds_iter [1]	Number of iterations

Other Options

Option	
<code>-maxiter [1000]</code>	Maximum number of iterations
<code>-tol [1.0e-12]</code>	Convergence criteria
<code>-print [0]</code>	Display of the residual
<code>-print {none 0}</code>	None
<code>-print {mem 1}</code>	Saves the residual history in memory
<code>-print {out 2}</code>	Displays the residual history
<code>-print {all 3}</code>	Saves the residual history and displays it on the screen
<code>-scale [0]</code>	Selection of scaling. The result will overwrite the original matrix and vectors
<code>-scale {none 0}</code>	No scaling
<code>-scale {jacobi 1}</code>	Jacobi scaling $D^{-1}Ax = D^{-1}b$ D represents the diagonal of $A = (a_{ij})$
<code>-scale {symm_diag 2}</code>	Diagonal scaling $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$ $D^{-1/2}$ represents a diagonal matrix that has $1/\sqrt{a_{ii}}$ as a diagonal element
<code>-initx_zeros [true]</code>	Behavior of initial vector x_0
<code>-initx_zeros {false 0}</code>	Given values
<code>-initx_zeros {true 1}</code>	All elements are set to 0.
<code>-omp_num_threads [t]</code>	Number of execution threads t represents the maximum number of threads

Precision Default: `-precision double`

Precision	Option	Auxiliary Option
DOUBLE	<code>-precision {double 0}</code>	
QUAD	<code>-precision {quad 1}</code>	

Solution

To solve the linear equation $Ax = b$, the following functions are used:

- C `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, integer ierr)`

3.5 Sample Program

This is a program for solving the linear equation $Ax = b$ with a specified iterative method and preconditioner to provide an approximate solution for the equation, where matrix A is a tridiagonal matrix of 12×12 .

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

Vector b on the right-hand side of the equation is solved such that approximate solution x will be always 1.

This sample program is stored in the `lis-($VERSION)/test` directory.

Test program: test3.c

```
1: #include <stdio.h>
2: #include "lis.h"
3: main(int argc, char *argv[])
4: {
5:     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(MPI_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_CRS);
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_iters(solver,&iter);
34:    printf("iter = %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: }
}
```

Test program: test3f.f

```
1:      implicit none
2:
3: #include "lisf.h"
4:
5:      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_CRD,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_iters(solver,iter,ierr)
35:     write(*,*) 'iter = ',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
```

3.6 Compiling and Linking

Provided below are examples for cases in which the test programs `test3.c` located in the `lis-($VERSION)/test` directory are compiled on the SGI Altix 3700 using the Intel C/C++ Compiler 8.1 (`icc`) and Intel Fortran Compiler 8.1 (`ifort`). Since the library includes the Fortran 90 code if the SA-AMG preconditioner is used, the link must be processed with the Fortran 90 compiler.

Sequential

Compile

```
>icc -c -I$(INSTALLDIR)/include test3.c
```

Link

```
>icc -o test3 test3.o -llis
```

Link(Use SA-AMG)

```
>ifort -nofor_main -o test3 test3.o -llis
```

OpenMP

Compile

```
>icc -c -openmp -I$(INSTALLDIR)/include test3.c
```

Link

```
>icc -openmp -o test3 test3.o -llis
```

Link(Use SA-AMG)

```
>ifort -nofor_main -openmp -o test3 test3.o -llis
```

MPI

Compile

```
>icc -c -DUSE_MPI -I$(INSTALLDIR)/include test3.c
```

Link

```
>icc -o test3 test3.o -llis -lmpi
```

Link(Use SA-AMG)

```
>ifort -nofor_main -o test3 test3.o -llis -lmpi
```

OpenMP + MPI

Compile

```
>icc -c -openmp -DUSE_MPI -I$(INSTALLDIR)/include test3.c
```

Link

```
>icc -openmp -o test3 test3.o -llis -lmpi
```

Link(Use SA-AMG)

```
>ifort -nofor_main -openmp -o test3 test3.o -llis -lmpi
```

Provided below are examples for cases in which the test programs `test3f.f` located in the `lis-($VERSION)/test` directory are compiled on the SGI Altix 3700 using the Intel Fortran Compiler 8.1 (ifort). Since the `#include` statement is used for the user's FORTRAN program, the compiler option must be specified to use the pre-processor. The option for ifort is `-fpp`.

Sequential

Compile

```
>ifort -c -fpp -I$(INSTALLDIR)/include test3f.f
```

Link

```
>ifort -o test3 test3.o -llis
```

OpenMP

Compile

```
>ifort -c -fpp -openmp -I$(INSTALLDIR)/include test3f.f
```

Link

```
>ifort -openmp -o test3 test3.o -llis
```

MPI

Compile

```
>ifort -c -fpp -DUSE_MPI -I$(INSTALLDIR)/include test3f.f
```

Link

```
>ifort -o test3 test3.o -llis -lmpi
```

OpenMP + MPI

Compile

```
>ifort -c -fpp -openmp -DUSE_MPI -I$(INSTALLDIR)/include test3f.f
```

Link

```
>ifort -openmp -o test3 test3.o -llis -lmpi
```

3.7 Execution

Execute on the SGI Altix 3700 the test programs `test3.c` and `test3f.f` in the `lis-($VERSION)/test` directory by entering the following:

sequential

```
>./test3 -i bicgstab
```

OpenMP

```
>env OMP_NUM_THREADS=2 ./test3 -i bicgstab
```

MPI

```
>mpirun -np 2 ./test3 -i bicgstab
```

OpenMP + MPI

```
>mpirun -np 2 env OMP_NUM_THREADS=2 ./test3 -i bicgstab
```

Then, the following results will be returned:

```
SOLVER : BiCGSTAB
```

```
PRECON : None
```

```
lis_solve is normal end
```

```
iter = 6
```

```
0 1.000000e+000
1 1.000000e+000
2 1.000000e+000
3 1.000000e+000
4 1.000000e+000
5 1.000000e+000
6 1.000000e+000
7 1.000000e+000
8 1.000000e+000
9 1.000000e+000
10 1.000000e+000
11 1.000000e+000
```

4 Quadruple precision operation

The double precision operation sometimes requires a large of number of iterations for convergence because of the rounding error. The high-precision operation is effective for the improvement of convergence[14]. In this library, we implement the quadruple precision operations, which have "double-double" precision[15, 16] by combining two double precision floating point numbers. To use quadruple precision by the same interface as double precision, coefficient matrix A , solution x , and b of the right-hand side are assumed to be double precision. Acceleration was achieved by using the SSE2 SIMD instruction[21].

4.1 Executing quadruple precision operation

Execute on a Linux PC (CPU:Xeon) the test programs `test4.c` in the `lis-($VERSION)/test` directory by entering the following:

Sequential and double precision

```
>./test4 200 2.0 -precision double
```

Then, the following results will be returned:

```
n=200 gamma=2.000000
Initial vector x = 0
PRECISION : DOUBLE
SOLVER    : BiCG 2
PRECON    : None
CONV_COND : ||r||_2 <= 1.0e-12*||r_0||_2
STORAGE   : CRS
lis_solve is LIS_MAXITER(code=4)
BiCG: iter      = 1001 iter_double = 1001 iter_quad = 0
BiCG: times     = 2.044368e-02
BiCG: p_times  = 4.768372e-06 (p_c = 4.768372e-06 p_i = 0.000000e+00 )
BiCG: i_times  = 2.043891e-02
BiCG: Residual = 8.917591e+01
```

Sequential and quadruple precision

```
>./test4 200 2.0 -precision quad
```

Then, the following results will be returned:

```
n=200 gamma=2.000000
Initial vector x = 0
PRECISION : QUAD
SOLVER    : BiCG 2
PRECON    : None
CONV_COND : ||r||_2 <= 1.0e-12*||r_0||_2
STORAGE   : CRS
lis_solve is normal end
BiCG: iter      = 230 iter_double = 0 iter_quad = 230
BiCG: times     = 2.267408e-02
BiCG: p_times  = 4.549026e-04 (p_c = 5.006790e-06 p_i = 4.498959e-04 )
BiCG: i_times  = 2.221918e-02
BiCG: Residual = 6.499145e-11
```

5 Matrix Storage Format

This section describes the matrix storage types that can be used for the library. Assume that the matrix row (column) number begins with 0 and that the number of non-zero elements of matrix A of $n \times n = (a_{ij})$ is nnz .

5.1 Compressed Row Storage (CRS)

The CRS format uses three arrays (`ptr`, `index`, `value`) to store data.

- The `value` array, a double-precision array with a length of nnz , stores non-zero elements of matrix A along the row.
- The `index` array, an integer array with a length of nnz , stores the column numbers of the non-zero elements stored in the `value` array.
- The `ptr` array, an integer array with a length of $n + 1$, stores the starting points of the rows of the `value` and `index` arrays.

5.1.1 How to Create a Matrix (Sequential and OpenMP)

The right-hand diagram in Figure 2 shows how matrix A in Figure 2 is stored in the CRS format. The program to create this matrix in the CRS format is as follows:

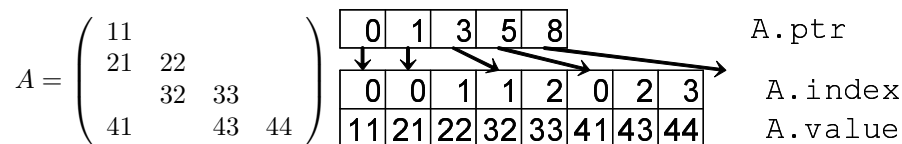


Figure 2: Data structures of CRS.

Sequential and OpenMP

```

1: int      n,nnz;
2: int      *ptr,*index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnz = 8;
6: ptr = (int *)malloc( (n+1)*sizeof(int) );
7: index = (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_crs(nnz,ptr,index,value,A);
19: lis_matrix_assemble(A);

```

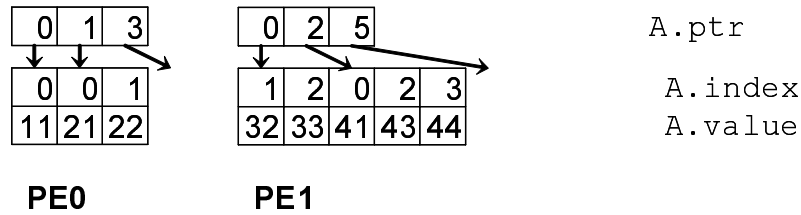


Figure 3: Data structures of CRS.

5.1.2 How to Create a Matrix (MPI)

Figure 3 shows how matrix A in Figure 2 is stored in the CRS format with two processors. The program to create this matrix in the CRS format with two processors is as follows:

```

MPI
1: int      i,k,n,nnz,my_rank;
2: 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      = (int *)malloc( (n+1)*sizeof(int) );
9: index    = (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_crs(nnz,ptr,index,value,A);
22: lis_matrix_assemble(A);

```

5.1.3 Pertinent Function

Associating arrays

To associate the arrays required by the CRS format with matrix A , the following function is used:

- `int lis_matrix_set_crs(int nnz, int *row, int *index, LIS_SCALAR *value, LIS_MATRIX A)`

5.2 Compressed Column Storage (CCS)

The CSS format uses three arrays (`ptr`, `index`, `value`) to store data.

- The `value` array, a double-precision array with a length of `nnz`, stores the values for the non-zero elements of matrix A along the column.
- The `index` array, an integer array with a length of `nnz`, stores the row numbers of the non-zero elements stored in the `value` array.
- The `ptr` array, an integer array with a length of $n + 1$, stores the starting points of the rows of the `value` and `index` arrays.

5.2.1 How to Create a Matrix (Sequential and OpenMP)

The right-hand diagram in Figure 4 shows how matrix A in Figure 4 is stored in the CCS format. The program to create this matrix in the CCS format is as follows:

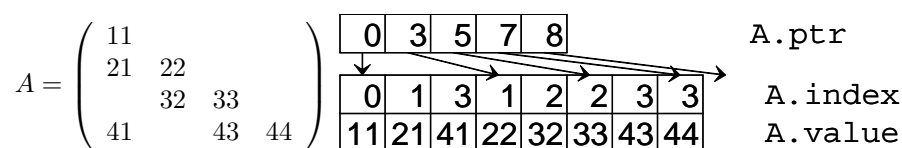


Figure 4: Data structures of CCS.

Sequential and OpenMP

```

1: int      n,nnz;
2: int      *ptr,*index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnz = 8;
6: ptr = (int *)malloc( (n+1)*sizeof(int) );
7: index = (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_ccs(nnz,ptr,index,value,A);
19: lis_matrix_assemble(A);

```

5.2.2 How to Create a Matrix (MPI)

Figure 5 shows how matrix A in Figure 4 is stored with two processors. The program to create this matrix in the CCS format with two processors is as follows:

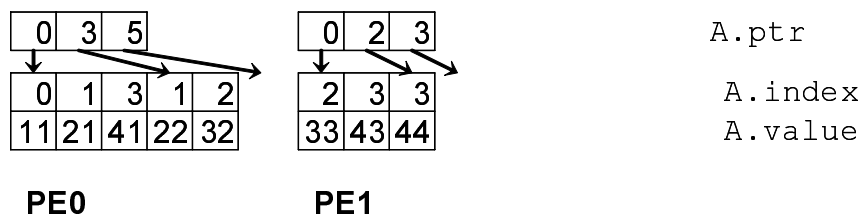


Figure 5: Data structures of CCS.

MPI

```

1: int      i,k,n,nnz,my_rank;
2: 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 = (int *)malloc( (n+1)*sizeof(int) );
9: index = (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_ccs(nnz,ptr,index,value,A);
22:     lis_matrix_assemble(A);

```

5.2.3 Pertinent Function

Associating arrays

To associate the arrays required by the CCS format with matrix A , the following function is used:

- `int lis_matrix_set_ccs(int nnz, int *row, int *index, LIS_SCALAR *value, LIS_MATRIX A)`

5.3 Modified Compressed Sparse Row (MSR)

The MSR format is a modified version of the CRS format. The MSR format is different in that it divides the diagonal before storing it. The MSR format uses two arrays (`index,value`) to store data. Assume that `ndz` represents the number of zero elements of the diagonal.

- The `value` array, a double-precision array with a length of $nnz + ndz + 1$, stores the diagonal of matrix A down to the n -th element. The $n + 1$ -th element is not used. For the $n + 2$ -th and later elements, the values of non-zero elements except the diagonal of matrix A are stored along the row.
- The `index` array, an integer array with a length of $nnz + ndz + 1$, stores the starting points of the rows of the non-diagonals of matrix A down to the $n + 1$ -th element. For the $n + 2$ -th and later elements, it stores the row numbers of the non-diagonals of matrix A stored in the `value` array.

5.3.1 How to Create a Matrix (Sequential and OpenMP)

The right-hand diagram in Figure 6 shows how matrix A is stored in the MSR format. The program to create this matrix in the MSR format is as follows:

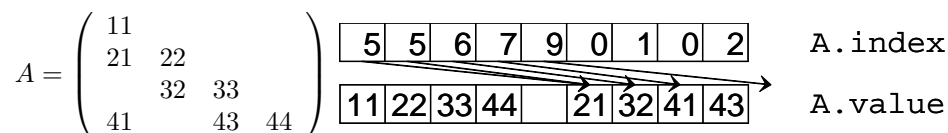


Figure 6: Data structures of MSR.

Sequential and OpenMP

```

1: int      n,nnz,ndz;
2: int      *index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnz = 8; ndz = 0;
6: index = (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 How to Create a Matrix (MPI)

Figure 7 shows how matrix A in Figure 6 is stored in the MSR format with two processors. The program to create this matrix in the MSR format with two processors is as follows:

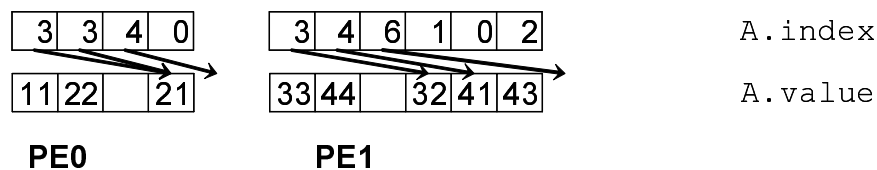


Figure 7: Data structures of MSR.

```

MPI
1: int      i,k,n,nnz,ndz,my_rank;
2: 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 = (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 Pertinent Function

Associating arrays

To associate the arrays required by the MSR format with matrix A , the following function is used:

- `int lis_matrix_set_msr(int nnz, int ndz, int *index, LIS_SCALAR *value, LIS_MATRIX A)`

5.4 Diagonal (DIA)

DIA uses two arrays (`index`, `value`) to store data. Assume that nnd represents the number of non-zero diagonals of matrix A .

- The `value` array, a double-precision array with a length of $nnd \times n$, stores non-zero diagonals of matrix A .
- The `index` array, an integer array with a length of nnd , stores the offsets from the main diagonal to the other diagonals.

For OpenMP, the following modifications have been made: DIA uses two arrays (`index`, `value`) to store data. Assume that $nprocs$ represents the number of threads. nnd_p is the number of non-zero diagonals of the partial matrix into which the row block of matrix A has been divided. $maxnnd$ is the maximum value nnd_p .

- The `value` array, a double-precision array with a length of $maxnnd \times n$, stores non-zero diagonals of matrix A .
- The `index` array, an integer array with a length of $nprocs \times maxnnd$, stores the offsets from the main diagonal to the other diagonals.

5.4.1 How to Create a Matrix (Sequential)

The right-hand diagram in Figure 8 shows how matrix A in Figure 8 is stored in DIA format. The program to create this matrix in the DIA format is as follows:

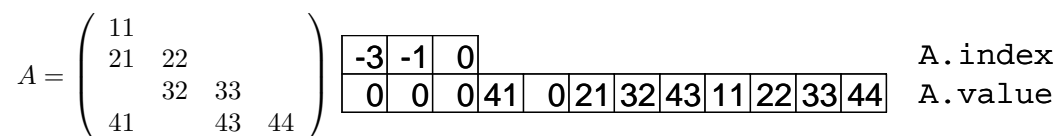


Figure 8: Data structures of DIA.

Sequential

```

1: int      n,nnd;
2: int      *index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnd = 3;
6: index = (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);

```


5.5.2 How to Create a Matrix (MPI)

Figure 12 shows how matrix A in Figure 11 is stored in the ELL format. The program to create this matrix in the ELL format with two processors is as follows:

0	0	0	1	1	0	2	2	2	3	A.index
11	21	0	22	32	41	33	43	0	44	A.value
PE0				PE1						

Figure 12: Data structures of ELL.

MPI

```

1: int          i,n,maxnzs,my_rank;
2: 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; maxnzs = 2;}
7: else          {n = 2; maxnzs = 3;}
8: index = (int *)malloc( n*maxnzs*sizeof(int) );
9: value = (LIS_SCALAR *)malloc( n*maxnzs*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(maxnzs,index,value,A);
21: lis_matrix_assemble(A);

```

5.5.3 Pertinent Function

Associating arrays

To associate an array required for the ELL format with matrix A , the following function is used:

- `int lis_matrix_set_ell(int maxnzs, int *index, LIS_SCALAR *value, LIS_MATRIX A)`

5.6 Jagged Diagonal (JDS)

JDS first sorts the non-zero elements of the rows in decreasing order of size, and then stores them along the column. JDS uses four arrays (`perm`, `ptr`, `index`, `value`) to store the elements. Assume that $maxnzs$ represents the maximum value for the number of non-zero elements of matrix A .

- The `perm` array, an integer array with a length of n , stores the sorted row numbers.
- The `value` array, a double-precision array with a length of nnz , stores jagged diagonals of sorted matrix A . The first jagged diagonals consist of the first non-zero elements of the rows. The next jagged diagonals, as well as later jagged diagonals, consist of the second non-zero elements.
- The `index` array, an integer array with a length of nnz , stores the row numbers of the non-zero elements stored in the `value` array.
- The `ptr` array, an integer array with a length of $maxnzs + 1$, stores the starting points of the jagged diagonals.

For OpenMP, the following modifications have been made: JDS uses four arrays (`perm`, `ptr`, `index`, `value`) to store data. Assume that $nprocs$ is the number of threads. $maxnzs_p$ is the number of non-zero diagonals of the partial matrix into which the row block of matrix A has been divided. $maxmaxnzs$ is the maximum value of $maxnzs_p$.

- The `perm` array, an integer array with a length of n , stores the sorted row numbers.
- The `value` array, a double-precision array with a length of nnz , stores jagged diagonals of sorted matrix A . The first jagged diagonals consist of the first non-zero elements of the rows. The next jagged diagonals, as well as the later diagonals, consist of the second non-zero elements.
- The `index` array, an integer array with a length of nnz , stores the row numbers of the non-zero elements stored in the `value` array.
- The `ptr` array, an integer array with a length of $nprocs \times (maxmaxnzs + 1)$, stores the starting points of the jagged diagonals.

5.6.1 How to Create a Matrix (Sequential)

The right-hand diagram in Figure 13 shows how matrix A in Figure 13 is stored in the JDS format. The program to create this matrix in the JDS format is as follows:

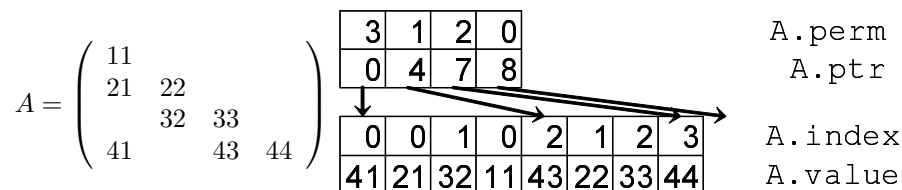


Figure 13: Data structures of JDS.

Sequential

```

1: int      n, nnz, maxnzs;
2: int      *perm, *ptr, *index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnz = 8; maxnzs = 3;
6: perm = (int *)malloc( n*sizeof(int) );
7: ptr = (int *)malloc( (maxnzs+1)*sizeof(int) );
8: index = (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_jds(nnz, maxnzs, perm, ptr, index, value, A);
21: lis_matrix_assemble(A);

```

5.6.2 How to Create a Matrix (OpenMP)

Figure 14 shows how matrix A in Figure 13 is stored in the JDS format with two threads. The program to create this matrix in the JDS format with two threads is as follows:

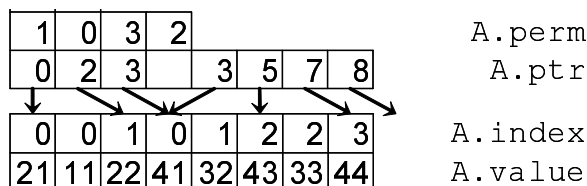


Figure 14: Data structures of JDS.

OpenMP

```

1: int      n, nnz, maxmaxnzs, nprocs;
2: int      *perm, *ptr, *index;
3: LIS_SCALAR *value;
4: LIS_MATRIX A;
5: n = 4; nnz = 8; maxmaxnzs = 3; nprocs = 2;
6: perm = (int *)malloc( n*sizeof(int) );
7: ptr = (int *)malloc( nprocs*(maxmaxnzs+1)*sizeof(int) );
8: index = (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_jds(nnz, maxmaxnzs, perm, ptr, index, value, A);
22: lis_matrix_assemble(A);

```

5.6.3 How to Create a Matrix (MPI)

Figure 15 shows how matrix A in Figure 13 is stored in the JDS format with two processors. The program to create this matrix in the JDS format with two processors is as follows:

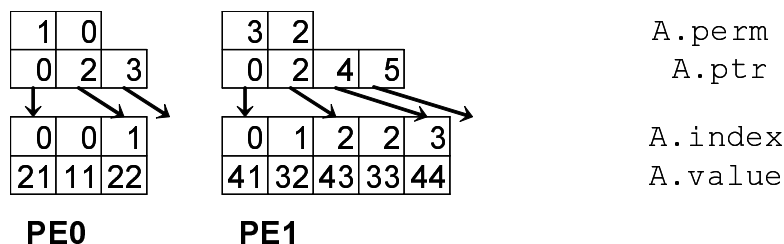


Figure 15: Data structures of JDS.

MPI

```

1: int      i,n,nnz,maxnzs,my_rank;
2: 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; maxnzs = 2;}
7: else      {n = 2; nnz = 5; maxnzs = 3;}
8: perm = (int *)malloc( n*sizeof(int) );
9: ptr = (int *)malloc( (maxnzs+1)*sizeof(int) );
10: index = (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_jds(nnz,maxnzs,perm,ptr,index,value,A);
25: lis_matrix_assemble(A);

```

5.6.4 Pertinent Function

Associating arrays

To associate an array required for the JDS format with matrix A , the following function is used:

- `int lis_matrix_set_jds(int nnz, int maxnzs, int *perm, int *ptr, int *index, LIS_SCALAR *value, LIS_MATRIX A)`

5.7 Block Sparse Row (BSR)

BSR breaks down each matrix into partial matrixes (called blocks) with a size of $r \times c$. BSR stores non-zero blocks (blocks in which at least one non-zero element exists) with the same step as that for the CRS format. Assume that $nr = n/r$ and $nnzb$ are the numbers of non-zero blocks of A . BSR uses three arrays (`bptr`, `bindex`, `value`) to store matrices.

- The `value` array, a double-precision array with a length of $nnzb \times r \times c$, stores all elements of the non-zero blocks.
- The `bindex` array, an integer array with a length of $nnzb$, stores block column numbers of the non-zero blocks.
- The `bptr` array, an integer array with a length of $nr + 1$, stores the starting points of the block rows in the `bindex` array.

5.7.1 How to Create a Matrix (Sequential and OpenMP)

The right-hand diagram in Figure 16 shows how matrix A in Figure 16 is stored in the BSR format. The program to create this matrix in the BSR format is as follows:

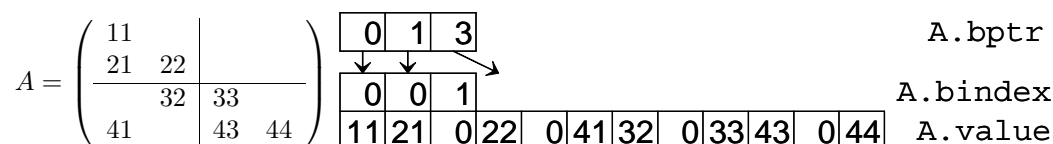


Figure 16: Data structures of BSR.

Sequential and OpenMP

```

1: int          n, bnr, bnc, nr, nc, bnnz;
2: 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  = (int *)malloc( (nr+1)*sizeof(int) );
7: bindex = (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 How to Create a Matrix (MPI)

Figure 17 shows how matrix A in Figure 16 is stored in the BSR format with two processors. The program to create this matrix in the BSR format with two processors is as follows:

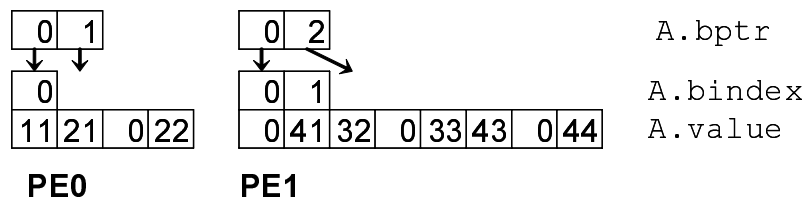


Figure 17: Data structures of BSR.

MPI

```

1: int      n, bnr, bnc, nr, nc, bnnz, my_rank;
2: 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 = (int *)malloc( (nr+1)*sizeof(int) );
9: bindex = (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 Pertinent Function

Associating arrays

To associate the arrays required by the BSR format with matrix A , the following function is used:

- `int lis_matrix_set_bsr(int bnr, int bnc, int bnnz, int *bptr, int *bindex, LIS_SCALAR *value, LIS_MATRIX A)`

5.8 Block Sparse Column (BSC)

BSC breaks down each matrix into partial matrixes (called blocks) with a size of $r \times c$. BSC stores non-zero blocks (blocks in which at least one non-zero block exists) in the same step as that for the CCS format. Assume that $nc = n/c$ and $nnzb$ are the numbers of non-zero blocks of A. BSC uses three arrays (`bptr`, `bindex`, `value`) to store matrices.

- The `value` array, a double-precision array with a length of $nnzb \times r \times c$, stores all elements of the non-zero blocks.
- The `bindex` array, an integer array with a length of $nnzb$, stores block row numbers of the non-zero blocks.
- The `bptr` array, an integer array with a length of $nc + 1$, stores the starting points of the block columns in the `bindex` array.

5.8.1 How to Create a Matrix (Sequential and OpenMP)

The right-hand diagram in Figure 18 shows how matrix A in Figure 18 is stored in the BSC format. The program to create this matrix in the BSC format is as follows:

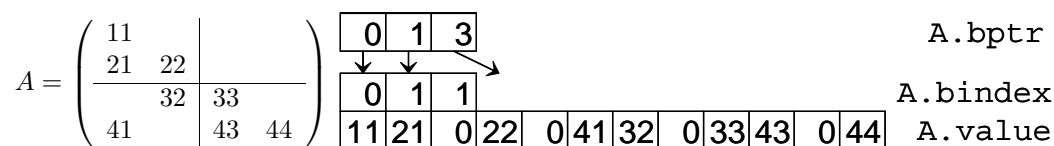


Figure 18: Data structures of BSC.

Sequential and OpenMP

```

1: int          n, bnr, bnc, nr, nc, bnnz;
2: 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  = (int *)malloc( (nc+1)*sizeof(int) );
7: bindex = (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 How to Create a Matrix (MPI)

Figure 19 shows how matrix A in Figure 18 is stored in the BSC format with two processors. The program to create this matrix in the BSC format with two processors is as follows:

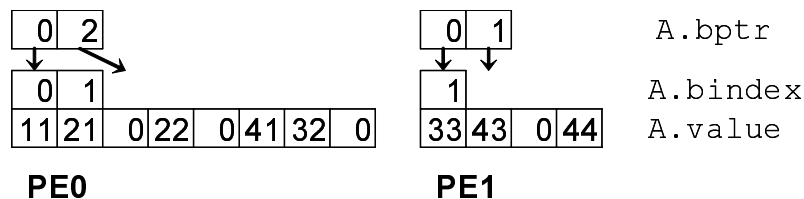


Figure 19: Data structures of BSC.

MPI

```

1: int      n, bnr, bnc, nr, nc, bnnz, my_rank;
2: 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 = (int *)malloc( (nr+1)*sizeof(int) );
9: bindex = (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 Pertinent Function

Associating arrays

To associate the arrays required by the BSC format with matrix A , the following function is used:

- `int lis_matrix_set_bsc(int bnr, int bnc, int bnnz, int *bptr, int *bindex, LIS_SCALAR *value, LIS_MATRIX A)`

5.9 Variable Block Row (VBR)

VBR is a generalized version of BSR. The division points of the rows and columns are given by the arrays (`row` and `col`). VBR stores non-zero blocks (blocks in which at least one non-zero block exists) in the same step as that for the CRS format. Assume that nr and nc are the numbers of row and column divisions, respectively, and that $nnzb$ denotes the number of non-zero blocks of A , and nnz denotes the total number of elements of the non-zero blocks. VBR uses six arrays (`bptr`, `bindex`, `row`, `col`, `ptr` and `value`) to store matrices.

- The `row` array, an integer array with a length of $nr + 1$, stores the starting row number of the block rows.
- The `col` array, an integer array with a length of $nc + 1$, stores the starting column number of the block columns.
- The `bindex` array, an integer array with a length of $nnzb$, stores the block column numbers of the non-zero blocks.
- The `bptr` array, an integer array with a length of $nr + 1$, stores the starting point of the block rows in the `bindex` array.
- The `value` array, a double-precision array with a length of nnz , stores all elements of the non-zero blocks.
- The `ptr` array, an integer array with a length of $nnzb + 1$, stores the starting points of the non-zero blocks in the `value` array.

5.9.1 How to Create a Matrix (Sequential and OpenMP)

The right-hand diagram in Figure 20 shows how matrix A in Figure 20 is stored in the VBR format. The program to create this matrix in the VBR format is as follows:

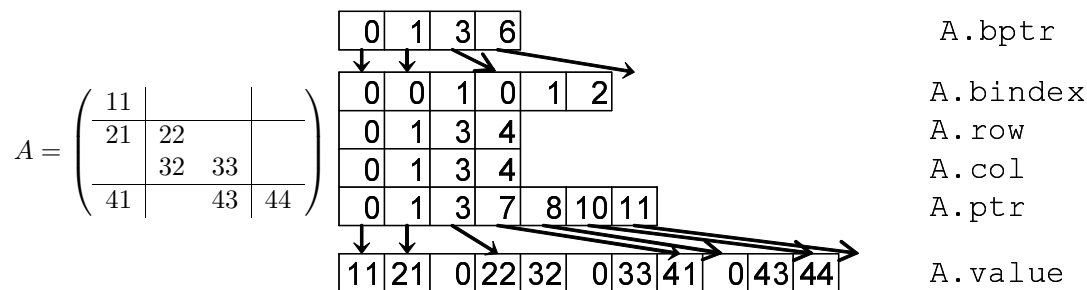


Figure 20: Data structures of VBR.

```

1: int          n,nnz,nr,nc,bnnz;
2: 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  = (int *)malloc( (nr+1)*sizeof(int) );
7: row   = (int *)malloc( (nr+1)*sizeof(int) );
8: col   = (int *)malloc( (nc+1)*sizeof(int) );
9: ptr   = (int *)malloc( (bnnz+1)*sizeof(int) );
10: bindex = (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 How to Create a Matrix (MPI)

Figure 21 shows how matrix A in Figure 20 is stored in the VBR format with two processors. The program to create this matrix in the VBR format with two processors is as follows:

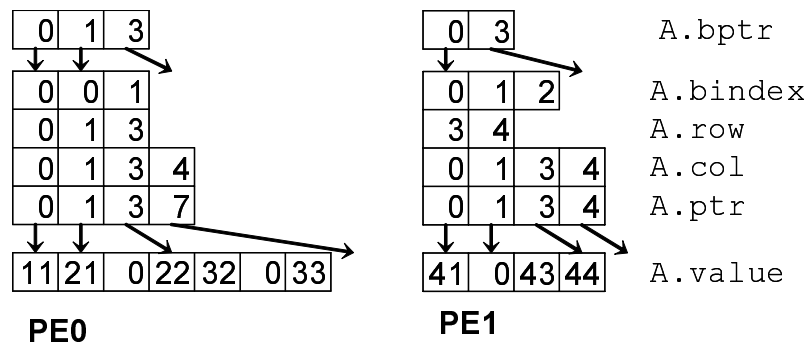


Figure 21: Data structures of VBR.

MPI

```
1: int          n,nnz,nr,nc,bnnz,my_rank;
2: 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        = (int *)malloc( (nr+1)*sizeof(int) );
9: row         = (int *)malloc( (nr+1)*sizeof(int) );
10: col        = (int *)malloc( (nc+1)*sizeof(int) );
11: ptr        = (int *)malloc( (bnnz+1)*sizeof(int) );
12: bindex     = (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 Pertinent Function

Associating arrays

To associate the arrays required by the VBR format with matrix A, the following function is used:

- `int lis_matrix_set_vbr(int nnz, int nr, int nc, int bnnz, int *row, int *col, int *ptr, int *bptr, int *bindex, LIS_SCALAR *value, LIS_MATRIX A)`

5.10.2 How to Create a Matrix (MPI)

Figure 23 shows how matrix A in Figure 22 is stored in the COO format with two processors. The program to create this matrix in the COO format with two processors is as follows:

0	1	1	3	2	2	3	3	A.row
0	0	1	0	1	2	2	3	A.col
11	21	22	41	32	33	43	44	A.value
PE0			PE1					

Figure 23: Data structures of COO.

MPI

```

1: int          n, nnz, my_rank;
2: 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  = (int *)malloc( nnz*sizeof(int) );
9: col  = (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 Pertinent Function

Associating arrays

To associate the arrays required by the COO format with matrix A , the following function is used:

- `int lis_matrix_set_coo(int nnz, int *row, int *col, LIS_SCALAR *value, LIS_MATRIX A)`

5.11.2 How to Create a Matrix (MPI)

Figure 25 shows how matrix A in Figure 24 is stored in the DNS format with two processors. The program to create this matrix in the DNS format with two processors is as follows:

11	21	0	22	0	41	32	0	A.Value
0	0	0	0	33	43	0	44	
PE0				PE1				

Figure 25: Data structures of DNS.

MPI

```
1: 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 Pertinent Function

Associating arrays

To associate the arrays required by the DNS format with matrix A , the following function is used:

- `int lis_matrix_set_dns(LIS_SCALAR *value, LIS_MATRIX A)`

6 Functions

This section describes the functions that can be employed by users. The return value of the function in C and the value of `ierr` in FORTRAN are as follows:

Return value

<code>LIS_SUCCESS(0)</code>	Normal termination
<code>LIS_ILL_OPTION(1)</code>	Illegal option
<code>LIS_BREAKDOWN(2)</code>	Breakdown
<code>LIS_OUT_OF_MEMORY(3)</code>	Insufficient working memory
<code>LIS_MAXITER(4)</code>	Did not converge within the maximum number of iterations
<code>LIS_NOT_IMPLEMENTED(5)</code>	Not implemented
<code>LIS_ERR_FILE_IO(6)</code>	File I/O error

6.1 Vector Operations

Assume that the order of vector v is `global_n` and that the row number of each block into which the row block of vector v has been divided with `nprocs` units of processors is `local_n`. `global_n` and `local_n` are called the global order and the local order, respectively.

6.1.1 `lis_vector_create`

```
C      int lis_vector_create(LIS_Comm comm, LIS_VECTOR *vec)
FORTRAN subroutine lis_vector_create(LIS_Comm comm, LIS_VECTOR vec, integer ierr)
```

Capability

Creating vector v

Input

`LIS_Comm` MPI communicator

Output

`vec` Vector

`ierr` Return code

Note

For sequential and OpenMP processing, the value for `comm` is ignored.

6.1.2 lis_vector_destroy

```
C      int lis_vector_destroy(LIS_VECTOR vec)
FORTRAN subroutine lis_vector_destroy(LIS_VECTOR vec, integer ierr)
```

Capability

Removing unwanted vectors from memory

Input

vec Vector to be removed from memory

Output

ierr Return code

6.1.3 lis_vector_duplicate

```
C      int lis_vector_duplicate(void *vin, LIS_VECTOR *vout)
FORTRAN subroutine lis_vector_duplicate(LIS_VECTOR vin, LIS_VECTOR vout,
integer ierr)
```

Capability

Creating a vector that has the same information as an existing vector.

Input

vin Source vector

Output

vout Destination vector

ierr Return code

Note

The `lis_vector_duplicate` function does not copy the values, but only reserves an area. To copy the values as well, the function `lis_vector_copy` must be used after this function.

6.1.4 `lis_vector_set_size`

```
C      int lis_vector_set_size(LIS_VECTOR vec, int local_n, int global_n)
FORTRAN subroutine lis_vector_set_size(LIS_VECTOR vec, integer local_n,
      integer global_n, integer ierr)
```

Capability

Assigning vector size

Input

<code>vec</code>	Vector
<code>local_n</code>	Local order of the vector
<code>global_n</code>	Global order of the vector

Output

<code>ierr</code>	Return code
-------------------	-------------

Note

Either `local_n` or `global_n` must be provided. This function can create a vector in one of the following ways: Creating partial vectors of order `local_n` from `local_n`, or creating from `global_n` partial vectors into which the row block of the vector of order `global_n` has been divided from a given number of processors.

In the case of Sequential and OpenMP processing, `local_n = global_n`. This means that both `lis_vector_set_size(v,n,0)` and `lis_vector_set_size(v,0,n)` create a vector of order n .

6.1.5 `lis_vector_get_size`

```
C      int lis_vector_get_size(LIS_VECTOR v, int *local_n, int *global_n)
FORTRAN subroutine lis_vector_get_size(LIS_VECTOR v, integer local_n,
      integer global_n, integer ierr)
```

Capability

Ascertaining the size of vector v

Input

<code>v</code>	Vector
----------------	--------

Output

<code>local_n</code>	Local order of the vector
<code>global_n</code>	Global order of the vector
<code>ierr</code>	Return code

Note

In the case of Sequential and OpenMP processing, `local_n = global_n`.

6.1.6 lis_vector_get_range

```
C      int lis_vector_get_range(LIS_VECTOR v, int *is, int *ie)
FORTRAN subroutine lis_vector_get_range(LIS_VECTOR v, integer is, integer ie,
      integer ierr)
```

Capability

Ascertaining where in the whole vector the partial vector v is located

Input

v Partial vector

Output

is Number of the row at which partial vector v starts in the whole vector.

ie 1+ number of the row at which partial vector v ends in the whole vector.

$ierr$ Return code

Note

For Sequential and OpenMP processing, an n -th order vector results in $is = 0$ and $ie = n$.

6.1.7 lis_vector_set_value

```
C      int lis_vector_set_value(int flag, int i, LIS_SCALAR value, LIS_VECTOR v)
FORTRAN subroutine lis_vector_set_value(integer flag, integer i, LIS_SCALAR value,
      LIS_VECTOR v, integer ierr)
```

Capability

Assigning a scalar $value$ to the i -th row of vector v

Input

$flag$ LIS_INS_VALUE Assignment : $v[i] = value$
LIS_ADD_VALUE Assignment add: $v[i] = v[i] + value$

i Place at which the value should be assigned

$value$ Scalar value to assign

v Destination Vector

Output

v Vector with the scalar $value$ assigned to its i -th row.

$ierr$ Return code

Note

For MPI, the i -th row of the whole vector must be specified instead of the i -th row of the partial vector.

6.1.8 `lis_vector_get_value`

```
C      int lis_vector_get_value(LIS_VECTOR v, int i, LIS_SCALAR *value)
FORTRAN subroutine lis_vector_get_value(LIS_VECTOR v, integer i, LIS_SCALAR value,
      integer ierr)
```

Capability

Ascertaining a scalar of the i -th row of vector v .

Input

i	Place to which the value should be assigned
v	Destination Vector

Output

value	Scalar of its i -th row.
ierr	Return code

Note

For MPI, the i -th row of the whole vector must be specified rather than the i -th row of the partial vector.

6.1.9 `lis_vector_copy`

```
C      int lis_vector_copy(LIS_VECTOR x, LIS_VECTOR y)
FORTRAN subroutine lis_vector_copy(LIS_VECTOR x, LIS_VECTOR y, integer ierr)
```

Capability

Copying the value of a vector: $y \leftarrow x$

Input

x	Source vector
-----	---------------

Output

y	Destination vector
ierr	Return code

6.1.10 lis_vector_set_all

```
C      int lis_vector_set_all(LIS_SCALAR value, LIS_VECTOR x)
FORTRAN subroutine lis_vector_set_all(LIS_SCALAR value, LIS_VECTOR x, integer ierr)
```

Capability

Assigning a Scalar Value `value` to all elements of a vector

Input

<code>value</code>	Scalar value to assign
<code>v</code>	Target vector

Output

<code>v</code>	Vector with the <code>value</code> assigned to all its elements
<code>ierr</code>	Return code

6.2 Matrix Operations

Assume that the order of matrix A is $\text{global_n} \times \text{global_n}$ and that the number of rows of each partial matrix into which the row block of matrix A has been divided with `nprocs` units of processors is `local_n`. Here, `global_n` and `local_n` are called the global number of rows and the local number of rows, respectively.

6.2.1 `lis_matrix_create`

```
C      int lis_matrix_create(LIS_Comm comm, LIS_MATRIX *A)
FORTRAN subroutine lis_matrix_create(LIS_Comm comm, LIS_MATRIX A, integer ierr)
```

Capability

Creating matrix A

Input

`LIS_Comm` MPI communicator

Output

`A` Matrix
`ierr` Return code

Note

For Sequential and OpenMP processing, the value for `comm` is ignored.

6.2.2 `lis_matrix_destroy`

```
C      int lis_matrix_destroy(LIS_MATRIX A)
FORTRAN subroutine lis_matrix_destroy(LIS_MATRIX A, integer ierr)
```

Capability

Removing unwanted matrixes from memory

Input

`A` Matrix to remove from memory

Output

`ierr` Return code

6.2.5 lis_matrix_set_value

```
C      int lis_matrix_set_value(int flag, int i, int j, LIS_SCALAR value,  
                             LIS_MATRIX A)  
FORTRAN subroutine lis_matrix_set_value(integer flag, integer i, integer j,  
                                       LIS_SCALAR value, LIS_MATRIX A, integer ierr)
```

Capability

Assigning an element to the cell of i -th row and j -th column of matrix A

Input

flag	LIS_INS_VALUE Assignment :A(i,j) = value LIS_ADD_VALUE Assignment add:A(i,j) = A(i,j) + value
i	Row number of the matrix
j	Column number of the matrix
value	Value to assign
A	Matrix

Output

A	Matrix with the element assigned to the cell of the i -th row and j -th column
ierr	Return code

Note

For MPI, the i -th row and j -th column of the whole matrix must be specified, rather than the i -th row and j -th column of the partial matrix.

The `lis_matrix_set_value` function stores the assigned value in a temporary internal format. For this reason, when `lis_matrix_set_value` has been used, the `lis_matrix_assemble` function must be called.

6.2.6 lis_matrix_assemble

```
C      int lis_matrix_assemble(LIS_MATRIX A)  
FORTRAN subroutine lis_matrix_assemble(LIS_MATRIX A, integer ierr)
```

Capability

Building a matrix

Input

A	Matrix
---	--------

Output

A	Matrix built in the specified storage format
ierr	Return code

6.2.7 lis_matrix_set_size

```
int lis_matrix_set_size(LIS_MATRIX A, int local_n, int global_n)
FORTRAN subroutine lis_matrix_set_size(LIS_MATRIX A, integer local_n,
integer global_n, integer ierr)
```

Capability

Assigning matrix size

Input

A	Matrix
local_n	Local number of rows of matrix <i>A</i>
global_n	Global number of rows of matrix <i>A</i>

Output

ierr	Return code
------	-------------

Note

Either `local_n` or `global_n` must be provided. This function can create matrices in one of the following two ways: Creating partial matrices of order `local_n` x `N` from `local_n`, or creating partial matrices into which the row block of the vector of order `global_n` x `global_n` from `global_n` has been divided with a given number of processors. `N` represents the total sum of `local_n` of each processor.

In case of Sequential and OpenMP processing, `local_n = global_n`. This means that both `lis_matrix_set_size(A,n,0)` and `lis_matrix_set_size(A,0,n)` create a matrix of `n` x `n`.

For MPI, `lis_matrix_set_size(A,n,0)` creates at each processor *p* a partial matrix of $n_p \times N$, where *N* is the total sum of n_p of each processor. On the other hand, `lis_matrix_set_size(A,0,n)` creates at each processor *p* a partial matrix of $m_p \times n$, where m_p is the number of the partial matrix, which is determined by the library.

6.2.8 lis_matrix_get_size

```
C      int lis_matrix_get_size(LIS_MATRIX A, int *local_n, int *global_n)
FORTRAN subroutine lis_matrix_get_size(LIS_MATRIX A, integer local_n,
integer global_n, integer ierr)
```

Capability

Ascertaining matrix size

Input

A	Matrix
---	--------

Output

local_n	Local number of rows of matrix <i>A</i>
global_n	Global number of rows of matrix <i>A</i>
ierr	Return code

Note

In case of Sequential and OpenMP processing, `local_n = global_n`.

6.2.9 lis_matrix_get_range

```
C      int lis_matrix_get_range(LIS_MATRIX A, int *is, int *ie)
FORTRAN subroutine lis_matrix_get_range(LIS_MATRIX A, integer is, integer ie,
      integer ierr)
```

Capability

Ascertaining where in the whole matrix partial matrix A is located.

Input

A Partial matrix

Output

is Number of the row at which partial matrix A starts in the whole matrix.

ie 1+ the number of the row at which partial matrix A ends in the whole matrix

ierr Return code

Note

For Sequential and OpenMP processing, a matrix of $n \times n$ results in $is = 0$ and $ie = n$.

6.2.10 lis_matrix_set_type

```
C      int lis_matrix_set_type(LIS_MATRIX A, int matrix_type)
FORTRAN subroutine lis_matrix_set_type(LIS_MATRIX A, int matrix_type, integer ierr)
```

Capability

Assigning storage format

Input

A	Matirx
matrix_type	Storage format

Output

ierr	Return code
------	-------------

Note

matrix_type of A is LIS_MATRIX_CRS when the matrix is created. The table below shows the allowable storage formats for matrix_type.

storage format		matrix_type
Compressed Row Storage	(CRS)	LIS_MATRIX_CRS
Compressed Column Storage	(CCS)	LIS_MATRIX_CCS
Modified Compressed Sparse Row	(MSR)	LIS_MATRIX_MSR
Diagonal	(DIA)	LIS_MATRIX_DIA
Ellpack-Itpack generalized diagonal	(ELL)	LIS_MATRIX_ELL
Jagged Diagonal	(JDS)	LIS_MATRIX_JDS
Block Sparse Row	(BSR)	LIS_MATRIX_BSR
Block Sparse Column	(BSC)	LIS_MATRIX_BSC
Variable Block Row	(VBR)	LIS_MATRIX_VBR
Dense	(DNS)	LIS_MATRIX_DNS
Coordinate	(COO)	LIS_MATRIX_COO

6.2.11 lis_matrix_get_type

```
C      int lis_matrix_get_type(LIS_MATRIX A, int *matrix_type)
FORTRAN subroutine lis_matrix_get_type(LIS_MATRIX A, integer matrix_type,
integer ierr)
```

Capability

Ascertaining storage format

Input

A	Matrix
---	--------

Output

matrix_type	Storage format
ierr	Return code

6.2.12 lis_matrix_set_blocksize

```
C      int lis_matrix_set_blocksize(LIS_MATRIX A, int bnr, int bnc, int row[],
      int col[])
FORTRAN subroutine lis_matrix_set_blocksize(LIS_MATRIX A, integer bnr, integer bnc,
      integer row[], integer col[], integer ierr)
```

Capability

Assigning block size for BSR, BSC, and VBR

Input

A	Matrix
bnr	Row block size for BSR (BSC) or the number of row blocks for VBR
bnc	Column block size for BSR (BSC) or the number of column blocks for VBR
row	Array of the row division information about VBR
col	Array of the column division information about VBR

Output

ierr	Return code
------	-------------

6.2.13 lis_matrix_convert

```
C      int lis_matrix_convert(LIS_MATRIX Ain, LIS_MATRIX Aout)
FORTRAN subroutine lis_matrix_convert(LIS_MATRIX Ain, LIS_MATRIX Aout, integer ierr)
```

Capability

Converting the storage type of matrix *Ain* into a desired type to create matrix *Aout*.

Input

Ain	Source matrix
-----	---------------

Output

Aout	Matrix with the storage type converted into the specified type
ierr	Return code

Note

The specification of the converted storage format is set to *Aout* by using `lis_matrix_set_type`. The specification of the block size of BSR, BSC, and VBR is set to *Aout* by using `lis_matrix_set_blocksize`.

In converting the storage type of the source matrix into a specified type, the conversions indicated by 1 in the table below are performed directly, and the other conversions are made via the indicated types. The conversions with no indication are made via the CRS type.

Src \ Dst	CRS	CCS	MSR	DIA	ELL	JDS	BSR	BSC	VBR	DNS	COO
CRS	1	1	1	1	1	1	1	CCS	1	1	1
COO	1	1	1	CRS	CRS	CRS	CRS	CCS	CRS	CRS	1

6.2.14 lis_matrix_copy

```
C      int lis_matrix_copy(LIS_MATRIX Ain, LIS_MATRIX Aout)
FORTRAN subroutine lis_matrix_copy(LIS_MATRIX Ain, LIS_MATRIX Aout, integer ierr)
```

Capability

Copies the values of the elements

Input

Ain Source matrix

Output

Aout Destination matrix

ierr Return code

6.2.15 lis_matrix_get_diagonal

```
C      int lis_matrix_get_diagonal(LIS_MATRIX A, LIS_VECTOR d)
FORTRAN subroutine lis_matrix_get_diagonal(LIS_MATRIX A, LIS_VECTOR d, integer ierr)
```

Capability

Storing the diagonals of matrix A in vector d

Input

A Matrix

Output

d Vector that contains the diagonals of the matrix

ierr Return code

6.2.16 lis_matrix_set_crs

```
C      int lis_matrix_set_crs(int nnz, int *ptr, int *index, LIS_SCALAR *value,
                             LIS_MATRIX A)
FORTRAN unsupported
```

Capability

Associating the arrays required by the CRS format created by the user with matrix A

Input

nnz Number of non-zero elements

ptr, index, value Arrays for the CRS format

A Matrix

Output

A Matrix and associated arrays

Note

When the `lis_matrix_set_crs` function has been used, the `lis_matrix_assemble` function must be called.

6.2.17 `lis_matrix_set_ccs`

```
C      int lis_matrix_set_ccs(int nnz, int *ptr, int *index, LIS_SCALAR *value,  
                             LIS_MATRIX A)  
FORTRAN unsupported
```

Capability

Associating the arrays required by the CCS format created by the user with matrix A

Input

<code>nnz</code>	Number of non-zero elements
<code>ptr, index, value</code>	Arrays for the CCS format
<code>A</code>	Matrix

Output

<code>A</code>	Matrix and associated arrays
----------------	------------------------------

Note

When the `lis_matrix_set_ccs` function has been used, the `lis_matrix_assemble` function must be called.

6.2.18 `lis_matrix_set_msr`

```
C      int lis_matrix_set_msr(int nnz, int ndz, int *index, LIS_SCALAR *value,  
                             LIS_MATRIX A)  
FORTRAN unsupported
```

Capability

Associating the arrays required by the MSR format created by the user with matrix A

Input

<code>nnz</code>	Number of non-zero elements
<code>ndz</code>	Number of non-zero elements at the diagonal
<code>index, value</code>	Arrays for the MSR format
<code>A</code>	Matrix

Output

<code>A</code>	Matrix and associated arrays
----------------	------------------------------

Note

When the `lis_matrix_set_msr` function has been used, the `lis_matrix_assemble` function must be called.

6.2.19 `lis_matrix_set_dia`

```
C      int lis_matrix_set_dia(int nnd, int *index, LIS_SCALAR *value,  
                             LIS_MATRIX A)  
FORTRAN unsupported
```

Capability

Associating the arrays required by the DIA format created by the user with matrix A

Input

<code>nnd</code>	Number of non-zero diagonals
<code>index, value</code>	Arrays for the DIA format
<code>A</code>	Matrix

Output

<code>A</code>	Matrix and associated arrays
----------------	------------------------------

Note

When the `lis_matrix_set_dia` function has been used, the `lis_matrix_assemble` function must be called.

6.2.20 `lis_matrix_set_ell`

```
C      int lis_matrix_set_ell(int maxnzs, int index[], LIS_SCALAR value[],  
                             LIS_MATRIX A)  
FORTRAN unsupported
```

Capability

Associating the arrays required by the ELL format created by the user with matrix A

Input

<code>maxnzs</code>	Maximum number of non-zero elements at each row
<code>index, value</code>	Arrays for the ELL format
<code>A</code>	Matrix

Output

<code>A</code>	Matrix and associated arrays
----------------	------------------------------

Note

When the `lis_matrix_set_ell` function has been used, the `lis_matrix_assemble` function must be called.

6.2.21 `lis_matrix_set_jds`

```
C      int lis_matrix_set_jds(int nnz, int maxnzs, int *perm, int *ptr, int *index,  
                             LIS_SCALAR *value, LIS_MATRIX A)  
FORTRAN unsupported
```

Capability

Associating the arrays required by the JDS format created by the user with matrix A

Input

<code>nnz</code>	Number of non-zero elements
<code>maxnzs</code>	Maximum number of non-zero elements in each row
<code>perm, ptr, index, value</code>	Arrays for the JDS format
A	Matrix

Output

A	Matrix and associated arrays
-----	------------------------------

Note

When the `lis_matrix_set_jds` function has been used, the `lis_matrix_assemble` function must be called.

6.2.22 `lis_matrix_set_bsr`

```
C      int lis_matrix_set_bsr(int bnr, int bnc, int bnnz, int *bptr, int *bindex,  
                             LIS_SCALAR value[], LIS_MATRIX A)  
FORTRAN unsupported
```

Capability

Associating the arrays required by the BSR format created by the user with matrix A

Input

<code>bnr</code>	Row block size
<code>bnc</code>	Column block size
<code>bnnz</code>	Number of non-zero blocks
<code>bptr, bindex, value</code>	Arrays for the BSR format
A	Matrix

Output

A	Matrix and associated arrays
-----	------------------------------

Note

When the `lis_matrix_set_bsr` function has been used, the `lis_matrix_assemble` function must be called.

6.2.23 `lis_matrix_set_bsc`

```
C      int lis_matrix_set_bsc(int bnr, int bnc, int bnnz, int *bptr, int *bindex,
      LIS_SCALAR *value, LIS_MATRIX A)
FORTRAN unsupported
```

Capability

Associating the arrays required by the BSC format created by the user with matrix *A*

Input

<code>bnr</code>	Row block size
<code>bnc</code>	Column block size
<code>bnnz</code>	Number of non-zero blocks
<code>bptr, bindex, value</code>	Arrays for the BSC format
<code>A</code>	Matrix

Output

<code>A</code>	Matrix and associated arrays
----------------	------------------------------

Note

When the `lis_matrix_set_bsc` function has been used, the `lis_matrix_assemble` function must be called.

6.2.24 `lis_matrix_set_vbr`

```
C      int lis_matrix_set_vbr(int nnz, int nr, int nc, int bnnz, int *row, int *col,
      int *ptr, int *bptr, int *bindex, LIS_SCALAR *value, LIS_MATRIX A)
FORTRAN unsupported
```

Capability

Associating the arrays required by the VBR format created by the user with matrix *A*

Input

<code>nnz</code>	Number of all non-zero elements
<code>nr</code>	Number of row blocks
<code>nc</code>	Number of column blocks
<code>bnnz</code>	Number of non-zero blocks
<code>row, col, ptr, bptr, bindex, value</code>	Arrays for the VBR format
<code>A</code>	Matrix

Output

<code>A</code>	Matrix and associated arrays
----------------	------------------------------

Note

When the `lis_matrix_set_vbr` function has been used, the `lis_matrix_assemble` function must be called.

6.2.25 `lis_matrix_set_coo`

```
C      int lis_matrix_set_coo(int nnz, int row[], int col[], LIS_SCALAR value[],  
                             LIS_MATRIX A)  
FORTRAN unsupported
```

Capability

Associating the arrays required by the COO format created by the user with matrix A

Input

<code>nnz</code>	Number of non-zero elements
<code>row, col, value</code>	Arrays for the COO format
<code>A</code>	Matrix

Output

<code>A</code>	Matrix and associated arrays
----------------	------------------------------

Note

When the `lis_matrix_set_coo` function has been used, the `lis_matrix_assemble` function must be called.

6.2.26 `lis_matrix_set_dns`

```
C      int lis_matrix_set_dns(LIS_SCALAR value[], LIS_MATRIX A)  
FORTRAN unsupported
```

Capability

Associating the arrays required by the DNS format created by the user with matrix A

Input

<code>value</code>	Arrays for the DNS format
<code>A</code>	Matrix

Output

<code>A</code>	Matrix and associated arrays
----------------	------------------------------

Note

When the `lis_matrix_set_dns` function has been used, the `lis_matrix_assemble` function must be called.

6.3 Vector and Matrix Calculations

6.3.1 lis_vector_scale

```
C      int lis_vector_scale(LIS_SCALAR alpha, LIS_VECTOR x)
FORTRAN subroutine lis_vector_scale(LIS_SCALAR alpha, LIS_VECTOR x, integer ierr)
```

Capability

Multiplying all values of a vector by **alpha**: $x \leftarrow \alpha x$

Input

alpha	Scalar value
x	Vector to multiply by alpha

Output

x	Vector with all its elements multiplied by alpha
ierr	Return code

6.3.2 lis_vector_dot

```
C      int lis_vector_dot(LIS_VECTOR x, LIS_VECTOR y, LIS_SCALAR *val)
FORTRAN subroutine lis_vector_dot(LIS_VECTOR x, LIS_VECTOR y, LIS_SCALAR val,
                                integer ierr)
```

Capability

Calculating the inner product: $val \leftarrow x^T y$

Input

x	Vector
y	Vector

Output

val	Inner product value
ierr	Return code

6.3.3 lis_vector_nrm2

```
C      int lis_vector_nrm2(LIS_VECTOR x, LIS_SCALAR *val)
FORTRAN subroutine lis_vector_nrm2(LIS_VECTOR x, LIS_SCALAR val, integer ierr)
```

Capability

Calculating 2-norms of a vector: $val \leftarrow \|x\|_2$

Input

x	Vector
---	--------

Output

val	2-norms of the vector
ierr	Return code

6.3.7 lis_vector_axpyz

```
C      int lis_vector_axpyz(LIS_SCALAR alpha, LIS_VECTOR x, LIS_VECTOR y,  
                          LIS_VECTOR z)  
FORTRAN subroutine lis_vector_axpyz(LIS_SCALAR alpha, LIS_VECTOR x, LIS_VECTOR y,  
                                  LIS_VECTOR z, integer ierr)
```

Capability

Calculating $z \leftarrow \alpha x + y$

Input

alpha	Scalar value
x, y	Vector

Output

z	Calculation result of $x + \alpha y$
ierr	Return code

6.3.8 lis_matrix_scaling

```
C      int lis_matrix_scaling(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR d, int action)  
FORTRAN subroutine lis_matrix_scaling(LIS_MATRIX A, LIS_VECTOR b,  
                                  LIS_VECTOR d, integer action, integer ierr)
```

Capability

Performing matrix scaling

Input

A	Matrix
b	Vector
action	LIS_SCALE_JACOBI Jacobi scaling $D^{-1}Ax = D^{-1}b$, D represents the diagonal of $A = (a_{ij})$ LIS_SCALE_SYMM_DIAG Diagonal scaling $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$, $D^{-1/2}$ represents an diagonal matrix that has $1/\sqrt{a_{ii}}$ as an diagonal element

Output

d	Vector that contains the diagonals of D^{-1} or $D^{-1/2}$
ierr	Return code

6.3.9 lis_matvec

```
C      void lis_matvec(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y)
FORTRAN subroutine lis_matvec(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y)
```

Capability

Performing the matrix vector product $y \leftarrow Ax$

Input

A	Matrix
x	Vector

Output

y	Vector
---	--------

6.3.10 lis_matvect

```
C      void lis_matvect(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y)
FORTRAN subroutine lis_matvect(LIS_MATRIX A, LIS_VECTOR x, LIS_VECTOR y)
```

Capability

Performing the transpose matrix vector product $y \leftarrow A^T x$

Input

A	Matrix
x	Vector

Output

y	Vector
---	--------

6.4 Solve

6.4.1 lis_solver_create

```
C      int lis_solver_create(LIS_SOLVER *solver)
FORTRAN subroutine lis_solver_create(LIS_SOLVER solver, integer ierr)
```

Capability

Creating SOLVER

Input

None

Output

solver	SOLVER
ierr	Return code

Note

SOLVER has information on the iterative method, the preconditioner, etc.

6.4.2 lis_solver_destroy

```
C      int lis_solver_destroy(LIS_SOLVER solver)
FORTRAN subroutine lis_solver_destroy(LIS_SOLVER solver, integer ierr)
```

Capability

Removing unwanted SOLVER from memory

Input

solver	SOLVER to remove from memory
--------	------------------------------

Output

ierr	Return code
------	-------------

Specifying a Preconditioner Default: -p none

Preconditioner	Option	Auxiliary Option	
None	-p {none 0}		
Jacobi	-p {jacobi 1}		
ILU(k)	-p {ilu 2}	-ilu_fill [0]	Fill level k
SSOR	-p {ssor 3}	-ssor_w [1.0]	Relaxation Coefficient ω ($0 < \omega < 2$)
Hybrid	-p {hybrid 4}	-hybrid_i [sor]	Iterative method
		-hybrid_maxiter [25]	Maximum number of iterations
		-hybrid_tol [1.0e-3]	Convergence criteria
		-hybrid_w [1.5]	Relaxation Coefficient ω for the SOR method ($0 < \omega < 2$)
		-hybrid_e11 [2]	Value for l of the BiCGSTAB(l) method
		-hybrid_restart [40]	Restart values for GMRES and Orthomin
I+S	-p {is 5}	-is_alpha [1.0]	Parameter α for preconditioner of a $I + \alpha S^{(m)}$ type
		-is_m [3]	Parameter m for preconditioner of a $I + \alpha S^{(m)}$ type
SAINV	-p {sainv 6}	-sainv_drop [0.05]	Drop criteria
SA-AMG	-p {saamg 7}	-saamg_unsym [false]	Selection of asymmetric version
Crout ILU	-p {iluc 8}	-iluc_drop [0.05]	Drop criteria
		-iluc_rate [5.0]	Ratio of Maximum fill-in
ILUT	-p {ilut 9}	-ilut_drop [0.05]	Drop criteria
		-ilut_rate [5.0]	Ratio of Maximum fill-in
additive Schwarz	-adds true	-adds_iter [1]	Number of iterations

Other Options

Option	
-maxiter [1000]	Maximum number of iterations
-tol [1.0e-12]	Convergence criteria
-print [0]	Display of the residual
	-print {none 0} None
	-print {mem 1} Saves the residual history in memory
	-print {out 2} Displays the residual history
	-print {all 3} Saves the residual history and displays it on the screen
-scale [0]	Selection of scaling. The result will overwrite the original matrix and vectors
	-scale {none 0} No scaling
	-scale {jacobi 1} Jacobi scaling $D^{-1}Ax = D^{-1}b$ D represents the diagonal of $A = (a_{ij})$
	-scale {symm_diag 2} Diagonal scaling $D^{-1/2}AD^{-1/2}x = D^{-1/2}b$ $D^{-1/2}$ represents an diagonal matrix that has $1/\sqrt{a_{ii}}$ as an diagonal element
-initx_zeros [true]	Behavior of initial vector x_0
	-initx_zeros {false 0} Given values
	-initx_zeros {true 1} All elements are set to 0.
-omp_num_threads [t]	Number of execution threads t represents the maximum number of threads

Precision Default: -precision double

Precision	Option	Auxiliary Option
DOUBLE	-precision {double 0}	
QUAD	-precision {quad 1}	

6.4.4 lis_solver_set_optionC

```
C      int lis_solver_set_optionC(LIS_SOLVER solver)
FORTRAN subroutine lis_solver_set_optionC(LIS_SOLVER solver, integer ierr)
```

Capability

Setting options for iterative methods, preconditioners, and other options on the command line

Input

None

Output

solver	SOLVER
ierr	Return code

6.4.5 lis_solve

```
C      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, integer ierr)
```

Capability

Solving the linear equation $Ax = b$ with a desired iterative method and preconditioner

Input

A	Coefficient matrix
b	Right-side vector
x	Initial vector
solver	SOLVER

Output

x	Approximate solution
solver	Number of iterations, Execution time, etc.
ierr	Return code

6.4.8 lis_solver_get_time

```
C      int lis_solver_get_time(LIS_SOLVER solver, double *times, double *itimes,  
                             double *ptimes)  
FORTRAN subroutine lis_solver_get_time(LIS_SOLVER solver, real*8 times,  
                                       real*8 itimes, real*8 ptimes, integer ierr)
```

Capability

Ascertaining execution time from SOLVER

Input

solver SOLVER

Output

times Execution time (seconds)
times Iteration time (seconds)
times Preconditioner time (seconds)
ierr Return code

6.4.9 lis_solver_get_timeex

```
C      int lis_solver_get_timeex(LIS_SOLVER solver, double *times, double *itimes,  
                                double *ptimes, double *p_c_times, double *p_i_times)  
FORTRAN subroutine lis_solver_get_timeex(LIS_SOLVER solver, real*8 times,  
                                         real*8 itimes, real*8 ptimes, real*8 p_c_times, real*8 p_i_times,  
                                         integer ierr)
```

Capability

Ascertaining execution time from SOLVER

Input

solver SOLVER

Output

times Total time (seconds)
itimes Iteration time (seconds)
ptimes Preconditioner time (seconds)
p_c_times Preconditioner in creation time (seconds)
p_i_times Preconditioner in iteration time (seconds)
ierr Return code

6.5 File Input and Output

6.5.1 lis_input

```
C      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, integer ierr)
```

Capability

Reading matrix and vector data from a file

Input

filename	Name of the source file
----------	-------------------------

Output

A	Matrix in the storage format specified
b	Right-side vector
x	Solution vector
ierr	Return code

Note

The supported file formats are shown below:

- MatrixMarket format (extended to allow vector data to be read)
- Harwell-Boeing format

6.5.2 lis_input_vector

```
C      int lis_input_vector(LIS_VECTOR v, char *filename)
FORTRAN subroutine lis_input_vector(LIS_VECTOR v, character filename, integer ierr)
```

Capability

Reading vector data from a file

Input

filename	Name of the source file
----------	-------------------------

Output

v	Vector
ierr	Return code

Note

The supported file formats are shown below:

- PLAIN format
- MM format

6.5.5 lis_output_residual_history

```
C      int lis_output_residual_history(LIS_SOLVER solver, char *filename)
FORTRAN subroutine lis_output_residual_history(LIS_SOLVER solver, character filename)
```

Capability

Writing residual history into a file

Input

solver	SOLVER
filename	Name of the target file

Output

ierr	Return code
------	-------------

6.6 Other Functions

6.6.1 `lis_initialize`

```
C      int lis_initialize(int* argc, char** argv[])
FORTRAN subroutine lis_initialize(integer ierr)
```

Capability

Initialization

Input

<code>argc</code>	Number of command line arguments
<code>argv</code>	Command line argument

Output

<code>ierr</code>	Return code
-------------------	-------------

6.6.2 `lis_finalize`

```
C      void lis_finalize()
FORTRAN subroutine lis_finalize(integer ierr)
```

Capability

finalization

Input

None

Output

<code>ierr</code>	Return code
-------------------	-------------

6.6.3 `lis_wtime`

```
C      double lis_wtime()
FORTRAN function lis_wtime()
```

Capability

Measuring elapsed time

Input

None

Output

Time (in seconds) elapsed from a given point is returned as double data type

Note

To measure the processing time, use `lis_wtime` to measure the time immediately before starting time measurement and immediately after ending time measurement, and then determine the difference.

References

- [1] S. Fujino, M. Fujiwara and M. Yoshida. BiCGSafe method based on minimization of associate residual (in Japanese). Transactions of JSCES, Paper No.20050028, 2005. <http://save.k.u-tokyo.ac.jp/jsces/trans/trans2005/No20050028.pdf>.
- [2] T. Sogabe, M. Sugihara and S. Zhang. An Extension of the Conjugate Residual Method for Solving Nonsymmetric Linear Systems(in Japanese). Transactions of the Japan Society for Industrial and Applied Mathematics, Vol. 15, No. 3, pp. 445–460, 2005.
- [3] 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.
- [4] 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.
- [5] Y. Saad. A Flexible Inner-outer Preconditioned GMRES Algorithm. SIAM J. Sci. Stat. Comput., Vol. 14, pp. 461–469, 1993.
- [6] Y. Saad. ILUT: a dual threshold incomplete LU factorization. Numerical linear algebra with applications, Vol. 1, No. 4, pp. 387–402, 1994.
- [7] ITSOL: ITERATIVE SOLVERS package
<http://www-users.cs.umn.edu/~saad/software/ITSOL/index.html>.
- [8] N. Li, Y. Saad and E. Chow. Crout version of ILU for general sparse matrices. SIAM J. Sci. Comput., Vol. 25, pp. 716–728, 2003.
- [9] Toshiyuki Kohno, Hisashi Kotakemori and Hiroshi Niki. Improving the Modified Gauss-Seidel Method for Z -matrices. Linear Algebra and its Applications, Vol. 267, pp. 113–123, 1997.
- [10] A. Fujii, A. Nishida and Y. Oyanagi. A Parallel AMG Algorithm Based on Domain Decomposition (in Japanese). IPSJ Transactions on Advanced Computing Systems, Vol. 44, No. SIG6(ACS1), pp. 1–8, 2003.
- [11] K. Abe, S. Zhang, H. Hasegawa and R. Himeno. A SOR-base Variable Preconditioned CGR Method (in Japanese). Trans. JSIAM, Vol. 11, No. 4, pp. 157–170, 2001.
- [12] R. Bridson and W.-P. Tang. Refining an approximate inverse. J. Comput. Appl. Math., Vol. 123, pp. 293–306, 2000.
- [13] P. Soonerveld and M. B. van Gijzen. IDR(s): a family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. TR 07-07, Math. Anal., Delft Univ. of Tech., 2007. http://ta.twi.tudelft.nl/TWA_Reports/07-07.pdf.
- [14] H. Hasegawa. Utilizing the Quadruple-Precision Floating-Point Arithmetic Operation for the Krylov Subspace Methods. the 8th SIAM Conference on Applied Linear Algebra, 2003.
- [15] D. H. Bailey. A fortran-90 double-double library. <http://www.nersc.gov/~dhbailey/mpdist/mpdist.html>.
- [16] 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.
- [17] T. Dekker. A floating-point technique for extending the available precision. Numerische Mathematik, vol.18 pp.224–242, 1971.
- [18] D. E. Knuth., The Art of Computer Programming: Seminumerical Algorithms, vol.2., Addison-Wesley, 1969.

- [19] D. H. Bailey., High-precision arithmetic in scientific computation. In *SC06 Technical Paper*, 2006.
- [20] Intel Fortran Compiler User's Guide Vol I.
- [21] H. Kotakemori, A. Fujii, H. Hasegawa and A. Nishida. Fast Quad Precision Iterative Solver Library Lis using SSE2 (in Japanese). IPSJ SIG Technical Report, 2006-HPC-108, pp. 7–12, 2006.
- [22] R. Barrett, et al. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, 1994.
- [23] Y. Saad. SPARSKIT: a basic tool kit for sparse matrix computations, version 2, June 1994. <http://www.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html>.
- [24] S. Balay, et al. PETSc users manual. Technical Report ANL-95/11, Argonne National Laboratory, August 2004.
- [25] R. S. Tuminaro, et al. Official Aztec user's guide, version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, November 1999.
- [26] R. Bramley and X. Wang. SPLIB: A library of iterative methods for sparse linear system. Technical report, Indiana University–Bloomington, 1995.
- [27] Matrix Market. <http://math.nist.gov/MatrixMarket>.

A Appendix A: File Formats

This section describes the file formats available for the library.

A.1 MatrixMarket Format

The MatrixMarket format[27] is not designed to store vector data. For this library, it has been so extended that it can store vector data. Assume that the number of non-zero elements for matrix $A = (a_{ij})$ of $M \times N$ is L and that $a_{ij} = A(I,J)$. The file is structured as follows:

```
%%MatrixMarket matrix coordinate real general <-- Header
%
% | Comment lines with 0 or more lines
%
% <--
M N L B X <-- Numbers of rows, columns, and
I1 J1 A(I1,J1) <-- non-zero elements (0 or 1) (0 or 1)
I2 J2 A(I2,J2) | Row and column number values
. . . | The index is 1-base
IL JL A(IL,JL) <--
I1 B(I1) <--
I2 B(I2) | Exists only when B=1
. . . | Row number value
IM B(IM) <--
I1 X(I1) <--
I2 X(I2) | Exists only when X=1
. . . | Row number value
IM X(IM) <--
```

The MatrixMarket file for matrix A and vector b in Equation (A.1) is structured as follows:

$$A = \begin{pmatrix} 2 & 1 & & \\ 1 & 2 & 1 & \\ & 1 & 2 & 1 \\ & & 1 & 2 \end{pmatrix} \quad b = \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} \quad (\text{A.1})$$

```
%%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 Format

The Harwell-Boeing format inputs and outputs the matrix in the CCS storage format. Assume that the array `value` stores the value of non-zero elements of matrix A , the array `index` stores the row indices of the non-zero elements and the array `ptr` stores pointers to the beginning of each column in the arrays `value` and `index`. The file is structured as follows:

```

Line 1 (A72,A8)
  1 - 72 Title
  73 - 80 Key
Line 2 (5I14)
  1 - 14 Total number of lines excluding header
  15 - 28 Number of lines for ptr
  29 - 42 Number of lines for index
  43 - 56 Number of lines for value
  57 - 70 Number of lines for right-hand sides
Line 3 (A3,11X,4I14)
  1 - 3 Matrix type
      Col.1: R Real matrix
            C Complex matrix (Non-support)
            P Pattern only (Non-support)
      Col.2: S Symmetric
            U Unsymmetric
            H Hermitian (Non-support)
            Z Skew symmetric (Non-support)
            R Rectangular (Non-support)
      Col.3: A Assembled
            E Elemental matrices (Non-support)
  4 - 14 Blank space
  15 - 28 Number of rows
  29 - 42 Number of columns
  43 - 56 Number of non-zero elements
  57 - 70 0
Line 4 (2A16,2A20)
  1 - 16 Format for ptr
  17 - 32 Format for index
  33 - 52 Format for value
  53 - 72 Format for right-hand sides
Line 5 (A3,11X,2I14) Only present if there are right-hand sides present
  1   Right-hand side type
      F for full storage
      M for same format as matrix (Non-support)
  2   G if a starting vector is supplied
  3   X if an exact solution vector is supplied
  4 - 14 Blank space
  15 - 28 Number of right-hand sides
  29 - 42 Number of non-zero elements

```

The Harwell-Boeing file for matrix A and vector b in Equation (A.1) is structured as follows:

```

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           3           6           9
      1           2           1           2           3           2           3           4           3           4
2.00000000000000000000E+00  1.00000000000000000000E+00  1.00000000000000000000E+00
2.00000000000000000000E+00  1.00000000000000000000E+00  1.00000000000000000000E+00
2.00000000000000000000E+00  1.00000000000000000000E+00  1.00000000000000000000E+00
2.00000000000000000000E+00
0.00000000000000000000E+00  1.00000000000000000000E+00  2.00000000000000000000E+00
3.00000000000000000000E+00

```

A.3 MatrixMarketFormat (for Vectors)

The MatrixMarket format[27] has been extended so that it can store vector data. Assume that vector $b = (b_i)$ is a vector of order N and that $b_i = B(I)$. The file is structured as follows:

```
%%MatrixMarket vector coordinate real general <-- Header
% <--+
% | Comment lines with 0 or more lines
% <--+
N <-- Number of rows
I1 B(I1) <--+
I2 B(I2) | Row number value
. . . | The index is 1-base
IN B(IN) <--+
```

The MatrixMarket file for vector b in Equation (A.1) is structured as follows:

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

A.4 PLAIN Format (for Vectors)

The PLAIN format is designed to write vector values from the beginning in order. Assume that vector $b = (b_i)$ is a vector of order N and that $b_i = B(I)$. The file is structured as follows:

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

For vector b in Equation (A.1), the file is structured as follows:

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