Lis 1.2.29 Users Manual

This project is supported by "Development of Software Infrastructure for Large Scale Scientific Simulation" Team, CREST, JST.
http://www.ssisc.org/

July 21, 2010

Contents

0 Changes from Lis 1.1

1. Added eigensolvers.
2. Changed specifications of the following APIs:
   (a) Changed the names of lis_output_residual_history() and lis_get_residual_history() to lis_solver_output_rhistory() and lis_solver_get_rhistory(), respectively.
   (b) Changed origin of Fortran subroutines lis_vector_set_value() and lis_vector_get_value() to 1.

1 Introduction

Lis, a Library of Iterative Solvers for linear systems, is a numerical library written in C and Fortran for solving the linear equations of the form

$$Ax = b$$

and the standard eigenvalue problems of the form

$$Ax = \lambda x$$

with real sparse matrices using iterative methods. Lis has the following features:

- 22 linear equation solvers, 7 eigensolvers, and 10 preconditioners are supported
- 11 sparse matrix storage formats are supported
- Both the serial and parallel codes are supported by the common interface
- Both the double and quadruple precision operations are supported by the common interface

The solvers supported here are listed in Table 1 and 2, and the preconditioners are listed in Table 3. The matrix storage formats are listed in Table 4.

Table 1: Linear Equation Solvers

CG	CR
BiCG	BiCR[2]
CGS	CRS[3]
BiCGSTAB	BiCRSTAB[3]
GPBiCG	GPBiCR[3]
BiCGSafe[1]	BiCRSafe[4]
BiCGSTAB(l)	TFQMR
Jacobi	Orthomin(m)
Gauss-Seidel	GMRES(m)
SOR	FGMRES(m)[5]
IDR(s)[13]	MINRES[14]

Table 2: Eigensolvers

Power Iteration
Inverse Iteration
Approximate Inverse Iteration
Conjugate Gradient[18, 19]
Lanczos Iteration
Subspace Iteration
Conjugate Residual [20]

Table 3: Preconditioners

Jacobi
SSOR
ILU(k)
ILUT[6, 7]
Crout ILU[8, 7]
I+S[9]
SA-AMG[10]
Hybrid[11]
SAINV[12]
Additive Schwarz
User defined

Table 4: Matrix Storage Formats

Compressed Row Storage	(CRS)
Compressed Column Storage	(CCS)
Modified Compressed Sparse Row	(MSR)
Diagonal	(DIA)
Ellpack-Itpack generalized diagonal	(ELL)
Jagged Diagonal	(JDS)
Block Sparse Row	(BSR)
Block Sparse Column	(BSC)
Variable Block Row	(VBR)
Dense	(DNS)
Coordinate	(COO)

2 Installation

This section describes the instructions for installing and testing Lis. We assume Lis being installed on a Linux cluster.

2.1 Requirements

Installation of Lis requires a C compiler. Fortran interface requires a compiler which supports FORTRAN 77. AMG preconditioner requires a compiler which supports Fortran 90. For parallel computing environments, OpenMP or MPI-1 is used. Lis has been tested on the environments shown in Table 5 (see also Table 7).

Table 5: Major Tested Platforms

C Compilers	OS
Intel C/C++ Compiler 7.0, 8.0, 9.0, 9.1, 10.1, 11.0	Linux
IBM XL C/C++ V7.0, 9.0	AIX Linux
Sun WorkShop 6 update 2, Sun ONE Studio 7, Sun Studio 11, Sun Studio 12	Solaris
PGI C++ 6.0, 7.1, 10.5	Linux
gcc 3.3, 4.3	Linux
Fortran Compilers (Optional)	**OS**
Intel Fortran Compiler 8.1, 9.0, 9.1, 10.1, 11.0	Linux
IBM XL Fortran V9.1, 11.1	AIX Linux
Sun WorkShop 6 update 2, Sun ONE Studio 7, Sun Studio 11, Sun Studio 12	Solaris
PGI Fortran 6.0, 7.1, 10.5	Linux
g77 3.3 gfortran 4.3, 4.4 g95 0.91	Linux

2.2 Decompression

Enter the following command to decompress the archive, where `($VERSION)` represents the version:

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

It creates a directory `lis-($VERSION)` 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 Configuration

Enter the following command to run the script:

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

Table 6 shows the options which can be specified for the configuration. Table 7 shows the computing environments which can be specified by `TARGET`.

Table 6: Configuration Options

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

2.4 Compilation

In the `lis-($VERSION)` directory, run the script:

```
>make
```

To ensure that the library has been successfully built, enter as follows in the `lis-($VERSION)` directory:

```
>make check
```

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

Table 7: Major TARGET options (see `configure` for complete list)

<target>	Configure scripts
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"
fujitsu_pq	./configure CC=fcc FC=frt ac_cv_sizeof_void_p=8 CFLAGS="-O3 -Kfast,ocl,preex" FFLAGS="-O3 -Kfast,ocl,preex -Cpp" FCFLAGS="-O3 -Kfast,ocl,preex -Cpp -Am" ax_f77_mangling="lower case, underscore, no extra underscore"
hitachi	./configure CC=cc FC=f90 FCLDFLAGS="-lf90s" ac_cv_sizeof_void_p=8 CFLAGS="-Os -noparallel" FCFLAGS="-Oss -noparallel" ax_f77_mangling="lower case, underscore, no extra underscore"
ibm_bgl	./configure CC=blrts_xlc FC=blrts_xlf90 CFLAGS="-O3 -qarch=440d -qtune=440 -qstrict -I/bgl/BlueLight/ppcfloor/bglsys/include" FFFLAGS="-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"
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"
nec_sx9_cross	./configure CC=sxmpic++ FC=sxmpif90 AR=sxar RANLIB=true ac_cv_sizeof_void_p=8 ax_vector_machine=yes cross_compiling=yes ax_f77_mangling="lower case, no underscore, extra underscore"

default

```
100 x 100 matrix  460 entries
Initial vector x = 0
PRECISION : DOUBLE
SOLVER    : BiCG 2
PRECON    : None
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
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
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
```

2.5 Installation

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

```
>make install
```

It copies the files as follows:

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

`lis.h` and `lisf.h` are the header files required for C and Fortran, respectively. `liblis.a` is the library file.

2.6 Test Programs

2.6.1 test1

```
Usage: test1 matrix_filename rhs_setting solution_filename residual_filename [options]
```

It reads the data of the coefficient matrix from `matrix_filename` and solves the linear equation $Ax = b$ with the solver specified by `options`. It also writes the solution to `solution_filename` and the residual history to `residual_filename`. The MatrixMarket format is supported. One of the following values can be used for `rhs_setting`:

0	use the right hand side vector b, including the matrix file
1	use $b = (1, \ldots, 1)^T$
2	use $b = A \times (1, \ldots, 1)^T$
`rhs_filename`	right hand side vector filename

The PLAIN and MM formats are supported for `rhs_filename`. `test1f.F` is the Fortran version of `test1.c`.

2.6.2 test2

```
Usage: test2 m n matrix_type solution_filename residual_filename [options]
```

It solves a discretized two dimensional Poisson equation $Ax = b$ using the five point central difference scheme, with the coefficient matrix A of size mn in the storage format specified by `matrix_type` and the solver specified by `options`. It also writes the solution to `solution_filename` and the residual history to `residual_filename`. The right hand side vector is set to make all the elements for the solution to be 1. The values `m` and `n` represent the numbers of lattice points in each dimension.

2.6.3 test3

```
Usage: test3 l m n matrix_type solution_filename residual_filename [options]
```

It solves a discretized three dimensional Poisson equation $Ax = b$ using the seven point central difference scheme, with the coefficient matrix A of size lmn in the storage format specified by `matrix_type` and the solver specified by `options`. It also writes the solution to `solution_filename` and the residual history to `residual_filename`. The right hand side vector is set to make all the elements for the solution to be 1. The values `l`, `m` and `n` represent the numbers of lattice points in each dimension.

2.6.4 test4

This is a program for solving the linear equation $Ax = b$ with a specified solver and a preconditioner, where A is a tridiagonal matrix

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

of size 12. The right hand side vector b is set to make all the elements of the solution x to be 1. `test4f.F` is the Fortran version of `test4.c`.

2.6.5 test5

```
Usage: test5 n gamma [options]
```

This command solves a linear equation $Ax = b$, where A is a Toeplitz matrix

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

with the solver specified by `options`. Note that the right hand vector is set to make all the elements of the solution to be 1. The value `n` is the size of the matrix A. The value `gamma` is γ.

2.6.6 etest1

```
Usage: etest1 matrix_filename solution_filename residual_filename [options]
```

It reads the matrix data from `matrix_filename` and solves the eigenvalue problem $Ax = \lambda x$ with the solver specified by `options`. It writes the associated eigenvector to `solution_filename` and the residual history to `residual_filename`. The MatrixMarket format is supported. `etest1f.F` is the Fortran version of `etest1.c`.

2.6.7 etest2

```
Usage: etest2 m n matrix_type solution_filename residual_filename [options]
```

It solves the eigenvalue problem $Ax = \lambda x$, where the coefficient matrix A of size mn is derived from a discretized two dimensinal Helmholtz equation using the five point central difference scheme, with the coefficient matrix in the storage format specified by `matrix_type` and the solver specified by `options`. It writes the associated eigenvector to `solution_filename` and the residual history to `residual_filename`. The values `m` and `n` represent the numbers of lattice points in each dimension.

2.6.8 etest3

```
Usage: etest3 l m n matrix_type solution_filename residual_filename [options]
```

It solves the eigenvalue problem $Ax = \lambda x$, where the coefficient matrix A of size lmn is derived from a discretized three dimensinal Helmholtz equation using the seven point central difference scheme, with the coefficient matrix in the storage format specified by `matrix_type` and the solver specified by `options`. It writes the associated eigenvector to `solution_filename` and the residual history to `residual_filename`. The values `l`, `m` and `n` represent the numbers of lattice points in each dimension.

2.6.9 etest4

```
Usage: etest4 n [options]
```

It is a program for solving the eigenvalue problem $Ax = \lambda x$ with a specified solver, where A is a

tridiagonal matrix

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

of size $n \times n$. `etest4f.F` is the Fortran version of `etest4.c`.

2.6.10 etest5

```
Usage: etest5 evalue_filename evector_filename
```

It is a program for solving the eigenvalue problem $Ax = \lambda x$ with subspace iteration, where A is a tridiagonal matrix

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

of size 12×12. It writes 2 extreme eigenvalues of smallest magnitude to `evalue_filename` and the associated eigenvectors to `evector_filename` in the extended MatrixMarket format (see Appendix).

2.6.11 spmvtest1

```
Usage: spmvtest1 n iter
```

It computes the multiply of a tridiagonal matrix derived from a discretized one dimensional Poisson equation using the three point central difference scheme

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

of size n and a vector $(1, \ldots, 1)^T$. FLOPS performance is measured as the average of `iter` iterations.

2.6.12 spmvtest2

```
Usage: spmvtest2 m n iter
```

It computes the multiply of a sparse matrix, derived from a discretized two dimensional Poisson equation using the five point central difference scheme with available matrix storage formats, and a vector $(1, \ldots, 1)^T$. The values `m` and `n` represent the numbers of lattice points in the vertical and horizontal directions. FLOPS performance is measured as the average of `iter` iterations.

2.6.13 spmvtest3

```
Usage: spmvtest3 l m n iter
```

It computes the multiply of a sparse matrix, derived from a discretized three dimensional Poisson equation using the seven point central difference scheme with available matrix storage formats, and a

vector $(1, \ldots, 1)^T$. The values `l`, `m` and `n` represent the numbers of lattice points in each dimension. FLOPS performance is measured as the average of `iter` iterations.

2.6.14 spmvtest4

```
Usage: spmvtest4 matrix_filename_list iter
```

It reads the data of matrices from the files listed in `matrix_filename_list`, and computes the multiplies of the matrices with available matrix storage formats and a vector $(1, \ldots, 1)^T$. FLOPS performance is measured twice as the average of `iter` iterations.

2.6.15 spmvtest5

```
Usage: spmvtest5 matrix_filename matrix_type iter
```

It reads the data of a matrix from `matrix_filename` and compute the multiply of the matrix with `matrix_type` and a vector $(1, \ldots, 1)^T$. FLOPS performance is measured twice as the average of `iter` iterations.

2.7 Restrictions

The current version has the following restrictions:

- Preconditioners
  - If a preconditioner other than the Jacobi or SSOR is selected and the matrix A is not in the CRS format, a new matrix is created in the CRS format for preconditioning.
- Quadruple precision operations
  - Jacobi, Gauss-Seidel, SOR, and IDR(s) methods do not support quadruple precision operations.
  - Conjugate gradient and conjugate residual methods for eigenproblems do not support quadruple precision operations.
  - Jacobi, Gauss-Seidel and SOR do not support quadruple precision operations in the hybrid preconditioner.
  - I+S and SA-AMG preconditioners do not support quadruple precision operations.
- Matrix storage formats
  - In the MPI environment, CRS is the only accepted format for user defined arrays.

3 Basic Operations

This section describes how to use the library. A program requires the following statements:

- Initialization
- Matrix creation
- Vector creation
- Solver creation
- Value assignment for matrix and vector
- Solver assignment for linear equation or eigenvalue problem
- Solver execution
- Finalization

In addition, it must include one of the following `include` statements:

- `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 of the execution environment must be called at the beginning and the end of the program as follows:

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 call initialization, the following functions are used:

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

This function calls initialization of MPI, and specifies options on the command line.

Finalization

To call finalization, the following functions are used:

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

3.2 Vector Operations

Assume that the size of the vector v is `global_n`, and the size of each partial vector stored on `nprocs` processing elements is `local_n`. If `global_n` is divisible, then `local_n` = `global_n` / `nprocs`. For example, when the vector v is stored on two processing elements, as shown by Equation (3.1), `global_n` and `local_n` are 4 and 2, respectively.

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

In the case of creating the vector v in Equation (3.1), the serial and OpenMP versions create the vector v itself, while the MPI version creates partial vectors stored on a given number of processing elements.

Programs to create the vector v are as follows, where the number of the processing elements for the MPI version is assumed to be two:

C (for serial and OpenMP versions)

```
 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 (for MPI version)

```
 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:  }
```

Fortran (for serial and OpenMP versions)

```
 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
```

Fortran (for MPI version)

```
 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
```

Declaring Variables

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 *v)`
- Fortran `subroutine lis_vector_create(LIS_Comm comm, LIS_VECTOR v, integer ierr)`

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

Assigning Vector Sizes

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

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

Either `local_n` or `global_n` must be provided. This function creates a vector in one of the following ways: Creates partial vectors of size `local_n` if `local_n` is given, or create partial vectors of size `global_n`, stored on a given number of processing elements, if `global_n` is given.

In the case of the serial and OpenMP versions, `local_n` = `global_n`. It means that both `lis_vector_set_size(v,n,0)` and `lis_vector_set_size(v,0,n)` create a vector of size n.

For the MPI version, `lis_vector_set_size(v,n,0)` creates a partial vector of size n_p on the processing element p. On the other hand, `lis_vector_set_size(v,0,n)` creates a partial vector of size m_p on the processing element p. The value for m_p is determined by the library.

Assigning Elements

To assign an element to the i-th row of the 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 the MPI version, the i-th row of the global vector must be specified, rather than the i-th row of the partial vector. Either

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

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

must be provided for `flag`.

Duplicating Vectors

To create a vector which 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)`

This function does not copy the elements of the vector. To copy the elements as well, the following functions must be added after the above functions:

- C `int lis_vector_copy(LIS_VECTOR vsrc, LIS_VECTOR vdst)`
- Fortran `subroutine lis_vector_copy(LIS_VECTOR vsrc, LIS_VECTOR vdst, integer ierr)`

Destroying Vectors

To destroy a vector, the following functions are used:

- C `int lis_vector_destroy(LIS_VECTOR v)`
- Fortran `subroutine lis_vector_destroy(LIS_VECTOR v, integer ierr)`

3.3 Matrix Operations

Assume that the size of the matrix A is `global_n` $\times$ `global_n`, and that the size of each row block of the matrix A stored on `nprocs` processing elements is `local_n` $\times$ `global_n`. If `global_n` is divisible, then `local_n` = `global_n` / `nprocs`. For example, when the row block of the matrix A is stored on two processing elements, as shown by Equation (3.2), `global_n` and `local_n` are 4 and 2, respectively.

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

A matrix for a specific storage format can be created in one of the following three ways:

1. The library defines the arrays required in the specified storage type.
2. The user defines the arrays required in the specified storage type.
3. Matrix and vector data are read from a file.

Approach 1: The library defines the arrays required in the specific storage type

In the case of creating the matrix A in Equation (3.2) in the CRS format, the serial and OpenMP versions create the matrix A itself, and the MPI version creates on each processing element a partial matrix stored on given number of processing elements.

Programs to create the matrix A in the CRS format are as follows, where the number of the processing elements for the MPI version is assumed to be two:

C (for serial and OpenMP versions)

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

C (for MPI version)

```
 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 (for serial and OpenMP versions)

```
 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 (for MPI version)

```
 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)
```

Declaring Variables

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

```
LIS_MATRIX    A;
```

Creating Matrices

To create the 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 the serial and OpenMP versions, the value for `comm` is ignored.

Assigning Matrix Sizes

To assign a size to the 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 creates matrices in one of the following two ways: Create partial matrices of size `local_n x N` if `local_n` is given, or creating partial matrices of size `global_n x global_n` stored on a given number of processing elements, if `global_n` is given. `N` represents the total sum of `local_n`.

In the case of the serial and OpenMP versions, `local_n` = `global_n`. It 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 the MPI version, `lis_matrix_set_size(A,n,0)` creates a partial matrix of size $n_p \times N$ on the processing element p, where N is the total sum of n_p. On the other hand, `lis_matrix_set_size(A,0,n)` creates a partial matrix of size $m_p \times n$ on the processing element p, 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 the j-th column of the 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 the MPI version, the i-th row and the j-th column of the global matrix must be specified, rather than the i-th row and the j-th column of the partial matrix. Either

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

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

must be provided for `flag`.

Assigning Storage Formats

To assign a storage format to the 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 Matrices

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`.

Destroying Matrices

To destroy a 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 defines the arrays required in the specified storage type

In the case of creating the matrix A in Equation (3.2) in the CRS format, the serial and OpenMP versions creates the matrix A itself, and the MPI version creates on each processing element a partial matrix stored on a given number of processing elements.

Programs to create the matrix A in the CRS format are as follows, where the number of the processing elements for the MPI version is assumed to be two:

C (for serial and OpenMP versions)

```
 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 (for MPI version)

```
 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 the 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` `subroutine lis_matrix_set_crs(integer nnz, integer row(), integer index(), LIS_SCALAR value(), LIS_MATRIX A, integer ierr)`

Approach 3: Matrix and vector data are read from a file

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

C (for serial, OpenMP and MPI versions)

```
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 (for serial, OpenMP and MPI versions)

```
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 destination file `matvec.mtx` is as follows:

```
%%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 from Files

To read the data for the matrix `A` from a file, the following functions are used:

- C `int lis_input_matrix(LIS_MATRIX A, char *filename)`
- Fortran `subroutine lis_input(LIS_MATRIX A, character filename, integer ierr)`

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

- MatrixMarket format
- Harwell-Boeing format
- Lis format (original format)

To read the data for the 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 destination 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 Solving Linear Equations

A program to solve the linear equation $Ax = b$ with a specified solver is as follows:

C (for serial, OpenMP and MPI versions)

```
 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 (for serial, OpenMP and MPI versions)

```
 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 Solvers

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, 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)`

`lis_solver_set_optionC` is a function which sets the options specified on the command line, and pass them to `solver` when the user's program is run.

The table below shows the available command line options, where `-i {cg|1}` means `-i cg` or `-i 1` and `-maxiter [1000]` indicates that `-maxiter` defaults to 1,000.

Specifying Linear Equation Solvers (Default: `-i bicg`)

Method	Option	Auxiliary Option	
CG	`-i {cg|1}`		
BiCG	`-i {bicg|2}`		
CGS	`-i {cgs|3}`		
BiCGSTAB	`-i {bicgstab|4}`		
BiCGSTAB(l)	`-i {bicgstabl|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}`	`-irestart [2]`	Value for restart s
MINRES	`-i {minres|22}`		

Specifying Preconditioners (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]`	Linear equation solver
		`-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(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 unsymmetric version (Matrix structure must be symmetric.)
		`-saamg_theta [0.05|0.12]`	Drop criteria $a_{ij}^2 \leq \theta^2 \lvert a_{ii}\rvert\lvert a_{jj}\rvert$ (symmetric or unsymmetric)
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 a diagonal matrix with $1/\sqrt{a_{ii}}$ as diagonal
`-initx_zeros [true]`	Behavior of the 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 threads `t` represents the maximum number of threads	

Precision (Default: `-precision double`)

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

Solving Linear Equations

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 Solving Eigenvalue Problems

A program to solve the eigenvalue problem $Ax = \lambda x$ with a specified solver is as follows:

C (for serial, OpenMP and MPI versions)

```
 1: LIS_MATRIX A;
 2: LIS_VECTOR x;
 3: LIS_REAL evalue;
 4: LIS_ESOLVER esolver;
 5:
 6: /* Create matrix and vector */
 7:
 8: lis_esolver_create(&esolver);
 9: lis_esolver_set_option("-e ii -i bicg -p none",esolver);
10: lis_esolver_set_option("-etol 1.0e-12 -tol 1.0e-12",esolver);
11: lis_esolve(A,x,evalue,esolver);
```

Fortran (for serial, OpenMP and MPI versions)

```
 1: LIS_MATRIX A
 2: LIS_VECTOR x
 3: LIS_REAL evalue
 4: LIS_ESOLVER esolver
 5:
 6: /* Create matrix and vector */
 7:
 8: call lis_esolver_create(esolver,ierr)
 9: call lis_esolver_set_option('-e ii -i bicg -p none',esolver,ierr)
10: call lis_esolver_set_option('-etol 1.0e-12 -tol 1.0e-12',esolver,ierr)
11: call lis_esolve(A,x,evalue,esolver,ierr)
```

Creating Eigensolvers

To create an eigensolver, the following functions are used:

- C `int lis_esolver_create(LIS_ESOLVER *esolver)`
- Fortran `subroutine lis_esolver_create(LIS_ESOLVER esolver, integer ierr)`

Specifying Options

To specify options, the following functions are used:

- C `int lis_esolver_set_option(char *text, LIS_ESOLVER esolver)`
- Fortran `subroutine lis_esolver_set_option(character text, LIS_ESOLVER esolver, integer ierr)`

or

- C `int lis_esolver_set_optionC(LIS_ESOLVER esolver)`
- Fortran `subroutine lis_esolver_set_optionC(LIS_ESOLVER esolver, integer ierr)`

`lis_esolver_set_optionC` is a function which sets the options specified on the command line, and pass them to `esolver` when the user's program is run.

The table below shows the available command line options, where `-e {pi|1}` means `-e pi` or `-e 1` and `-emaxiter [1000]` indicates that `-emaxiter` defaults to 1,000.

Specifying Eigensolvers (Default: `-i bicg`)

Method	Option	Auxiliary Option	
Power Iteration	`-e {pi|1}`		
Inverse Iteration	`-e {ii|2}`	`-i [bicg]`	Linear equation solver
Approximate Inverse Iteration	`-e {aii|3}`		
Conjugate Gradient	`-e {cg|4}`		
Lanczos Iteration	`-e {li|5}`	`-ss [2]`	Size of subspace
		`-m [0]`	Mode
Subspace Iteration	`-e {si|6}`	`-ss [2]`	Size of subspace
		`-m [0]`	Mode
Conjugate Residual	`-e {cr|7}`		

Specifying Preconditioners (Default: `-p ilu`)

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]`	Linear equation solver
		`-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(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 unsymmetric version (Matrix structure must be symmetric.)
		`-saamg_theta [0.05|0.12]`	Drop criteria $a_{ij}^2 \leq \theta^2 \lvert a_{ii}\rvert\lvert a_{jj}\rvert$ (symmetric or unsymmetric)
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	
`-emaxiter [1000]`	Maximum number of iterations
`-etol [1.0e-12]`	Convergence criteria
`-eprint [0]`	Display of the residual
	`-eprint {none|0}` None
	`-eprint {mem|1}` Saves the residual history in memory
	`-eprint {out|2}` Displays the residual history
	`-eprint {all|3}` Saves the residual history and displays it on the screen
`-ie [ii]`	Inner eigensolver used in Lanczos Iteration or Subspace Iteration
	`-ie {pi|1}` Power Iteration (Subspace Iteration only)
	`-ie {ii|2}` Inverse Iteration
	`-ie {aii|3}` Approximate Inverse Iteration
`-shift [0.0]`	Amount of shift
`-initx_ones [true]`	Behavior of the initial vector x_0
	`-initx_ones {false|0}` Given values
	`-initx_ones {true|1}` All elements are set to 1.
`-omp_num_threads [t]`	Number of threads
	`t` represents the maximum number of threads

Precision (Default: `-eprecision double`)

Precision	Option	Auxiliary Option
DOUBLE	`-eprecision {double|0}`	
QUAD	`-eprecision {quad|1}`	

Solving Eivenvalue Problems

To solve the eigenvalue problem $Ax = \lambda x$, the following functions are used:

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

3.6 Sample Programs

The following are the programs for solving the linear equation $Ax = b$ with a specified solver to provide an approximate solution for the equation, where the matrix A is a tridiagonal matrix

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

of size 12. The the right hand side vector b is set to make all the elements of the solution x is 1. The program is located in the directory `lis-(&#36;VERSION)/test`.

Test program: test4.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(LIS_COMM_WORLD,&A);
12:     lis_matrix_set_size(A,0,n);
13:     lis_matrix_get_size(A,&n,&gn)
14:     lis_matrix_get_range(A,&is,&ie)
15:     for(i=is;i<ie;i++)
16:     {
17:         if( i>0 ) lis_matrix_set_value(LIS_INS_VALUE,i,i-1,-1.0,A);
18:         if( i<gn-1 ) lis_matrix_set_value(LIS_INS_VALUE,i,i+1,-1.0,A);
19:         lis_matrix_set_value(LIS_INS_VALUE,i,i,2.0,A);
20:     }
21:     lis_matrix_set_type(A,LIS_MATRIX_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: test4f.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_CRS,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
```

3.7 Compiling and Linking

Provided below is an example `test4.c` located in the directory `lis-($VERSION)/test`, compiled on SGI Altix 3700 using Intel C/C++ Compiler 8.1 (icc) and Intel Fortran Compiler 8.1 (ifort). Since the library includes Fortran 90 codes when the SA-AMG preconditioner is used, linking is processed by a Fortran 90 compiler.

for serial verion

Compiling

```
>icc -c -I$(INSTALLDIR)/include test4.c
```

Linking

```
>icc -o test4 test4.o -llis
```

Linking (with SA-AMG)

```
>ifort -nofor_main -o test4 test4.o -llis
```

for OpenMP verion

Compiling

```
>icc -c -openmp -I$(INSTALLDIR)/include test4.c
```

Linking

```
>icc -openmp -o test4 test4.o -llis
```

Linking (with SA-AMG)

```
>ifort -nofor_main -openmp -o test4 test4.o -llis
```

for MPI version

Compiling

```
>icc -c -DUSE_MPI -I$(INSTALLDIR)/include test4.c
```

Linking

```
>icc -o test4 test4.o -llis -lmpi
```

Linking (with SA-AMG)

```
>ifort -nofor_main -o test4 test4.o -llis -lmpi
```

for OpenMP + MPI version

Compiling

```
>icc -c -openmp -DUSE_MPI -I$(INSTALLDIR)/include test4.c
```

Linking

```
>icc -openmp -o test4 test4.o -llis -lmpi
```

Linking (with SA-AMG)

```
>ifort -nofor_main -openmp -o test4 test4.o -llis -lmpi
```

Provided below is an example `test4f.F` located in the directory `lis-($VERSION)/test`, compiled on SGI Altix 3700 using Intel Fortran Compiler 8.1 (ifort). Since an `#include` statement is used in the program, a compiler option `-fpp` is specified to use the preprocessor.

for serial version

Compiling

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

Linking

```
>ifort -o test4 test4.o -llis
```

for OpenMP version

Compiling

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

Linking

```
>ifort -openmp -o test4 test4.o -llis
```

for MPI version

Compiling

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

Linking

```
>ifort -o test4 test4.o -llis -lmpi
```

for OpenMP + MPI version

Compiling

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

Linking

```
>ifort -openmp -o test4 test4.o -llis -lmpi
```

3.8 Running

Run the test programs `test4` and `test4f` in the directory `lis-($VERSION)/test` by entering as follows:

for serial version

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

for OpenMP version

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

MPI

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

for OpenMP + MPI version

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

Double precision operations sometimes require a large number of iterations for convergence because of the rounding error. High precision operations are effective for the improvement of convergence. In Lis, we implement quadruple precision operations, which have "double-double" precision[15, 16] by combining two double precision floating point numbers. To use the quadruple precision operations with the same interface as the double precision operations, both the given matrix and vectors and the solution vectors are assumed to be double precision. Lis implements performance acceleration with SIMD instructions, such as Intel's SSE2[24].

4.1 Using Quadruple Precision Operations

The test program `test4.c` in the directory `lis-($VERSION)/test` supports SSE2 instructions:

Double precision (for serial version)
By entering `>./test4 200 2.0 -precision double`
the following results will be returned:

```
n=200 gamma=2.000000
Initial vector x = 0
PRECISION : DOUBLE
SOLVER    : BiCG 2
PRECON    : None
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
```

Quadruple precision (for serial version)
By entering `>./test4 200 2.0 -precision quad`
the following results will be returned:

```
n=200 gamma=2.000000
Initial vector x = 0
PRECISION : QUAD
SOLVER    : BiCG 2
PRECON    : None
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 Formats

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

5.1 Compressed Row Storage (CRS)

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

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

5.1.1 Creating Matrices (for Serial and OpenMP Versions)

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

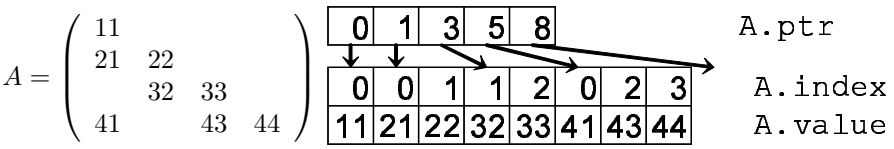

Figure 2: The data structure of the CRS format (for serial and OpenMP versions).

for serial and OpenMP versions

```
 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);
```

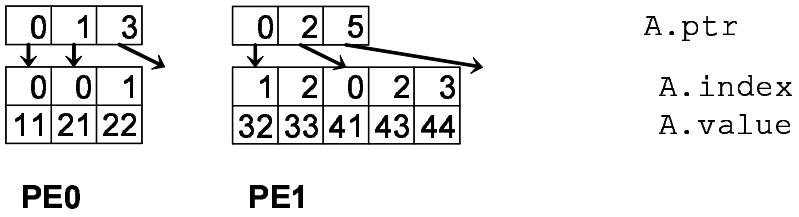

Figure 3: The data structure of the CRS format (for MPI version).

5.1.2 Creating Matrices (for MPI Version)

Figure 3 shows how the matrix A in Figure 2 is stored in the CRS format on two processing elements. A program to create the matrix in the CRS format on two processing elements is as follows:

for MPI version

```
 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 Associating Arrays

To associate the arrays required by the CRS format with the 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` `subroutine lis_matrix_set_crs(integer nnz, integer row(), integer index(), LIS_SCALAR value(), LIS_MATRIX A, integer ierr)`

5.2 Compressed Column Storage (CCS)

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

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

5.2.1 Creating Matrices (for Serial and OpenMP Versions)

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

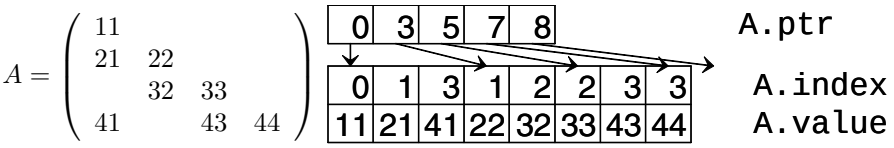

Figure 4: The data structure of the CCS format (for serial and OpenMP versions).

for serial and OpenMP versions

```
 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 Creating Matrices (for MPI Version)

Figure 5 shows how the matrix A in Figure 4 is stored on two processing elements. A program to create the matrix in the CCS format on two processing elements is as follows:

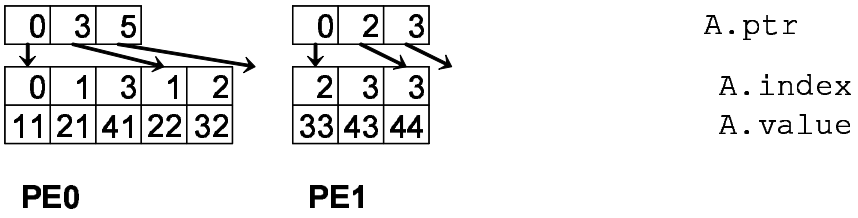

Figure 5: The data structure of the CCS format (for MPI version).

for MPI version

```
 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 Associating Arrays

To associate the arrays required by the CCS format with the matrix `A`, the following functions are used:

- C `int lis_matrix_set_ccs(int nnz, int row[], int index[], LIS_SCALAR value[],`

- Fortran `subroutine lis_matrix_set_ccs(integer nnz, integer row(), integer index(), LIS_SCALAR value(), LIS_MATRIX A, integer ierr)`

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 separates the diagonal elements before storing it. The MSR format uses two arrays `index` and `value` to store data. Assume that ndz represents the number of the zero elements of the diagonal.

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

5.3.1 Creating Matrices (for Serial and OpenMP Versions)

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

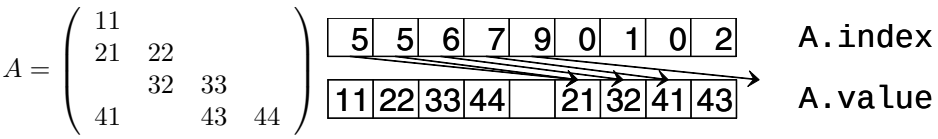

Figure 6: The data structure of the MSR format (for serial and OpenMP versions).

for serial and OpenMP versions

```
 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 Creating Matrices (for MPI Version)

Figure 7 shows how the matrix A in Figure 6 is stored in the MSR format on two processing element elements. A program to create the matrix in the MSR format on two processing element is as follows:

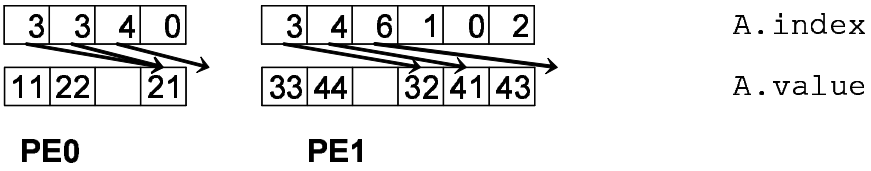

Figure 7: The data structure of the MSR format (for MPI version).

for MPI version

```
 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 Associating Arrays

To associate the arrays required by the MSR format with the matrix `A`, the following functions are used:

- `C  int lis_matrix_set_msr(int nnz, int ndz, int index[], LIS_SCALAR value[], LIS_MATRIX A)`
- `Fortran subroutine lis_matrix_set_msr(integer nnz, integer ndz, integer index(), LIS_SCALAR value(), LIS_MATRIX A, integer ierr)`

5.4 Diagonal (DIA)

DIA uses two arrays `index` and `value` to store data. Assume that *nnd* represents the number of the nonzero diagonal elements of the matrix A.

- `value` is a double-precision array with a length of $nnd \times n$, which stores nonzero diagonal elements of the matrix A.
- `index` is an integer array with a length of *nnd*, which stores the offsets from the main diagonal.

For the OpenMP version, the following modifications have been made: DIA uses two arrays `index` and `value` to store data. Assume that *nprocs* represents the number of the threads. nnd_p is the number of the nonzero diagonal elements of the partial matrix into which the row block of the matrix A is divided. *maxnnd* is the maximum value nnd_p.

- `value` is a double-precision array with a length of $maxnnd \times n$, which stores nonzero diagonal elements of the matrix A.
- `index` is an integer array with a length of $nprocs \times maxnnd$, which stores the offsets from the main diagonal.

5.4.1 Creating Matrices (for Serial Version)

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

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

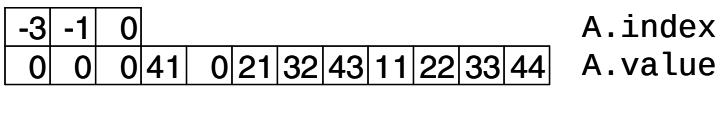

Figure 8: The data structure of the DIA format (for serial version).

for serial version

```
 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);
```

5.4.2 Creating Matrices (for OpenMP Version)

Figure 9 shows how the matrix A in Figure 8 is stored in the DIA format on two threads. A program to create the matrix in the DIA format on two threads is as follows:

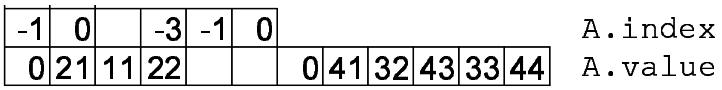

Figure 9: The data structure of the DIA format (for OpenMP version).

for OpenMP version

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

5.4.3 Creating Matrices (for MPI Version)

Figure 10 shows how the matrix A in Figure 8 is stored in the DIA format on two processing elements. A program to create the matrix in the DIA format on two processing elements is as follows:

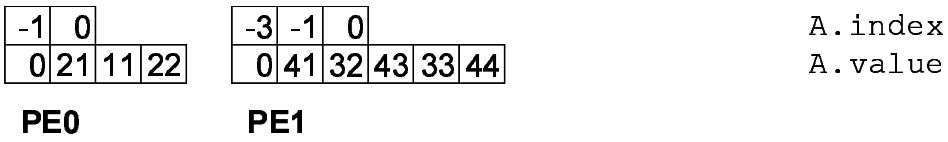

Figure 10: The data structure of the DIA format (for MPI version).

for MPI version

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

5.4.4 Associating Arrays

To associate the arrays required by the DIA format with the matrix `A`, the following functions are used:

- `C` `int lis_matrix_set_dia(int nnd, int index[], LIS_SCALAR value[], LIS_MATRIX A)`
- `Fortran` `subroutine lis_matrix_set_dia(integer nnd, integer index(), LIS_SCALAR value(), LIS_MATRIX A, integer ierr)`

5.5 Ellpack-Itpack generalized diagonal (ELL)

ELL uses two arrays `index` and `value` to store data. Assume that $maxnzr$ is the maximum value for the number of the nonzero elements in the rows of the matrix A.

- `value` is a double-precision array with a length of $maxnzr \times n$, which stores the nonzero elements of the rows of the matrix A along the column. The first column consists of the number of the first nonzero elements of each row. If there is no nonzero elements to be stored, then 0 is stored.
- `index` is an integer array with a length of $maxnzr \times n$, which stores the column numbers of the nonzero elements stored in the array `value`. If the number of the nonzero elements in the i-th row is nnz, then index[nnz_i*n + i] stores row number i.

5.5.1 Creating Matrices (for Serial and OpenMP Versions)

The right diagram in Figure 11 shows how the matrix A in Figure 11 is stored in the ELL format. A program to create the matrix in the ELL format is as follows:

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

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

Figure 11: The data structure of the ELL format (for serial and OpenMP versions).

for serial and OpenMP versions

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

5.5.2 Creating Matrices (for MPI Version)

Figure 12 shows how the matrix A in Figure 11 is stored in the ELL format. A program to create the matrix in the ELL format on two processing elements is as follows:

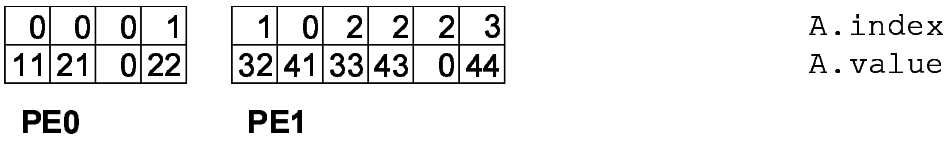

Figure 12: The data structure of the ELL format (for MPI version).

for MPI version

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

5.5.3 Associating Arrays

To associate an array required for the ELL format with the matrix `A`, the following functions are used:

- `C` `int lis_matrix_set_ell(int maxnzr, int index[], LIS_SCALAR value[], LIS_MATRIX A)`
- `Fortran` `subroutine lis_matrix_set_ell(integer maxnzr, integer index(), LIS_SCALAR value(), LIS_MATRIX A, integer ierr)`

5.6 Jagged Diagonal (JDS)

JDS first sorts the nonzero elements of the rows in decreasing order of size, and then stores them along the column. JDS uses four arrays `perm`, `ptr`, `index` and `value` to store date. Assume that $maxnzr$ represents the maximum value of the number of the nonzero elements of the matrix A.

- `perm` is an integer array with a length of n, which stores the sorted row numbers.
- `value` is a double-precision array with a length of nnz, which stores the jagged diagonal elements of the sorted matrix A. The first jagged diagonal consists of the first nonzero elements of each row. The next jagged diagonal consists of the second nonzero elements, and so on.
- `index` is an integer array with a length of nnz, which stores the row numbers of the nonzero elements stored in the array `value`.
- `ptr` is an integer array with a length of $maxnzr + 1$, which stores the starting points of the jagged diagonal elements.

For the OpenMP version, the following modifications have been made: JDS uses four arrays `perm`, `ptr`, `index` and `value` to store data. Assume that $nprocs$ is the number of the threads. $maxnzr_p$ is the number of the nonzero diagonal elements of the partial matrix into which the row block of the matrix A is divided. $maxmaxnzr$ is the maximum value of $maxnzr_p$.

- `perm` is an integer array with a length of n, which stores the sorted row numbers.
- `value` is a double-precision array with a length of nnz, which stores jagged diagonal elements of the sorted matrix A. The first jagged diagonal consists of the first nonzero elements of each row. The next jagged diagonal consist of the second nonzero elements of each row, and so on.
- `index` is an integer array with a length of nnz, which stores the row numbers of the nonzero elements stored in the array `value`.
- `ptr` is an integer array with a length of $nprocs \times (maxmaxnzr + 1)$, which stores the starting points of the jagged diagonal elements.

5.6.1 Creating Matrices (for Serial Version)

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

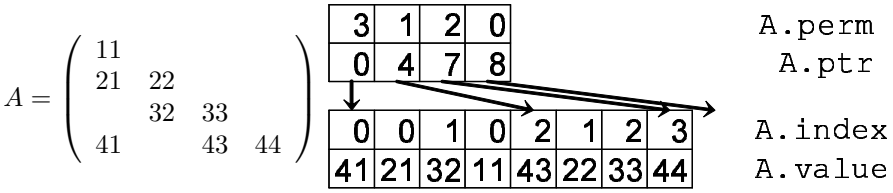

Figure 13: The data structure of the JDS format (for serial version).

for serial version

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

5.6.2 Creating Matrices (for OpenMP Version)

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

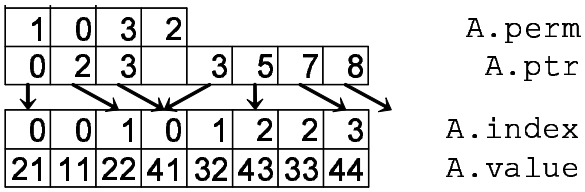

Figure 14: The data structure of the JDS format (for OpenMP version).

for OpenMP version

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

5.6.3 Creating Matrices (for MPI Version)

Figure 15 shows how the matrix A in Figure 13 is stored in the JDS format on two processing elements. A program to create the matrix in the JDS format on two processing elements is as follows:

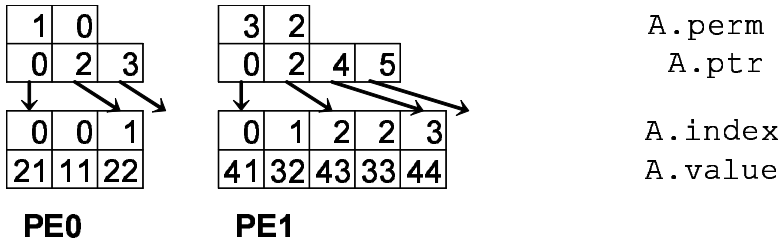

Figure 15: The data structure of the JDS format (for MPI version).

for MPI version

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

5.6.4 Associating Arrays

To associate an array required for the JDS format with the matrix `A`, the following functions are used:

- C `int lis_matrix_set_jds(int nnz, int maxnzr, int perm[], int ptr[], int index[], LIS_SCALAR value[], LIS_MATRIX A)`
- Fortran `subroutine lis_matrix_set_jds(integer nnz, integer maxnzr, integer ptr(), integer index(), LIS_SCALAR value(), LIS_MATRIX A, integer ierr)`

5.7 Block Sparse Row (BSR)

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

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

5.7.1 Creating Matrices (for Serial and OpenMP Versions)

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

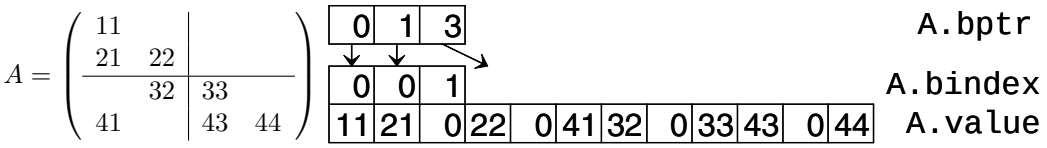

Figure 16: The data structure of the BSR format (for serial and OpenMP versions).

for serial and OpenMP versions

```
 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 Creating Matrices (for MPI Version)

Figure 17 shows how the matrix A in Figure 16 is stored in the BSR format on two processing elements. A program to create the matrix in the BSR format on two processing elements is as follows:

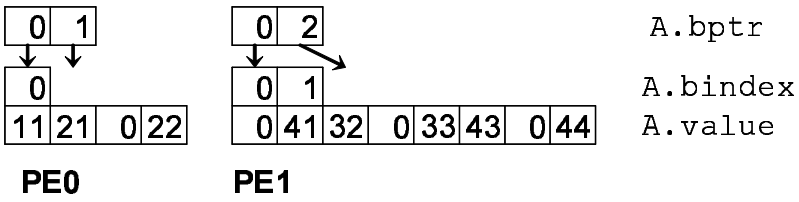

Figure 17: The data structure of the BSR format (for MPI version).

for MPI version

```
 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 Associating Arrays

To associate the arrays required by the BSR format with the matrix `A`, the following functions are used:

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

5.8 Block Sparse Column (BSC)

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

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

5.8.1 Creating Matrices (for Serial and OpenMP Versions)

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

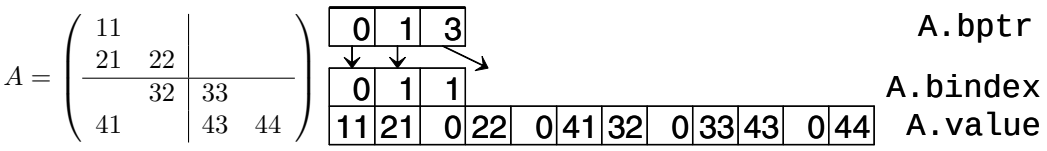

Figure 18: The data structure of the BSC format (for serial and OpenMP versions).

for serial and OpenMP versions

```
 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 Creating Matrices (for MPI Version)

Figure 19 shows how the matrix A in Figure 18 is stored in the BSC format on two processing elements. A program to create the matrix in the BSC format on two processing elements is as follows:

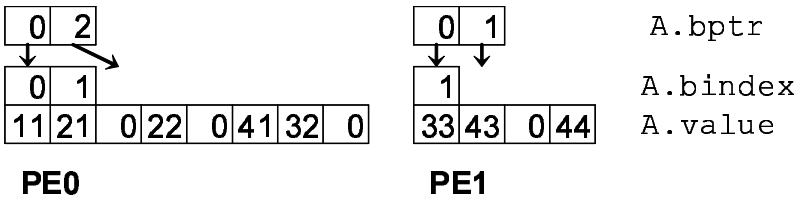

Figure 19: The data structure of the BSC format (for MPI version).

for MPI version

```
 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 Associating Arrays

To associate the arrays required by the BSC format with the matrix `A`, the following functions are used:

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

5.9 Variable Block Row (VBR)

VBR is the generalized version of BSR. The division points of the rows and columns are given by the arrays `row` and `col`. VBR stores the nonzero blocks (the blocks in which at least one nonzero 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 the nonzero blocks of A, and nnz denotes the total number of the elements of the nonzero blocks. VBR uses six arrays `bptr`, `bindex`, `row`, `col`, `ptr` and `value` to store matrices.

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

5.9.1 Creating Matrices (for Serial and OpenMP Versions)

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

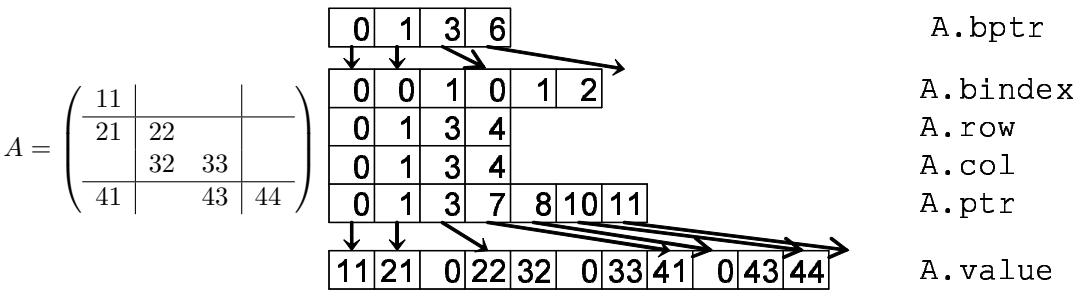

Figure 20: The data structure of the VBR format (for serial and OpenMP versions).

for serial and OpenMP versions

```
 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 Creating Matrices (for MPI Version)

Figure 21 shows how the matrix A in Figure 20 is stored in the VBR format on two processing elements. A program to create the matrix in the VBR format on two processing elements is as follows:

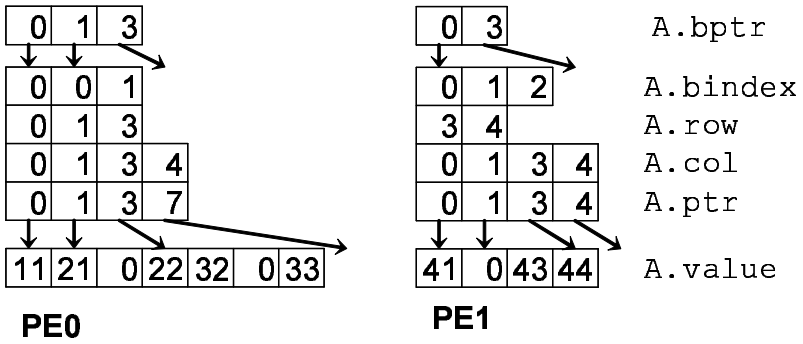

Figure 21: The data structure of the VBR format (for MPI version).

for MPI version

```
 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 Associating Arrays

To associate the arrays required by the VBR format with the matrix `A`, the following functions are used:

- `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 subroutine lis_matrix_set_vbr(integer nnz, integer nr, integer nc, integer bnnz, integer row(), integer col(), integer ptr(), integer bptr(), integer bindex(), LIS_SCALAR value(), LIS_MATRIX A, integer ierr)`

5.10 Coordinate (COO)

COO uses three arrays `row`, `col` and `value` to store data.

- `value` is a double-precision array with a length of nnz, which stores the nonzero elements.
- `row` is an integer array with a length of nnz, which stores the row numbers of the nonzero elements.
- `col` is an integer array with a length of nnz, which stores the column numbers of the nonzero elements.

5.10.1 Creating Matrices (for Serial and OpenMP Versions)

The right diagram in Figure 22 shows how the matrix A in Figure 22 is stored in the COO format. A program to create the matrix in the COO format is as follows:

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

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

Figure 22: The data structure of the COO format (for serial and OpenMP versions).

for serial and OpenMP versions

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

5.10.2 Creating Matrices (for MPI Version)

Figure 23 shows how the matrix A in Figure 22 is stored in the COO format on two processing elements. A program to create the matrix in the COO format on two processing elements is as follows:

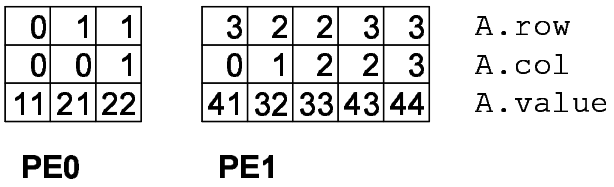

Figure 23: The data structure of the COO format (for MPI version).

for MPI version

```
 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 Associating Arrays

To associate the arrays required by the COO format with the matrix `A`, the following functions are used:

- `C` `int lis_matrix_set_coo(int nnz, int row[], int col[], LIS_SCALAR value[], LIS_MATRIX A)`
- `Fortran` `subroutine lis_matrix_set_coo(integer nnz, integer row(), integer col(), LIS_SCALAR value(), LIS_MATRIX A, integer ierr)`

5.11 Dense (DNS)

DNS uses one array `value` to store data.

- `value` is a double-precision array with a length of $n \times n$, which stores the elements with priority given to the columns.

5.11.1 Creating Matrices (for Serial and OpenMP Versions)

The right diagram in Figure 24 shows how the matrix A in Figure 24 is stored in the DNS format. A program to create the matrix in the DNS format is as follows:

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

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

A.Value

Figure 24: The data structure of the DNS format (for serial and OpenMP versions).

for serial and OpenMP versions

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

5.11.2 Creating Matrices (for MPI Version)

Figure 25 shows how the matrix A in Figure 24 is stored in the DNS format on two processing elements. A program to create the matrix in the DNS format on two processing elements is as follows:

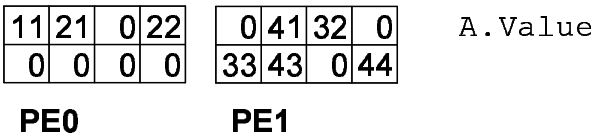

Figure 25: The data structure of the DNS format (for MPI version).

for MPI version

```
 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 Associating Arrays

To associate the arrays required by the DNS format with the matrix `A`, the following functions are used:

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

6 Functions

This section describes the functions which can be employed by users. The return values of the functions in C and the values of ierr in Fortran are as follows:

Return Values

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

6.1 Vector Operations

Assume that the size of the vector v is global_n and that the size of the partial vectors stored on nprocs processing elements is local_n. global_n and local_n are called the global size and the local size, respectively.

6.1.1 lis_vector_create

```
C       int lis_vector_create(LIS_Comm comm, LIS_VECTOR *v)
Fortran subroutine lis_vector_create(LIS_Comm comm, LIS_VECTOR v, integer ierr)
```

Description

Create vector

Input

LIS_Comm	MPI communicator

Output

v	Vector
ierr	Return code

Note

For the serial and OpenMP versions, the value for comm is ignored.

6.1.2 lis_vector_destroy

```
C       int lis_vector_destroy(LIS_VECTOR v)
Fortran subroutine lis_vector_destroy(LIS_VECTOR v, integer ierr)
```

Description

Destroy vector

Input

v	Vector to be destroyed

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)
```

Description

Create vector which has same information as original

Input

vin	Source vector

Output

vout	Destination vector
ierr	Return code

Note

The function `lis_vector_duplicate` 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 v, int local_n, int global_n)
Fortran subroutine lis_vector_set_size(LIS_VECTOR v, integer local_n,
         integer global_n, integer ierr)
```

Description

Assign size of vector

Input

v	Vector
local_n	Size of partial vector
global_n	Size of global vector

Output

ierr	Return code

Note

Either `local_n` or `global_n` must be provided. This function can create a vector in one of the following ways: Creates partial vectors of size `local_n` if `local_n` is given, or creates partial vectors stored on a given number of processing elements, if `global_n` is given.

In the case of the serial and OpenMP versions, `local_n` = `global_n`. It means that both `lis_vector_set_size(v,n,0)` and `lis_vector_set_size(v,0,n)` create a vector of size 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)
```

Description

Get size of vector

Input

v	Vector

Output

local_n	Size of partial vector
global_n	Size of global vector
ierr	Return code

Note

In the case of the serial and OpenMP versions, `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)
```

Description

Get location of partial vector in global vector

Input

v	Partial vector

Output

is	Location where partial vector v starts in global vector
ie	1+ location where partial vector v ends in global vector
ierr	Return code

Note

For the serial and OpenMP versions, a vector of size n 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)
```

Description

Assign scalar `value` to i-th row of vector `v`

Input

flag	`LIS_INS_VALUE` : v[i] = value `LIS_ADD_VALUE` : v[i] = v[i] + value
i	Location where value is assigned
value	Scalar value to be assigned
v	Destination vector

Output

v	Vector with scalar `value` assigned to i-th row
ierr	Return code

Note

For the MPI version, the i-th row of the global 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)
```

Description

Get value of i-th row of vector `v`

Input

i	Location where value should be assigned
v	Destination vector

Output

value	Value of i-th row
ierr	Return code

Note

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

6.1.9 lis_vector_set_values

```
C       int lis_vector_set_values(int flag, int count, int index[],
         LIS_SCALAR value[], LIS_VECTOR v)
Fortran subroutine lis_vector_set_values(integer flag, integer count,
         integer index(), LIS_SCALAR value(), LIS_VECTOR v, integer ierr)
```

Description

Assign scalar values `value[i]` to the `index[i]`-th row of vector `v`

Input

flag	`LIS_INS_VALUE` : v[index[i]] = value[i] `LIS_ADD_VALUE` : v[index[i]] = v[index[i]] + value[i]
count	Number of elements of array which stores scalar values to be assigned
index	Array which stores location where scalar values should be assigned
value	Array which stores scalar values to be assigned
v	Destination vector

Output

v	Vector with scalar `value[i]` assigned to its `index[i]`-th row
ierr	Return code

Note

For the MPI version, the `index[i]`-th row of the global vector must be specified instead of the `index[i]`-th row of the partial vector.

6.1.10 lis_vector_get_values

```
C       int lis_vector_get_values(LIS_VECTOR v, int start, int count,
         LIS_SCALAR value[])
Fortran subroutine lis_vector_get_values(LIS_VECTOR v, integer start,
         integer count, LIS_SCALAR value(), integer ierr)
```

Description

Get scalar values of `start` + i-th row of vector `v`, where $i = 0, 1, ..., $ `count` $- 1$

Input

start	Starting location
count	Number of values to get
v	Destination vector

Output

value	Vector to store scalar values
ierr	Return code

Note

For the MPI version, the `start` + i-th row of the global vector must be specified rather than the `start` + i-th row of the partial vector.

6.1.11 lis_vector_scatter

```
C       int lis_vector_scatter(LIS_SCALAR value[], LIS_VECTOR v)
Fortran subroutine lis_vector_scatter(LIS_SCALAR value(), LIS_VECTOR v, integer ierr)
```

Description

Assign scalar values of i-th row of vector v, where $i = 0, 1, ..., \texttt{global\_n} - 1$

Input

value	Array which stores scalar values to be assigned

Output

v	Destination vector
ierr	Return code

Note

6.1.12 lis_vector_gather

```
C       int lis_vector_gather(LIS_VECTOR v, LIS_SCALAR value[])
Fortran subroutine lis_vector_gather(LIS_VECTOR v, LIS_SCALAR value(), integer ierr)
```

Description

Get scalar values of i-th row of vector v, where $i = 0, 1, ..., \texttt{global\_n} - 1$

Input

v	Source vector

Output

value	Vector to store scalar values
ierr	Return code

Note

6.1.13 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)
```

Description

Copy vector: $y \leftarrow x$

Input

x	Source vector

Output

y	Destination vector
ierr	Return code

6.1.14 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)
```

Description

Assign scalar value `value` to all elements of vector `x`

Input

value	Scalar value to be assigned
v	Destination vector

Output

v	Vector with `value` assigned to all elements
ierr	Return code

6.2 Matrix Operations

Assume that the size of the matrix A is `global_n` $\times$ `global_n` and that the size of each partial matrix stored on `nprocs` processing elements is `local_n` $\times$ `global_n`. Here, `global_n` and `local_n` are called the number of the rows of the global matrix and the number of the rows of the partial matrix, 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)
```

Description
Create matrix

Input

LIS_Comm	MPI communicator

Output

A	Matrix
ierr	Return code

Note
For the equential and the OpenMP versions, 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)
```

Description
Destroy matrix

Input

A	Matrix to be destroyed

Output

ierr	Return code

6.2.3 lis_matrix_duplicate

```
C       int lis_matrix_duplicate(LIS_MATRIX Ain, LIS_MATRIX *Aout)
Fortran subroutine lis_matrix_duplicate(LIS_MATRIX Ain, LIS_MATRIX Aout,
         integer ierr)
```

Description

Create matrix which has same information as original

Input

Ain	Source matrix

Output

Aout	Destination matrix
ierr	Return code

Note

The function `lis_matrix_duplicate` does not copy the values of the elements of the matrix, but only reserves an area. To copy the values of the elements as well, the function `lis_matrix_copy` must be used.

6.2.4 lis_matrix_malloc

```
C       int lis_matrix_malloc(LIS_MATRIX A, int nnz_row, int nnz[])
Fortran subroutine lis_matrix_malloc(LIS_MATRIX A, integer nnz_row, integer nnz[],
         integer ierr)
```

Description

Allocate memory for matrix

Input

A	Matrix
nnz_row	Average number of nonzero elements
nnz	Array of numbers of nonzero elements in each row

Output

ierr	Return code

Note

Either `nnz_row` or `nnz` must be provided.

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)
```

Description

Assign value to (i, j) element of matrix

Input

flag	`LIS_INS_VALUE` : A(i,j) = value
	`LIS_ADD_VALUE` : A(i,j) = A(i,j) + value
i	Row number of matrix
j	Column number of matrix
value	Value to be assigned
A	Matrix

Output

A	Matrix
ierr	Return code

Note

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

The function `lis_matrix_set_value` stores the assigned value in a temporary internal format. For this reason, when `lis_matrix_set_value` is used, the function `lis_matrix_assemble` 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)
```

Description

Build matrix in specified storage format

Input

A	Matrix

Output

A	Matrix built in 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)
```

Description

Assign size of matrix

Input

A	Matrix
local_n	Number of rows of partial matrix
global_n	Number of rows of global matrix

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: Create partial matrices of size `local_n x N` if `local_n` is given, or create partial matrices of size `global_n x global_n` is stored in a given number of processing elements, if `global_n` is given. `N` represents the total sum of `local_n`.

In case of the serial and OpenMP versions, `local_n` = `global_n`. It 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 the MPI version, `lis_matrix_set_size(A,n,0)` creates on the processing element p a partial matrix of size $n_p \times N$, where N is the total sum of n_p. On the other hand, `lis_matrix_set_size(A,0,n)` creates on the processing element p a partial matrix of size $m_p \times n$, 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)
```

Description

Get size of matrix

Input

A	Matrix

Output

local_n	Number of rows of partial matrix
global_n	Number of rows of global matrix
ierr	Return code

Note

In case of the serial and OpenMP versions, `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)
```

Description

Get location of partial matrix A in global matrix

Input

A	Partial matrix

Output

is	Location where partial matrix A starts in global matrix
ie	1+ location where partial matrix A ends in global matrix
ierr	Return code

Note

For the serial and OpenMP versions, 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)
```

Description

Assign storage format

Input

A	Matrix
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 available 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)
```

Description

Get 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)
```

Description

Assign block size for BSR, BSC, and VBR

Input

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

Description

Convert matrix `Ain` into `Aout` of type specified with `lis_matrix_set_type`

Input

Ain	Source matrix

Output

Aout	Matrix converted into 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 one 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)
```

Description

Copy values of matrix 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)
```

Description

Store diagonal elements of matrix `A` to vector `d`

Input

A	Matrix

Output

d	Vector which stores diagonal elements of 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 subroutine lis_matrix_set_crs(integer nnz, integer row(), integer index(),
         LIS_SCALAR value(), LIS_MATRIX A, integer ierr)
```

Description

Set matrix `A` in CRS format

Input

nnz	Number of nonzero elements
ptr, index, value	Arrays for CRS format
A	Matrix

Output

A	Matrix and associated arrays

Note

When `lis_matrix_set_crs` is used, the function `lis_matrix_assemble` 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 subroutine lis_matrix_set_ccs(integer nnz, integer row(), integer index(),
         LIS_SCALAR value(), LIS_MATRIX A, integer ierr)
```

Description

Set matrix `A` in CCS format

Input

nnz	Number of nonzero elements
ptr, index, value	Arrays for CCS format
A	Matrix

Output

A	Matrix and associated arrays

Note

When `lis_matrix_set_ccs` is used, the function `lis_matrix_assemble` 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 subroutine lis_matrix_set_msr(integer nnz, integer ndz, integer index(),
         LIS_SCALAR value(), LIS_MATRIX A, integer ierr)
```

Description

Set matrix `A` in MSR format

Input

nnz	Number of nonzero elements
ndz	Number of nonzero elements in diagonal
index, value	Arrays for MSR format
A	Matrix

Output

A	Matrix and associated arrays

Note

When `lis_matrix_set_msr` is used, the function `lis_matrix_assemble` 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 subroutine lis_matrix_set_dia(integer nnd, integer index(),
         LIS_SCALAR value(), LIS_MATRIX A, integer ierr)
```

Description

Set matrix `A` in DIA format

Input

nnd	Number of nonzero diagonal elements
index, value	Arrays for DIA format
A	Matrix

Output

A	Matrix and associated arrays

Note

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

6.2.20 lis_matrix_set_ell

```
C       int lis_matrix_set_ell(int maxnzr, int index[], LIS_SCALAR value[],
         LIS_MATRIX A)
Fortran subroutine lis_matrix_set_ell(integer maxnzr, integer index(),
         LIS_SCALAR value(), LIS_MATRIX A, integer ierr)
```

Description

Set matrix `A` in ELL format

Input

maxnzr	Maximum number of nonzero elements in each row
index, value	Arrays for ELL format
A	Matrix

Output

A	Matrix and associated arrays

Note

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

6.2.21 lis_matrix_set_jds

```
C       int lis_matrix_set_jds(int nnz, int maxnzr, int perm[], int ptr[],
         int index[], LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_jds(integer nnz, integer maxnzr, integer ptr(),
         integer index(), LIS_SCALAR value(), LIS_MATRIX A, integer ierr)
```

Description

Set matrix `A` in JDS format

Input

nnz	Number of nonzero elements
maxnzr	Maximum number of nonzero elements in each row
perm, ptr, index, value	Arrays for JDS format
A	Matrix

Output

A	Matrix and associated arrays

Note

When `lis_matrix_set_jds` is used, the function `lis_matrix_assemble` 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 subroutine lis_matrix_set_bsr(integer bnr, integer bnc, integer bnnz,
         integer bptr(), integer bindex(), LIS_SCALAR value(), LIS_MATRIX A,
         integer ierr)
```

Description

Set matrix `A` in BSR format

Input

bnr	Row block size
bnc	Column block size
bnnz	Number of nonzero blocks
bptr, bindex, value	Arrays for BSR format
A	Matrix

Output

A	Matrix and associated arrays

Note

When `lis_matrix_set_bsr` is used, the function `lis_matrix_assemble` 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 subroutine lis_matrix_set_bsc(integer bnr, integer bnc, integer bnnz,
         integer bptr(), integer bindex(), LIS_SCALAR value(), LIS_MATRIX A,
         integer ierr)
```

Description

Set matrix `A` in BSC format

Input

bnr	Row block size
bnc	Column block size
bnnz	Number of nonzero blocks
bptr, bindex, value	Arrays for BSC format
A	Matrix

Output

A	Matrix and associated arrays

Note

When `lis_matrix_set_bsc` is used, the function `lis_matrix_assemble` 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 subroutine lis_matrix_set_vbr(integer nnz, integer nr, integer nc,
         integer bnnz, integer row(), integer col(), integer ptr(), integer bptr(),
         integer bindex(), LIS_SCALAR value(), LIS_MATRIX A, integer ierr)
```

Description

Set matrix `A` in VBR format

Input

nnz	Number of all nonzero elements
nr	Number of row blocks
nc	Number of column blocks
bnnz	Number of nonzero blocks
row, col, ptr, bptr, bindex, value	Arrays for VBR format
A	Matrix

Output

A	Matrix and associated arrays

Note

When `lis_matrix_set_vbr` is used, the function `lis_matrix_assemble` 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 subroutine lis_matrix_set_coo(integer nnz, integer row(), integer col(),
         LIS_SCALAR value(), LIS_MATRIX A, integer ierr)
```

Description

Set matrix `A` in COO format

Input

nnz	Number of nonzero elements
row, col, value	Arrays for COO format
A	Matrix

Output

A	Matrix and associated arrays

Note

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

6.2.26 lis_matrix_set_dns

```
C       int lis_matrix_set_dns(LIS_SCALAR value[], LIS_MATRIX A)
Fortran subroutine lis_matrix_set_dns(LIS_SCALAR value(), LIS_MATRIX A, integer ierr)
```

Description

Set matrix `A` in DNS format

Input

value	Array for DNS format
A	Matrix

Output

A	Matrix and associated arrays

Note

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

6.3 Vector and Matrix Operations

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)
```

Description

Multiply vector by `alpha`: $x \leftarrow \alpha x$

Input

alpha	Scalar value
x	Vector to be multiplied by `alpha`

Output

x	Vector 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)
```

Description

Calculate 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_nrm1

```
C       int lis_vector_nrm1(LIS_VECTOR x, LIS_SCALAR *val)
Fortran subroutine lis_vector_nrm1(LIS_VECTOR x, LIS_SCALAR val, integer ierr)
```

Description

Calculate 1-norm of vector: $val \leftarrow ||x||_1$

Input

x	Vector

Output

val	1-norm of vector
ierr	Return code

6.3.4 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)
```

Description

Calculate 2-norm of vector: $val \leftarrow ||x||_2$

Input

x	Vector

Output

val	2-norms of vector
ierr	Return code

6.3.5 lis_vector_nrmi

```
C       int lis_vector_nrmi(LIS_VECTOR x, LIS_SCALAR *val)
Fortran subroutine lis_vector_nrmi(LIS_VECTOR x, LIS_SCALAR val, integer ierr)
```

Description

Calculate infinity norm of vector: $val \leftarrow ||x||_\infty$

Input

x	Vector

Output

val	infinity norm of vector
ierr	Return code

6.3.6 lis_vector_axpy

```
C       int lis_vector_axpy(LIS_SCALAR alpha, LIS_VECTOR x, LIS_VECTOR y)
Fortran subroutine lis_vector_axpy(LIS_SCALAR alpha, LIS_VECTOR x, LIS_VECTOR y,
         integer ierr)
```

Description

Calculate $y \leftarrow \alpha x + y$

Input

alpha	Scalar value
x, y	Vectors

Output

y	$\alpha x + y$ (vector y is overwritten)
ierr	Return code

6.3.7 lis_vector_xpay

```
C       int lis_vector_xpay(LIS_VECTOR x, LIS_SCALAR alpha, LIS_VECTOR y)
Fortran subroutine lis_vector_xpay(LIS_VECTOR x, LIS_SCALAR alpha, LIS_VECTOR y,
         integer ierr)
```

Description

Calculate $y \leftarrow x + \alpha y$

Input

alpha	Scalar value
x, y	Vectors

Output

y	$x + \alpha y$ (vector y is overwritten)
ierr	Return code

6.3.8 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)
```

Description

Calculate $z \leftarrow \alpha x + y$

Input

alpha	Scalar value
x, y	Vectors

Output

z	$x + \alpha y$
ierr	Return code

6.3.9 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)
```

Description

Scale matrix

Input

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

Output

d	Vector which stores diagonal elements of D^{-1} or $D^{-1/2}$
ierr	Return code

6.3.10 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)
```

Description

Calculate matrix vector product $y \leftarrow Ax$

Input

A	Matrix
x	Vector

Output

y	Vector

6.3.11 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)
```

Description

Calculate transposed matrix vector product $y \leftarrow A^T x$

Input

A	Matrix
x	Vector

Output

y	Vector

6.4 Solving Linear Equations

6.4.1 lis_solver_create

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

Description
Create solver

Input

None

Output

solver	Solver
ierr	Return code

Note
`solver` has the information on the solver, 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)
```

Description
Destroy solver

Input

solver	Solver to be destroyed

Output

ierr	Return code

6.4.3 lis_solver_set_option

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

Description

Set options for solver

Input

text	Command line options

Output

solver	Solver
ierr	Return code

Note

The table below shows the available command line options, where `-i {cg|1}` means `-i cg` or `-i 1` and `-maxiter [1000]` indicates that `-maxiter` defaults to 1,000.

Specifying Linear Equation Solvers (Default: `-i bicg`)

Method	Option	Auxiliary Option	
CG	`-i {cg|1}`		
BiCG	`-i {bicg|2}`		
CGS	`-i {cgs|3}`		
BiCGSTAB	`-i {bicgstab|4}`		
BiCGSTAB(l)	`-i {bicgstabl|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}`	`-irestart [2]`	Value for restart s
MINRES	`-i {minres|22}`		

Specifying Preconditioners (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]`	Linear equation solver
		`-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(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 unsymmetric version (Matrix structure must be symmetric.)
		`-saamg_theta [0.05|0.12]`	Drop criteria $a_{ij}^2 \leq \theta^2 \lvert a_{ii}\rvert \lvert a_{jj}\rvert$ (symmetric or unsymmetric)
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 a diagonal matrix with $1/\sqrt{a_{ii}}$ as diagonal
`-initx_zeros [true]`	Behavior of the 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 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)
```

Description

Set options for solver on 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)
```

Description

Solve linear equation $Ax = b$ with specified solver

Input

A	Coefficient matrix
b	Right hand side vector
x	Initial vector
solver	Solver

Output

x	Solution
solver	Number of iterations, execution time, etc.
ierr	Return code (0)

6.4.6 lis_solve_kernel

```
C       int lis_solve_kernel(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,
         LIS_SOLVER solver, LIS_PRECON, precon)
Fortran subroutine lis_solve_kernel(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,
         LIS_SOLVER solver, LIS_PRECON precon, integer ierr)
```

Description

Solve linear equation $Ax = b$ with specified solver and predefined preconditioner

Input

A	Coefficient matrix
b	Right hand side vector
x	Initial vector
solver	Solver
precon	Preconditioner

Output

x	Solution
solver	Number of iterations, execution time, etc.
ierr	Return code (0)

6.4.7 lis_solver_get_status

```
C       int lis_solver_get_status(LIS_SOLVER solver, int *status)
Fortran subroutine lis_solver_get_status(LIS_SOLVER solver, integer status,
         integer ierr)
```

Description

Get status from solver

Input

solver	Solver

Output

status	Number of iterations
ierr	Return code

6.4.8 lis_solver_get_iters

```
C       int lis_solver_get_iters(LIS_SOLVER solver, int *iters)
Fortran subroutine lis_solver_get_iters(LIS_SOLVER solver, integer iters,
         integer ierr)
```

Description

Get number of iterations from solver

Input

solver	Solver

Output

iters	Number of iterations
ierr	Return code

6.4.9 lis_solver_get_itersex

```
C       int lis_solver_get_itersex(LIS_SOLVER solver, int *iters, int *iters_double,
         int *iters_quad)
Fortran subroutine lis_solver_get_itersex(LIS_SOLVER solver, integer iters,
         integer iters_double, integer iters_quad, integer ierr)
```

Description

Get number of iterations from solver

Input

solver	Solver

Output

iters	Number of iterations
iters_double	Number of double precision iterations
iters_quad	Number of quadruple precision iterations
ierr	Return code

6.4.10 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)
```

Description

Get execution time from solver

Input

solver	Solver

Output

times	Time in seconds for execution
times	Time in seconds for iteration
times	Time in seconds for preconditioning
ierr	Return code

6.4.11 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)
```

Description

Get execution time from solver

Input

solver	Solver

Output

times	Total time in seconds
itimes	Time in seconds for iteration
ptimes	Time in seconds for preconditioning
p_c_times	Time in seconds for creating preconditioner
p_i_times	Time in seconds for iteration in preconditioner
ierr	Return code

6.4.12 lis_solver_get_residualnorm

```
C       int lis_solver_get_residualnorm(LIS_SOLVER solver, LIS_REAL *residual)
Fortran subroutine lis_solver_get_residualnorm(LIS_SOLVER solver, LIS_REAL residual,
         integer ierr)
```

Description

Get 2-norm of $b - Ax$ from solver

Input

solver	Solver

Output

residual	2-norms of $b - Ax$
ierr	Return code

6.4.13 lis_solver_get_rhistory

```
C       int lis_solver_get_rhistory(VECTOR v)
Fortran subroutine lis_solver_get_rhistory(LIS_VECTOR v, integer ierr)
```

Description

Store residual norm history of solver

Input

None

Output

v	Vector
ierr	Return code

Note

The vector `v` must be created in advance with the function `lis_vector_create`. When the vector `v` is shorter than the residual history, it stores the residual history in order to the vector `v`.

6.4.14 lis_solver_get_solver

```
C       int lis_solver_get_solver(LIS_SOLVER solver, int *nsol)
Fortran subroutine lis_solver_get_solver(LIS_SOLVER solver, integer nsol,
         integer ierr)
```

Description

Get solver number from solver

Input

solver	Solver

Output

nsol	Solver number
ierr	Return code

Note

The number of the solver is as follows:

Solver	Number	Solver	Number
CG	1	Gauss-Seidel	11
BiCG	2	SOR	12
CGS	3	BiCGSafe	13
BiCGSTAB	4	CR	14
BiCGSTAB(l)	5	BiCR	15
GPBiCG	6	CRS	16
TFQMR	7	BiCRSTAB	17
Orthomin(m)	8	GPBiCR	18
GMRES(m)	9	BiCRSafe	19
Jacobi	10	FGMRES(m)	20
IDR(s)	21	MINRES	22

6.4.15 lis_get_solvername

```
C       int lis_get_solvername(int nsol, char *name)
Fortran subroutine lis_get_solvername(integer nsol, character name, integer ierr)
```

Description

Get solver name from solver number

Input

nsol	Solver number

Output

name	Solver name
ierr	Return code

6.5 Solving Eigenvalue Problems

6.5.1 lis_esolver_create

```
C       int lis_esolver_create(LIS_ESOLVER *esolver)
Fortran subroutine lis_esolver_create(LIS_ESOLVER esolver, integer ierr)
```

Description
Create eigensolver

Input

None

Output

esolver	Eigensolver
ierr	Return code

Note
`esolver` has the information on the eigensolver, the preconditioner, etc.

6.5.2 lis_esolver_destroy

```
C       int lis_esolver_destroy(LIS_ESOLVER esolver)
Fortran subroutine lis_esolver_destroy(LIS_ESOLVER esolver, integer ierr)
```

Description
Destroy eigensolver

Input

esolver	Eigensolver to be destoyed

Output

ierr	Return code

6.5.3 lis_esolver_set_option

```
C       int lis_esolver_set_option(char *text, LIS_ESOLVER esolver)
Fortran subroutine lis_esolver_set_option(character text, LIS_ESOLVER esolver,
         integer ierr)
```

Description

Set options for eigensolver

Input

text	Command line options

Output

esolver	Eigensolver
ierr	Return code

Note

The table below shows the available command line options, where `-e {pi|1}` means `-e pi` or `-e 1` and `-emaxiter [1000]` indicates that `-emaxiter` defaults to 1,000.

Specifying Eigensolvers (Default: `-e pi`)

Method	Option	Auxiliary Option	
Power Iteration	`-e {pi|1}`		
Inverse Iteration	`-e {ii|2}`	`-i [bicg]`	Linear equation solver
Preconditioned Power Iteration	`-e {aii|3}`		
Conjugate Gradient	`-e {cg|4}`		
Lanczos Iteration	`-e {li|5}`	`-ss [2]`	Size of subspace
		`-m [0]`	Mode
Subspace Iteration	`-e {si|6}`	`-ss [2]`	Size of subspace
		`-m [0]`	Mode
Conjugate Residual	`-e {cr|7}`		

Specifying Preconditioners (Default: `-p ilu`)

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]`	Linear equation solver
		`-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(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 unsymmetric version (Matrix structure must be symmetric.)
		`-saamg_theta [0.05|0.12]`	Drop criteria $a_{ij}^2 \leq \theta^2 \lvert a_{ii}\rvert\lvert a_{jj}\rvert$ (symmetric or unsymmetric)
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		
`-emaxiter [1000]`	Maximum number of iterations	
`-etol [1.0e-12]`	Convergence criteria	
`-eprint [0]`	Display of the residual	
	`-eprint {none|0}`	None
	`-eprint {mem|1}`	Saves the residual history in memory
	`-eprint {out|2}`	Displays the residual history
	`-eprint {all|3}`	Saves the residual history and displays it on the screen
`-ie [ii]`	Inner eigensolver used in Lanczos Iteration or Subspace Iteration	
	`-ie {pi|1}`	Power Iteration (Subspace Iteration only)
	`-ie {ii|2}`	Inverse Iteration
`-shift [0.0]`	Amount of shift	
	`-ie {aii|3}`	Preconditioned Power Iteration
`-initx_ones [true]`	Behavior of the initial vector x_0	
	`-initx_ones {false|0}`	Given values
	`-initx_ones {true|1}`	All elements are set to 1.
`-omp_num_threads [t]`	Number of threads	
	`t` represents the maximum number of threads	

Precision (Default: `-eprecision double`)

Precision	Option	Auxiliary Option
DOUBLE	`-eprecision {double|0}`	
QUAD	`-eprecision {quad|1}`	

6.5.4 lis_esolver_set_optionC

```
C       int lis_esolver_set_optionC(LIS_ESOLVER esolver)
Fortran subroutine lis_esolver_set_optionC(LIS_ESOLVER esolver, integer ierr)
```

Description

Set options for eigensolver on command line

Input

None

Output

esolver	Eigensolver
ierr	Return code

6.5.5 lis_esolve

```
C       int lis_esolve(LIS_MATRIX A, LIS_VECTOR x,
         LIS_REAL evalue, LIS_ESOLVER esolver)
Fortran subroutine lis_esolve(LIS_MATRIX A, LIS_VECTOR x,
         LIS_REAL evalue, LIS_ESOLVER esolver, integer ierr)
```

Description

Solve eigenvalue problem $Ax = \lambda x$ with specified eigensolver

Input

A	Matrix
x	Initial vector
esolver	Eigensolver

Output

evalue	Eigenvalue of mode specified by `-m [0]` option
x	Associated eigenvector
esolver	Number of iterations, execution time, etc.
ierr	Return code (0)

6.5.6 lis_esolver_get_status

```
C       int lis_esolver_get_status(LIS_ESOLVER esolver, int *status)
Fortran subroutine lis_esolver_get_status(LIS_ESOLVER esolver, integer status,
         integer ierr)
```

Description

Get status from eigensolver

Input

esolver	Eigensolver

Output

status	Number of iterations
ierr	Return code

6.5.7 lis_esolver_get_iters

```
C       int lis_esolver_get_iters(LIS_ESOLVER esolver, int *iters)
Fortran subroutine lis_esolver_get_iters(LIS_ESOLVER esolver, integer iters,
         integer ierr)
```

Description

Get number of iterations from eigensolver

Input

esolver	Eigensolver

Output

iters	Number of iterations
ierr	Return code

6.5.8 lis_esolver_get_itersex

```
C       int lis_esolver_get_itersex(LIS_ESOLVER esolver, int *iters,
         int *iters_double, int *iters_quad)
Fortran subroutine lis_esolver_get_itersex(LIS_ESOLVER esolver, integer iters,
         integer iters_double, integer iters_quad, integer ierr)
```

Description

Get number of iterations from eigensolver

Input

esolver	Eigensolver

Output

iters	Number of iterations
iters_double	Number of double precision iterations
iters_quad	Number of quadruple precision iterations
ierr	Return code

6.5.9 lis_esolver_get_time

```
C       int lis_esolver_get_time(LIS_ESOLVER esolver, double *times, double *itimes,
         double *ptimes)
Fortran subroutine lis_esolver_get_time(LIS_ESOLVER esolver, real*8 times,
         real*8 itimes, real*8 ptimes, integer ierr)
```

Description

Get execution time from eigensolver

Input

esolver	Eigensolver

Output

times	Time in seconds for execution
times	Time in seconds for iteration
times	Time in seconds for preconditioning
ierr	Return code

6.5.10 lis_esolver_get_timeex

```
C       int lis_esolver_get_timeex(LIS_ESOLVER esolver, double *times,
         double *itimes, double *ptimes, double *p_c_times, double *p_i_times)
Fortran subroutine lis_esolver_get_timeex(LIS_ESOLVER esolver, real*8 times,
         real*8 itimes, real*8 ptimes, real*8 p_c_times, real*8 p_i_times,
         integer ierr)
```

Description

Get execution time from eigensolver

Input

esolver	Eigensolver

Output

times	Total time in seconds
itimes	Time in seconds for iteration
ptimes	Time in seconds for preconditioning
p_c_times	Time in seconds for creating preconditioner
p_i_times	Time in seconds for iteration in preconditioner
ierr	Return code

6.5.11 lis_esolver_get_residualnorm

```
C       int lis_esolver_get_residualnorm(LIS_ESOLVER esolver, LIS_REAL *residual)
Fortran subroutine lis_esolver_get_residualnorm(LIS_ESOLVER esolver,
         LIS_REAL residual, integer ierr)
```

Description

Get 2-norm of $(\lambda x - Ax)/\lambda$ from eigensolver

Input

esolver	Eigensolver

Output

residual	2-norm of $(\lambda x - Ax)/\lambda$
ierr	Return code

6.5.12 lis_esolver_get_rhistory

```
C       int lis_esolver_get_rhistory(VECTOR v)
Fortran subroutine lis_esolver_get_rhistory(LIS_VECTOR v, integer ierr)
```

Description

Store residual norm history of eigensolver

Input

None

Output

v	Vector
ierr	Return code

Note

The vector `v` must be created in advance with the function `lis_vector_create`. When the vector `v` is shorter than the residual history, it stores the residual history in order to the vector `v`.

6.5.13 lis_esolver_get_evalues

```
C       int lis_esolver_get_evalues(LIS_ESOLVER esolver, LIS_VECTOR v)
Fortran subroutine lis_esolver_get_evalues(LIS_ESOLVER esolver,
         LIS_VECTOR v, integer ierr)
```

Description

Store eigenvalues in vector

Input

esolver	Eigensolver

Output

v	Vector which stores eigenvalues
ierr	Return code

Note

The vector `v` must be created in advance with the function `lis_vector_create`.

6.5.14 lis_esolver_get_evectors

```
C       int lis_esolver_get_evectors(LIS_ESOLVER esolver, LIS_MATRIX A)
Fortran subroutine lis_esolver_get_evectors(LIS_ESOLVER esolver,
         LIS_MATRIX A, integer ierr)
```

Description

Store eigenvectors in matrix

Input

esolver	Eigensolver

Output

A	Matrix in CRS format which stores eigenvectors
ierr	Return code

Note

The matrix `A` must be created in advance with the function `lis_matrix_create`.

6.5.15 lis_esolver_get_esolver

```
C       int lis_esolver_get_esolver(LIS_ESOLVER esolver, int *nesol)
Fortran subroutine lis_esolver_get_esolver(LIS_ESOLVER esolver, integer nesol,
         integer ierr)
```

Description

Get eigensolver number from eigensolver

Input

esolver	Eigensolver

Output

nesol	Eigensolver number
ierr	Return code

Note

The number of the eigensolver is as follows:

Method	Number
Power Iteration	1
Inverse Iteration	2
Approximate Inverse Iteration	3
Conjugate Gradient	4
Lanczos Iteration	5
Subspace Iteration	6
Conjugate Residual	7

6.5.16 lis_get_esolvername

```
C       int lis_get_esolvername(int nesol, char *ename)
Fortran subroutine lis_get_esolvername(integer nesol, character ename, integer ierr)
```

Description

Get eigensolver name from eigensolver number

Input

nesol	Eigensolver number

Output

name	Eigensolver name
ierr	Return code

6.6 File I/O

6.6.1 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)
```

Description
Read matrix and vector from file

Input

filename	Source file

Output

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

Note
The supported file formats are shown below:

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

6.6.2 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)
```

Description
Read vector from file

Input

filename	Source file

Output

v	Vector
ierr	Return code

Note
The following formats are supported:

- PLAIN format
- MM format

6.6.3 lis_input_matrix

```
C       int lis_input_matrix(LIS_MATRIX A, char *filename)
Fortran subroutine lis_input_matrix(LIS_MATRIX A, LIS_VECTOR x,
         character filename, integer ierr)
```

Description

Read matrix from file

Input

filename	Source file

Output

A	Matrix in specified storage format
x	Solution
ierr	Return code

Note

The supported file formats are shown below:

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

6.6.4 lis_output

```
C       int lis_output(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x, int format,
         char *filename)
Fortran subroutine lis_output(LIS_MATRIX A, LIS_VECTOR b, LIS_VECTOR x,
         integer format, character path, integer ierr)
```

Description

Write matrix and vector into file

Input

A	Matrix
b	Right hand side vector (If no vector is written into a file, then `NULL` must be input.)
x	Solution (If no vector is written into a file, then `NULL` must be input.)
format	File format
	LIS_FMT_MM MatrixMarket format
filename	Destination file

Output

ierr	Return code

6.6.5 lis_output_vector

```
C       int lis_output_vector(LIS_VECTOR v, int format, char *filename)
Fortran subroutine lis_output_vector(LIS_VECTOR v, integer format,
         character filename, integer ierr)
```

Description

Write vector into file

Input

v	Vector	
format	File format	
	LIS_FMT_PLAIN	PLAIN format
	LIS_FMT_MM	MM format
	LIS_FMT_LIS	Lis format(ASCII)
filename	Destination file	

Output

ierr	Return code

6.6.6 lis_output_matrix

```
C       int lis_output_matrix(LIS_MATRIX A, int format, char *filename)
Fortran subroutine lis_output_matrix(LIS_MATRIX A, integer format, character path,
         integer ierr)
```

Description

Write matrix into file

Input

A	Matrix	
format	File format	
	LIS_FMT_MM	MatrixMarket format
filename	Destination file	

Output

ierr	Return code

6.7 Other Functions

6.7.1 lis_initialize

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

Description
Initialize execution environment

Input

argc	Number of command line arguments
argv	Command line argument

Output

ierr	Return code

6.7.2 lis_finalize

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

Description
Finalize execution environment

Input

None

Output

ierr	Return code

6.7.3 lis_wtime

```
C       double lis_wtime()
Fortran function lis_wtime()
```

Description
Measure elapsed time

Input

None

Output
Elapsed time in seconds from given point is returned as double number

Note
To measure the processing time, call `lis_wtime` to get the starting time, call it again to get the ending time, and calculate 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. Evaluation of Parallel Aggregate Creation Orders : Smoothed Aggregation Algebraic Multigrid Method. High Performance Computational Science And Engineering, pp. 99–122, Springer, 2005.

[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. Sonnerveld and M. B. van Gijzen. IDR(s): a family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. SIAM J. Sci. Comput., Vol. 31, Issue 2, pp. 1035–1062, 2008.

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

[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] A. V. Knyazev. Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. SIAM J. Sci. Comput., Vol. 23, No. 2, pp. 517–541, 2001.

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

[20] E. Suetomi and H. Sekimoto. Conjugate gradient like methods and their application to eigenvalue problems for neutron diusion equation. Annals of Nuclear Energy, Vol. 18, No. 4, pp. 205–227, 1991.

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

[22] D. H. Bailey. High-Precision Floating-Point Arithmetic in Scientific Computation. Computing in Science and Engineering, Volume 7, Issue 3, pp. 54–61, IEEE, 2005.

[23] Intel Fortran Compiler User's Guide Vol I.

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

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

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

[27] 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.

[28] S. Balay, et al. PETSc users manual. Technical Report ANL-95/11, Argonne National Laboratory, August 2004.

[29] R. S. Tuminaro, et al. Official Aztec user's guide, version 2.1. Technical Report SAND99-8801J, Sandia National Laboratories, November 1999.

[30] R. B. Lehoucq, D. C. Sorensen, and C. Yang. ARPACK Users' Guide: Solution of Large-scale Eigenvalue Problems with implicitly-restarted Arnoldi Methods. SIAM, 1998.

[31] R. Bramley and X. Wang. SPLIB: A library of iterative methods for sparse linear system. Technical report, Indiana University–Bloomington, 1995.

[32] Matrix Market. http://math.nist.gov/MatrixMarket.

A File Formats

This section describes the file formats available for the library.

A.1 Extended MatrixMarket Format

The MatrixMarket format[32] is not designed to store vector data. For this library, it has been extended to handle vector data. Assume that the number of the nonzero elements for the 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)                                 <-+    nonzero elements (0 or 1) (0 or 1)
I2 J2 A(I2,J2)                                   | Row and column number values
 . . .                                           | The index is one origin
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 extended MatrixMarket file for the matrix A and the vector b in Equation (A.1) is structured as follows:

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

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

A.2 Harwell-Boeing Format

The Harwell-Boeing format inputs and outputs the matrix in the CCS storage format. Assume that the array `value` stores the values of the nonzero elements of the matrix A, the array `index` stores the row indices of the nonzero 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 side vectors
Line 3 (A3,11X,4I14)
   1 -  3 Matrix type
            Col.1: R Real matrix
                   C Complex matrix (Not supported)
                   P Pattern only (Not supported)
            Col.2: S Symmetric
                   U Unsymmetric
                   H Hermitian (Not supported)
                   Z Skew symmetric (Not supported)
                   R Rectangular (Not supported)
            Col.3: A Assembled
                   E Elemental matrices (Not supported)
   4 - 14 Blank space
  15 - 28 Number of rows
  29 - 42 Number of columns
  43 - 56 Number of nonzero 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 side vectors
Line 5 (A3,11X,2I14) Only presents if there are right hand side vectors
   1      Right hand side vector type
            F for full storage
            M for same format as matrix (Not supported)
   2      G if a starting vector is supplied
   3      X if an exact solution is supplied
   4 - 14 Blank space
  15 - 28 Number of right hand side vectors
  29 - 42 Number of nonzero elements
```

The Harwell-Boeing file for the matrix A and the 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             0
      1      3      6      9
     1     2     1     2     3     2     3     4     3     4
  2.00000000000000000E+00  1.00000000000000000E+00  1.00000000000000000E+00
  2.00000000000000000E+00  1.00000000000000000E+00  1.00000000000000000E+00
  2.00000000000000000E+00  1.00000000000000000E+00  1.00000000000000000E+00
  2.00000000000000000E+00
  0.00000000000000000E+00  1.00000000000000000E+00  2.00000000000000000E+00
  3.00000000000000000E+00
```

A.3 Extended MatrixMarket Format for Vectors

The MatrixMarket format[32] has been extended to store vector data. Assume that the 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 one origin
IN B(IN)                                       <-+
```

The extended MatrixMarket file for the 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 for vectors is designed to write vector values in order. Assume that the 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 the vector b in Equation (A.1), the file is structured as follows:

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