反復解法ライブラリーLisの紹介

小武守　恒
（JST CREST／東京大学）
kota@is.s.u-tokyo.ac.jp

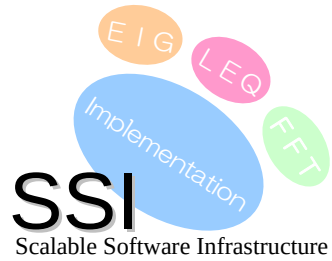
本日の内容

- Lisの紹介
  - 特徴
  - 他の反復解法ライブラリとの比較
- Lisの利用方法
  - インストール
  - 行列、ベクトル等の作り方
  - ユーザープログラムのコンパイル
  - センターの計算機上でのデモ
- Lisの性能結果

Lisとは？

- Lis (a Library of Iterative Solvers for linear systems)
- 大規模実疎行列係数の線型方程式 Ax = b に対する反復法ライブラリ
- C言語とFortran 90で記述

Lisの生い立ち



- 大規模シミュレーション向け基盤ソフトウェアの開発
  - JST CRESTの領域研究課題「シミュレーション技術の革新と実用化基盤の構築」の一プロジェクト
  - 2002年11月～2008年3月まで
  - 2004年10月からLisの開発開始
  - 2005年9月20日　Lis 1.0.0リリース
  - 2007年10月末　Lis 1.1.0リリース予定

Lisの特徴(1)

- 20通りの反復解法，10通りの前処理が組み合わせて利用できる

反復解法		前処理
CG	CR	Jacobi
BiCG	BiCR	SSOR
CGS	CRS	ILU(k)
BiCGSTAB	BiCRSTAB	Hybrid
GPBiCG	GPBiCR	I+S
BiCGSafe	BiCRSafe	SAINV
TFQMR	BiCGSTAB(l)	SA-AMG
Jacobi	GMRES(m)	ILUT
Gauss-Seidel	FGMRES(m)	ILUC
SOR	Orthomin(m)	additive schwarz

Lisの特徴(2)

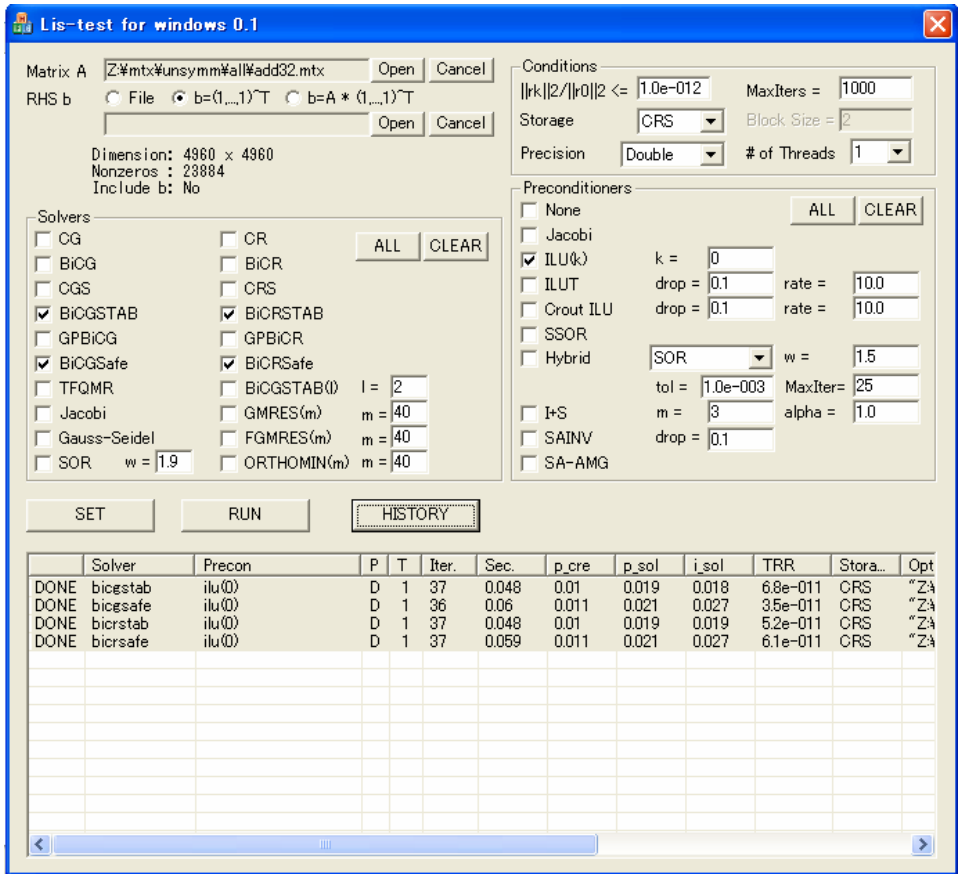
- 11通りの疎行列格納形式が利用できる

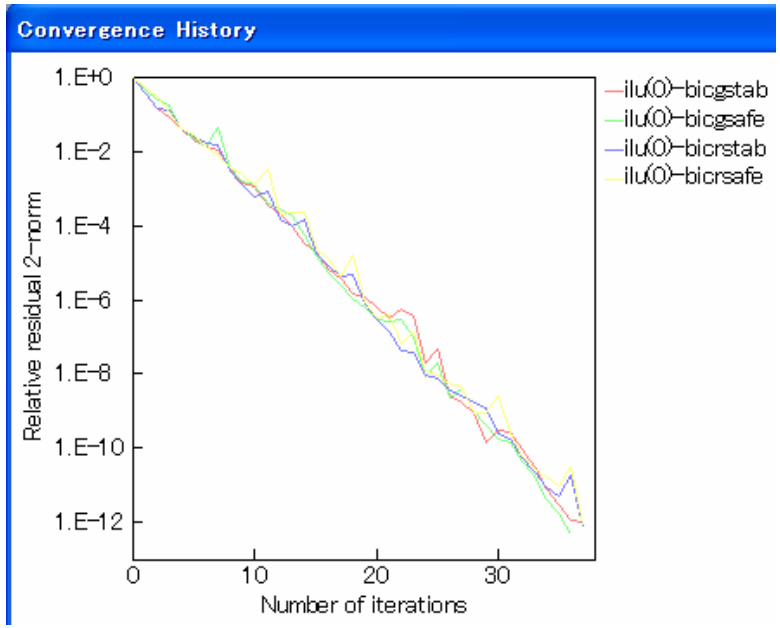
Point	Compressed Row Storage	(CRS)
	Compressed Column Storage	(CCS)
	Modified Compressed Sparse Row	(MSR)
	Diagonal	(DIA)
	Ellpack-Itpack generalized diagonal	(ELL)
	Jagged Diagonal	(JDS)
	Coordinate	(COO)
	Dense	(DNS)
Block	Block Sparse Row	(BSR)
	Block Sparse Column	(BSC)
	Variable Block Row	(VBR)

Lisの特徴(3)

- 逐次，OpenMP共有メモリ並列，MPI単独，あるいはOpenMP＋MPIのハイブリッド分散メモリ並列で動作可能
- 逐次と並列ともに共通のインタフェースで処理できる
- 4倍精度演算にも対応

Lis-test for Windows

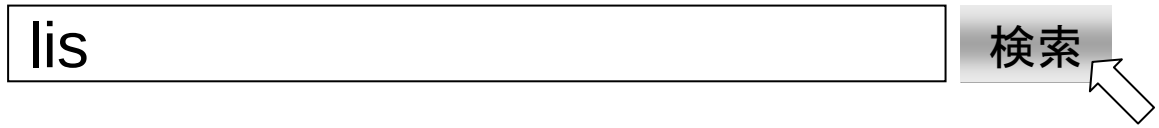




- いろいろな組み合わせが手軽に実行できる（結果はファイルに）
- 収束履歴グラフもボタン一つで表示できる
- GUIと実行形式の2つのファイル

Lisの取得

- http://www.ssisc.org/lis へアクセス



- 314件のダウンロード（ 2007/10/15現在）
  - 2006年　168件　　14件／月
  - 2007年　146件　　14.6件／月

他の反復解法ライブラリ

- PETSc（The Portable Extensible Toolkit for Scientific Computation ）
  - 様々な反復法、前処理が利用可能
  - いくつかの格納形式が利用可能
  - 逐次、MPIで動作可能
  - 複素数に対応

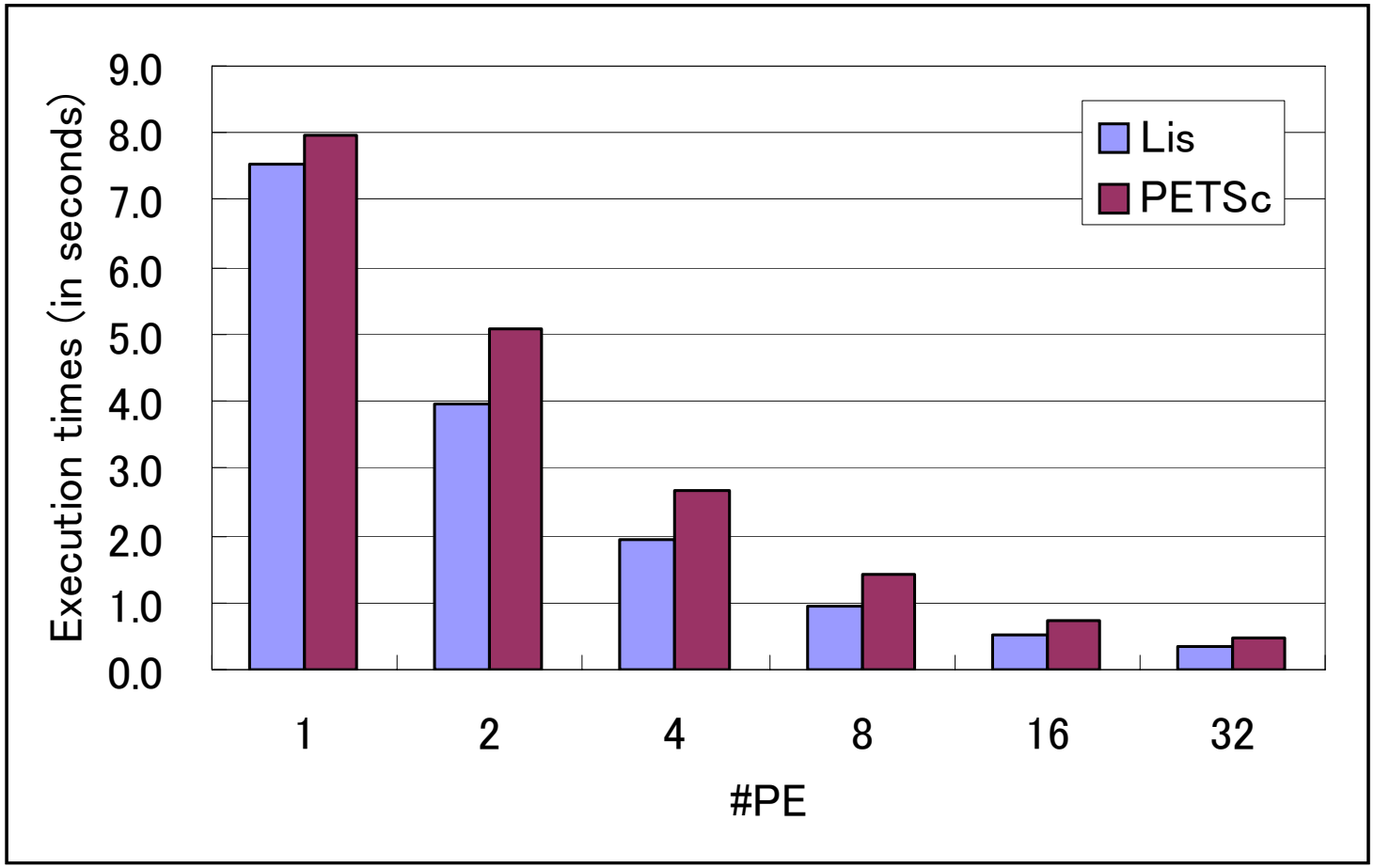
Lis vs PETSc（2.3.0）

	Lis	PETSc
反復解法	20	15
直接解法	×	○
前処理	10	10（外部+10）
演算精度	実のみ	実、複素
環境	逐次，MPI OpenMP MPI+OMP	逐次，MPI
API	C, FORTRAN	C,C++,FORTRAN
その他	**4倍精度**	非線形解法

性能比較

- SGI Altix 3700上で測定
  - Itanium2（1.3GHz）x32
  - Intel Compiler 9.1
  - PETSc：--with-vendor-compilers=intel --with-blas-lapack-dir=/opt/intel/mkl/8.0/ --CFLAGS=-O3
- 3次元ポアソン方程式を有限要素法で離散化
  - 次数：100万
  - 非零要素数：26,207,180
- CG法を50回反復

CG法を50反復した結果



- Lisのほうが32PEのとき27%高速

Lisのインストール

動作環境

- PC（Linux, Windows, Mac OS X）

- SGI Altix 3700
- Sun Fire 3800
- Cray XT3
- NEC SX6i
- 地球シミュレータ
- BlueGene
- FUJITSU PRIMEQUEST, PRIMERGY

必要なシステム

- Cコンパイラ（必須）
  - Intel C/C++ 7.0,8.0,9.0,9.1　IBM XL C 7.0
  - SUN WorkShop 6, ONE Studio 7, ONE Studio 11
  - GCC 3.3以降
- FORTRANコンパイラ（オプション）
  - FORTRAN APIを利用する場合　F77以上
  - SAAMG前処理を利用する場合　F90以上
  - Intel Fortran 8.1,9.0,9.1　IBM XL FORTRAN 9.1
  - SUN WorkShop 6, ONE Studio 7, ONE Studio 11
  - g77 3.3以降　gfortran 4.1(SAAMGでは×)　g95 0.91

Linuxでのビルド手順

1. ファイルの展開
   >gunzip -c lis-1.1.0.tar.gz | tar xvf –

2. `configure`スクリプトの実行
   >./configure

3. `make`の実行
   >make

4. インストール
   >make install

configureオプション

OpenMPを利用	--enable-omp
MPIを利用	--enable-mpi
FORTRAN APIを利用	--enable-fortran
SA-AMG前処理を利用	--enable-saamg
4倍精度演算を利用	--enable-quad

九大情報基盤センターでの手順（PRIMEQUEST，PRIMERGY）

- PRIMEQUESTの場合

  >./configure TARGET=fujitsu_pq　または

  >./configure

- PRIMERGYの場合

  >./configure TARGET=fujitsu_pg

実際のconfigureの値

PRIMEQUEST

CC=“fcc” F77="frt“ FC="frt"
MPICC="mpifcc“ MPIF77="mpifrt"
MPIFC="mpifrt“ MPIRUN="mpiexec"
CFLAGS="-O3“ FFLAGS="-O3 -Cpp"
FCFLAGS="-O3 -Cpp -Am"
ac_cv_sizeof_void_p=8
ax_f77_mangling="lower case,
underscore, no extra underscore"
MPIRUN="mpiexec"
MPINP="-n"
OMPFLAG="-KOMP"

PRIMERGY

CC=“fcc” F77="frt“ FC="frt"
MPICC="mpifcc“ MPIF77="mpifrt"
MPIFC="mpifrt“ MPIRUN="mpiexec"
CFLAGS="-O3“ FFLAGS="-O3 -Cpp"
FCFLAGS="-O3 -Cpp -Am"
ac_cv_sizeof_void_p=8
ax_f77_mangling="lower case,
underscore, no extra underscore"
MPIRUN="mpiexec"
MPINP="-n"
OMPFLAG="-KOMP"
cross_compiling="yes“
AMDEFS="-pg"

移植に伴う修正（1）

- CとFORTRANが混在する場合、Cのmain関数はMAIN__ に修正
  - configure --enable-saamg --enable-fortranの場合
  - configureでfccを検出し、USE_MAIN__を定義

```
#ifdef USE_MAIN__
        #define main MAIN__
#endif

int main(int argc, char* argv[])
{
  …
}
```

移植に伴う修正（2）

- switch文中にOpenMPのプラグマを挿入するとエラーとなる
  - #pragma omp: 飛込みや飛出しをもつ文は構造ブロックではありません。
  - caseとbreakの間に中カッコを挿入することで解決

```
switch(bn) {
  case 1:
  {
    #pragma omp parallel for
    for(i=0;i<n;i++) ….
  }
  break;
  case 2:
  {
    #pragma omp parallel for
    for(i=0;i<n;i++) ….
  }
  break;
}
```

移植に伴う制限

- 富士通製Cコンパイラ ではSSE2の組み込み関数をコンパイルする機能がない
  - ぜひとも機能追加を！
  - デフォルトでSSE2命令を利用するようコンパイルはしてくれる

Lisの利用方法

Ax = b を解くための基本操作

1. 初期化処理
2. 行列の作成
3. ベクトルの作成
4. ソルバー（反復解法、前処理等の情報を格納する構造体）の作成
5. 行列、ベクトルに値を代入
6. ソルバーに反復法、前処理等を設定
7. 求解
8. 終了処理

準備

- プログラムの先頭にinclude文を記述

```
C #include "lis.h"
F #include "lisf.h"
```

C

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

FORTRAN

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

初期化・終了処理(1)

- 初期化処理

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

  F `lis_initialize(integer ierr)`

  - MPIの初期化，コマンドライン引数の取得等を行う

- 終了処理

  C `lis_finalize()`

  F `lis_finalize(integer ierr)`

ベクトルの作成

- ベクトル $v = \begin{pmatrix} 0 & 1 & 2 & 3 \end{pmatrix}^T$ を作成するプログラム

C

```
 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);
 6:
 7: for(i=0;i<n;i++)
 8: {
 9:
lis_vector_set_value(LIS_INS_VA
LUE,i,(double)i,v);
10:  }
```

FORTRAN

```
 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-1),v,ierr)
10: enddo
```

ベクトルの宣言と作成

- 変数宣言

```
LIS_VECTOR    v;
```

- 作成

```
C  lis_vector_create(LIS_Comm comm,
   LIS_VECTOR *vec)
F lis_vector_create(LIS_Comm comm,
   LIS_VECTOR vec, integer ierr)
```

  - commにはMPIコミュニケータを指定
  - 逐次、OpenMPの場合はcommの値は無視される

ベクトルのサイズ（1）

- サイズの設定

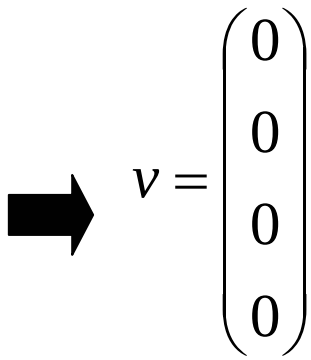
  C `lis_vector_set_size(LIS_VECTOR vec, int local_n, int global_n)`

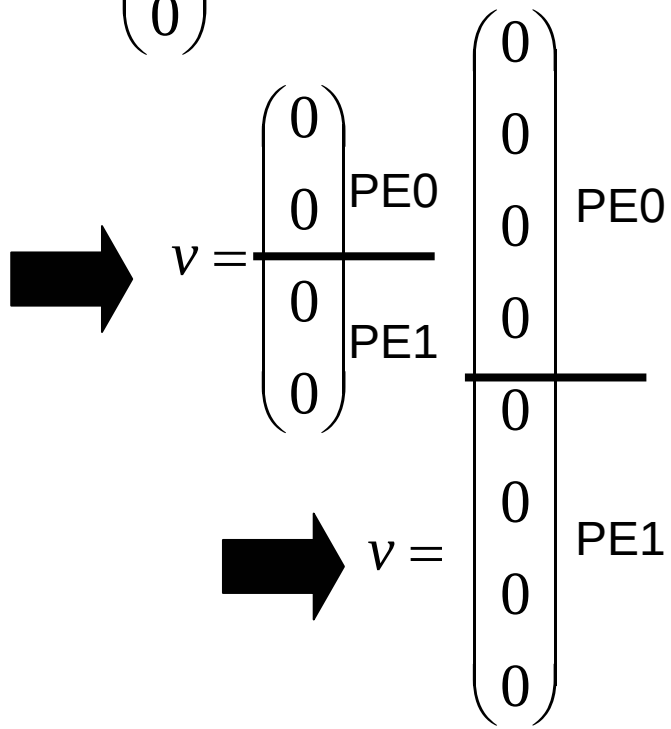
  F `lis_vector_set_size(LIS_VECTOR vec, integer local_n, integer global_n, integer ierr)`

  - `local_n` か `global_n` のどちらか一方

ベクトルのサイズ（2）

- 逐次、OpenMPの場合
  - `local_n = global_n`
  - lis_vector_set_size(v,4,0)
    lis_vector_set_size(v,0,4)
- MPIの場合（PE数は2）
  - lis_vector_set_size(v,0,4)
    全体ベクトルが次数
    4のベクトルを作成
  - lis_vector_set_size(v,4,0)
    各プロセッサに次数4の
    部分ベクトルを作成





ベクトルの要素の代入

- 要素の代入

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

  - MPIの場合、部分ベクトルのi行目ではなく全体ベクトルのi行目を指定
  - flag　LIS_INS_VALUE 挿入：v(i) = value
    LIS_ADD_VALUE 加算代入：v(i) = v(i) + value

ベクトルの複製

- 複製

```
C lis_vector_duplicate(LIS_VECTOR vin,
   LIS_VECTOR *vout)
F lis_vector_duplicate(LIS_VECTOR vin,
   LIS_VECTOR vout, integer ierr)
```

  - vinと同じ情報を持つベクトルを作成
  - ベクトルの値はコピーされず、領域のみ確保
  - vinにはLIS_MATRIXまたはLIS_VECTORが指定可能

ベクトルの破棄

- 破棄

```
C lis_vector_destroy(LIS_VECTOR v)
F lis_vector_destroy(LIS_VECTOR vec,
  integer ierr)
```

目的の格納形式で行列を作成

① 行番号、列番号、値を与えてライブラリ側で自動生成

② ユーザが目的の格納形式に必要な配列を用意する（FORTRANは未対応）

③ ファイルから行列データを読み込む

行番号、列番号、値を与えて ライブラリ側で自動生成

1. 変数宣言
2. 行列の作成
3. 行列のサイズ設定
4. 行列のサイズ取得
5. 行列の要素を格納する領域を確保
6. 要素の代入
7. 行列格納形式の設定
8. 行列の組立て

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

ユーザが目的の格納形式に必要な配列を用意する(FORTRANは未対応)

1. 変数宣言
2. 行列の作成
3. 行列のサイズ設定
4. 行列の要素を格納する領域を確保
5. 要素の代入
6. 配列を行列に関連付け
7. 行列の組立て

```
 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_create(LIS_COMM_WORLD,&A);
 7: lis_matrix_set_size(A,n,0);
 8: lis_matrix_malloc_crs
                (n,nnz,&ptr,&index,&value);
 9: lis_matrix_get_range(A,&is,&ie);
10: for(i=is;i<ie;i++) {
11:   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);
```

ファイルから読み込む

1. 変数宣言
2. 行列の作成
3. ベクトルの作成
4. 行列格納形式の設定
5. ファイルからの読み込み

```
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");
```

```
%%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
```

行列の宣言と作成

- 変数宣言

```
LIS_MATRIX    A;
```

- 作成

```
C lis_matrix_create(LIS_Comm comm,
   LIS_MATRIX *A)
F lis_matrix_create(LIS_Comm comm,
   LIS_MATRIX A, integer ierr)
```

  - commにはMPIコミュニケータを指定
  - 逐次、OpenMPの場合はcommの値は無視される
  - LIS_COMM_WORLD = MPI_COMM_WORLD

行列のサイズ

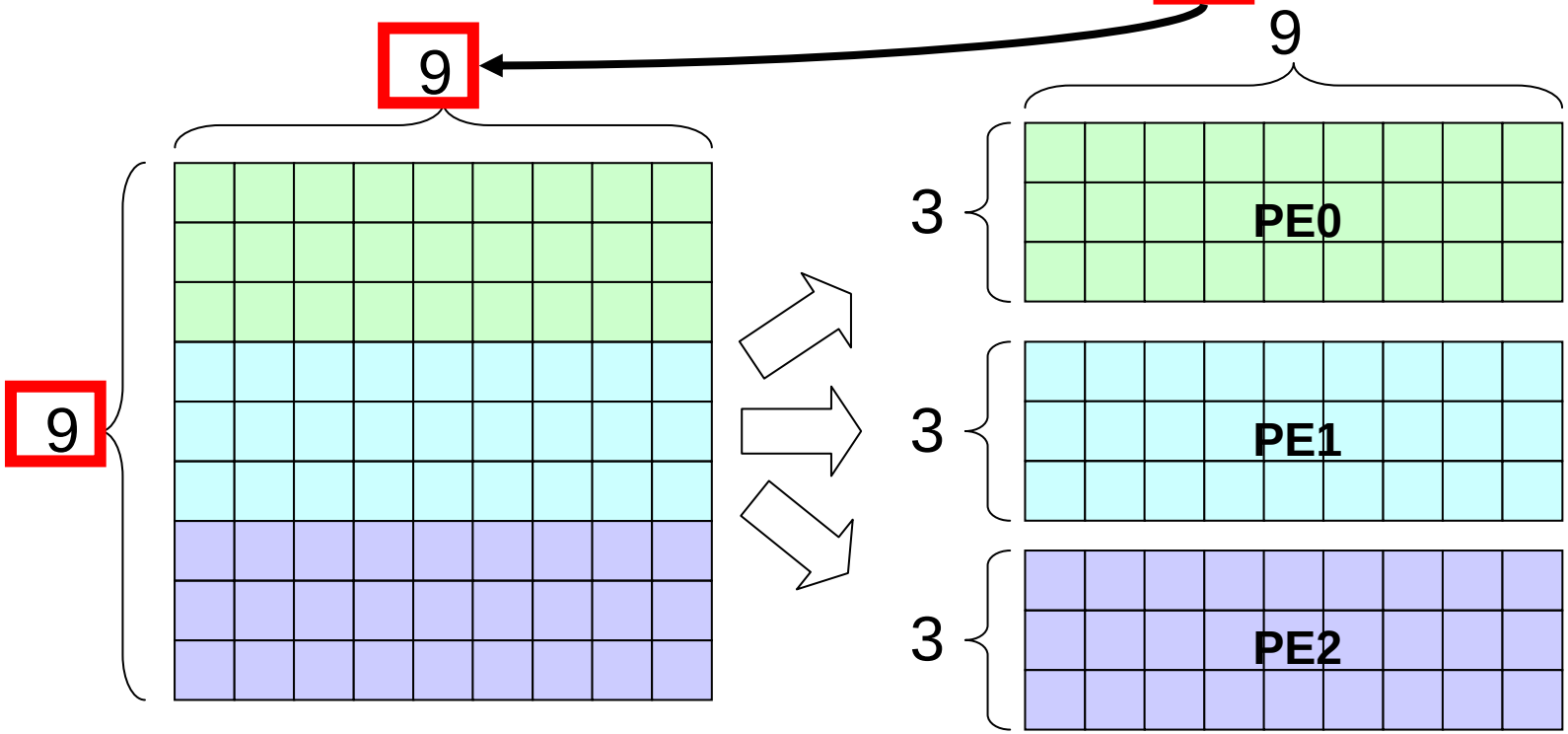
- サイズの設定

```
C lis_matrix_set_size(LIS_MATRIX A,
   int local_n, int global_n)
F lis_matrix_set_size(LIS_MATRIX A,
   integer local_n,integer global_n,
   integer ierr)
```

  - local_n か global_n のどちらか一方を与える
  - MPI環境ではサイズの設定の方法は2通り
  - 逐次、OpenMP環境ではどちらも同じ

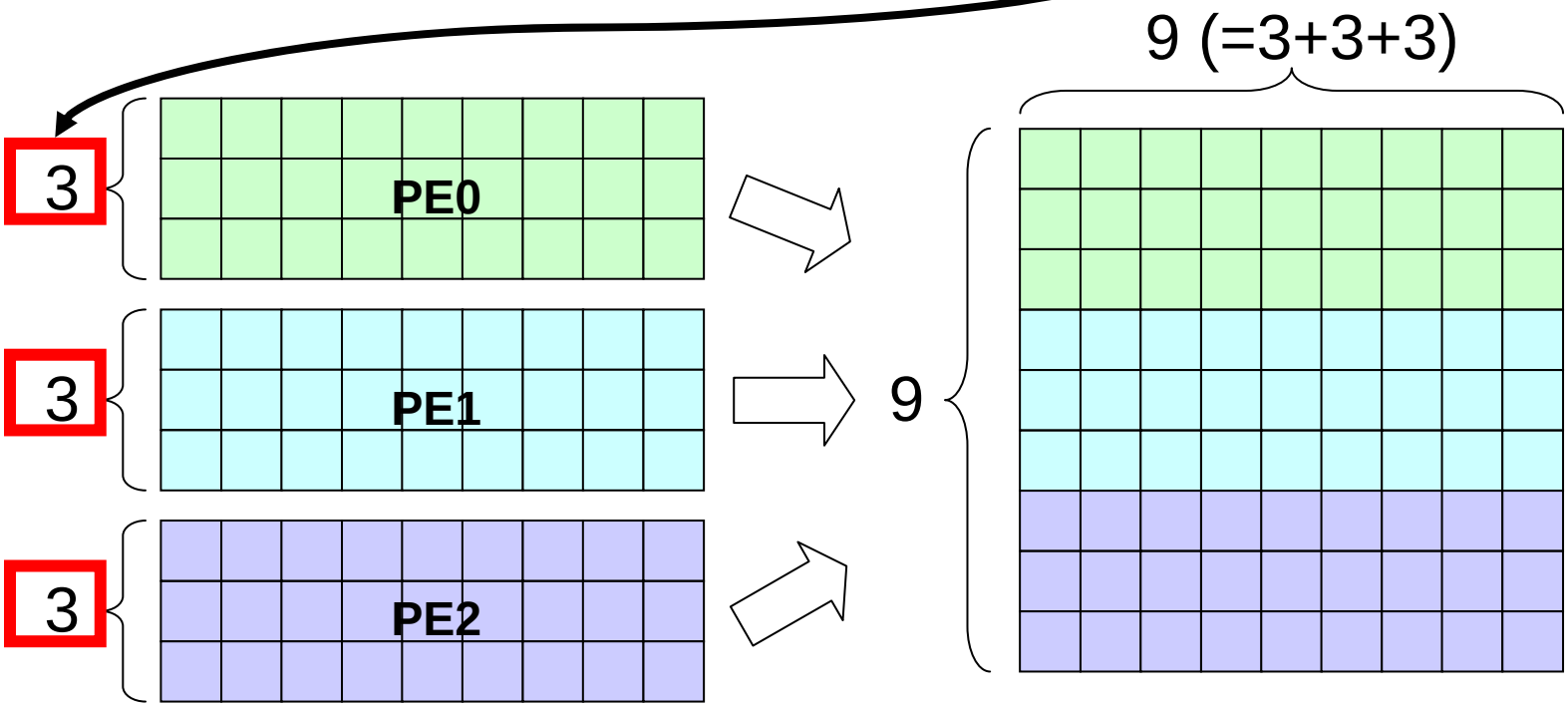
MPIでの行列サイズと分割数の決定(1)

- 行列サイズとPE数から分割数を決める
  - `lis_matrix_set_size(A,0,9)`



MPIでの行列サイズと分割数の決定(2)

- 分割数とPE数から行列サイズを決める
  - `lis_matrix_set_size(A, 3 ,0)`



行列の要素を格納する領域を確保

- 領域確保

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

  - lis_matrix_set_value で効率よく要素を代入できるように、あらかじめ領域を確保（ nnz_row = 10 ）
  - nnz_row または nnz のどちらか一方を指定
    - nnz_row:　平均非零要素数
    - nnz:　各行の非零要素数の配列

行列の要素の代入

- 行列 A の i 行 j 列目に要素を代入

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

  - MPIの場合、部分行列のi 行 j 列目ではなく全体行列の i 行 j 列目を指定する
  - flag　LIS_INS_VALUE 挿入：A(I,j) = value
    LIS_ADD_VALUE 加算代入：A(I,j) = A(I,j) + value

行列格納形式の設定

- 格納形式の設定

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

  - 行列作成時、A の matrix_type は LIS_MATRIX_CRS

行列の組立て

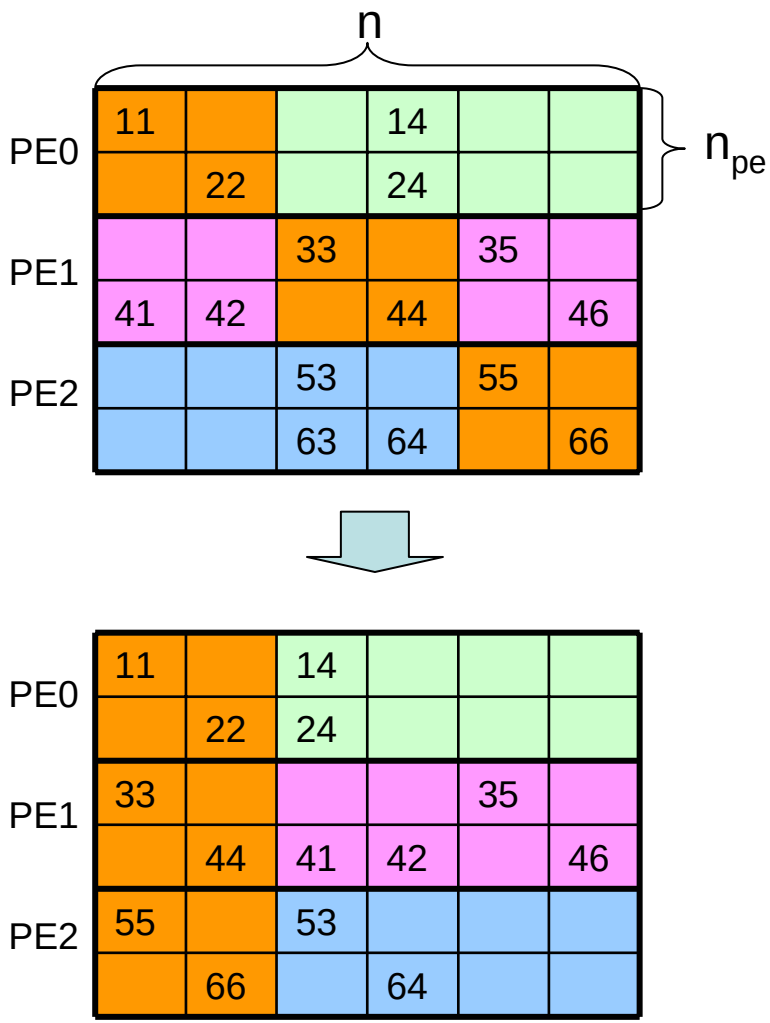
- 行列をライブラリで利用可能な状態にする

```
C lis_matrix_assemble(LIS_MATRIX A)
F lis_matrix_assemble(LIS_MATRIX A,
   integer ierr)
```

  - 行列に要素を代入した後、必ず呼び出す
  - lis_matrix_set_type で指定された格納形式に組み立てられる
  - MPIの場合、内部で全体行列の行または列番号から部分行列の番号へ変換と通信用のテーブルが作成される

MPI環境用行列への変換

- 各PEのn×n_{pe}のブロック対角部分が列の先頭になるように並べ替える
- その他の部分は非零な列を左につめる



ファイルからの入力

- ファイルからの入力

```
C lis_input(LIS_MATRIX A, LIS_VECTOR b,
   LIS_VECTOR x, char *filename)
F lis_input(LIS_MATRIX A, LIS_VECTOR b,
   LIS_VECTOR x, character filename,
   integer ierr)
```

  - 行列 A とベクトル b、xにファイルからデータを読み込む
  - 読み込むことができるファイルフォーマット
    - MatrixMarketフォーマット
    - Harwell-Boeing フォーマット（逐次、OpenMPのみ）
      - http://math.nist.gov/MatrixMarket/
      - http://www.cise.ufl.edu/research/sparse/matrices/

ソルバー

- ソルバーとは、反復解法や前処理、それらのパラメータを格納しておく構造体

- ソルバー関数
  - 作成　　lis_solver_create
  - 破棄　　lis_solver_destroy
  - オプション設定　lis_solver_set_option
  - 求解　　lis_solve

ソルバーの作成と破棄

- 作成

```
C lis_solver_create(LIS_SOLVER *solver)
F lis_solver_create(LIS_SOLVER solver,
    integer ierr)
```

- 破棄

```
C lis_solver_destroy(LIS_SOLVER solver)
F lis_solver_destroy(LIS_SOLVER solver,
    integer ierr)
```

ソルバーのオプションの設定（1）

- オプションの設定

```
C lis_solver_set_option(char *text,
  LIS_SOLVER solver)
F lis_solver_set_option(character text,
  LIS_SOLVER solver, integer ierr)
```

- コマンドラインからオプションを設定

```
C lis_solver_set_optionC(LIS_SOLVER
  solver)
F lis_solver_set_optionC(LIS_SOLVER
  solver, integer ierr)
```

ソルバーのオプションの設定（2）

- オプションの指定方法

  オプション　値

- 主なオプション
  - 反復解法の指定：-i [bicg]

    cg,bicg,cgs,bicgstab,bicgstabl,gpbicg,tfqmr

    orthomin,gmres,jacobi,gs,sor
  - 前処理の指定：-p [none]

    none,jacobi,ilu,ssor,hybrid,is,sainv,saamg,iluc

ソルバーのオプションの設定（３）

- 主なオプション
  - 最大反復回数：-maxiter [1000]
  - 収束判定基準：-tol [1.0e-12]
  - 演算精度：-precision [double]

    double,quad

求解

- 線型方程式 Ax = b を解く

```
C lis_solve(LIS_MATRIX A, LIS_VECTOR b,
   LIS_VECTOR x, LIS_SOLVER solver)
F lis_solve(LIS_MATRIX A, LIS_VECTOR b,
   LIS_VECTOR x, LIS_SOLVER solver,
   integer ierr)
```

  - ソルバーに与えられた出力は
    lis_solver_get_iters
    lis_solver_get_time
    lis_solver_get_residualnormで取得

求解までのサンプルプログラム

1. 初期化
2. 行列の作成
3. ベクトルの作成
4. ソルバーの作成
5. 行列、ベクトルに値を代入
6. ソルバーに反復法、前処理等を設定
7. 求解
8. 終了処理

```
 1: LIS_MATRIX    A;
 2: LIS_VECTOR    b,x;
 3: LIS_SOLVER    solver;
 4: int           iter;
 5: double        times,itime,ptime,pc,pi;
 6:
 7: lis_initialize(argc, argv);
 8: lis_matrix_create(LIS_COMM_WORLD,&A);
 9: lis_vector_create(LIS_COMM_WORLD,&b);
10: lis_vector_create(LIS_COMM_WORLD,&x);
11: lis_solver_create(&solver);
12: lis_input(A,b,x,argv[1]);
13: lis_vector_set_all(1.0,b);
14: lis_solver_set_optionC(solver);
15: lis_solve(A,b,x,solver);
16: lis_solver_get_iters(solver,&iter);
17: lis_solver_get_timeex(solver,&times,
    &itime,&ptime,&pc,&pi);
18: printf("iter = %d time = %e (p=%e
    i=%e)¥n",iter,times, ptimes, itimes);
19: lis_finalize();
```

ユーザプログラムのコンパイル

- 逐次、OpenMP

```
fcc -c [–KOMP]
   -I$(INSTALLDIR)/include test1.c
```

- MPI

```
mpifcc -c -DUSE_MPI
   -I$(INSTALLDIR)/include test1.c
```

  - PRIMERGYを利用する場合は –pg を追加

ユーザプログラムのリンク

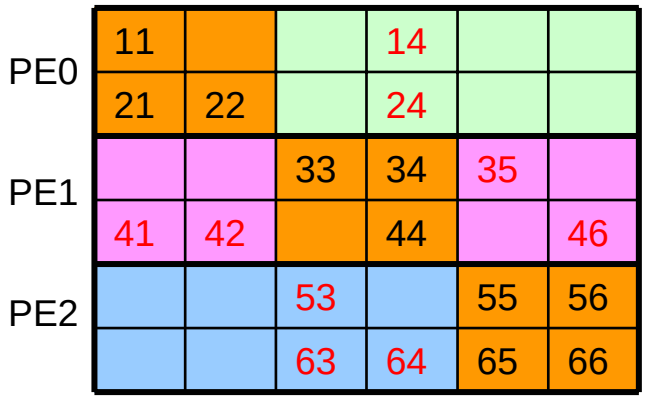
- 逐次、OpenMP
  - 通常時
    ```
    fcc [-KOMP] -o test1 test1.o –llis
    ```
  - SA-AMG前処理使用時
    ```
    frt [-KOMP] -o test1 test1.o –llis
    ```
- MPI
  - 通常時
    ```
    mpifcc [-KOMP] -o test1 test1.o –llis
    ```
  - SA-AMG前処理使用時
    ```
    mpifrt [-KOMP] -o test1 test1.o –llis
    ```
  - PRIMERGYを利用する場合は –pg を追加

Lis1.1.0の制限事項

- 前処理
  - Jacobi とSSOR 前処理以外が選択され行列A がCRS 形式以外のとき、前処理作成時にCRS 形式の行列A を作成する
  - BiCG 法を選択した場合、SA-AMG 前処理は非対応
  - OpenMP 環境ではSA-AMG 前処理は逐次、SAINV 前処理は前処理行列作成部分は逐次
- 4倍精度演算
  - 反復解法のJacobi、Gauss-Seidel、SOR 法は非対応
  - **HYBRID 前処理の内部反復解法の選択でJacobi、Gauss-Seidel、SOR は非対応**
  - **I+S とSA-AMG 前処理は非対応**

Lis1.1.0の注意事項

- 前処理
  - ILU, SSOR, SAINVに対して並列環境では通信が発生する要素は無視される



コンパイル･リンク･実行のデモ

デモ

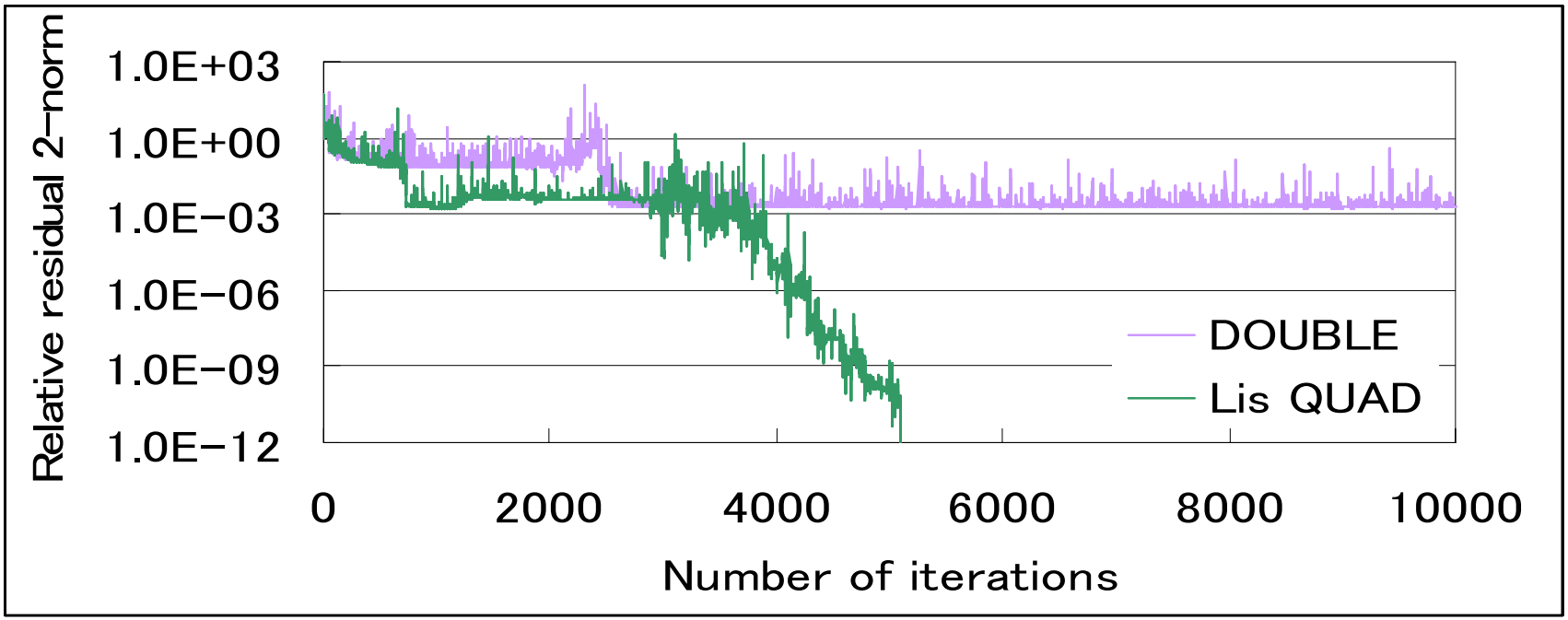
- デモのディレクトリ構造

```
demo
 ├test                    test1.c, test1f.F
 └local
   ├include        lis.f, lisf.h
   └lib                    (C only)      (C+FORTRAN)
                   liblisseq.a, liblisseqwf.a
                   liblisomp.a, liblisompwf.a
                   liblismpi.a, liblismpiwf.a
                   liblishyb.a, liblishybwf.a
```

Lisの4倍精度演算

4倍精度演算の収束性

- デバイスシミュレータで現れる行列（37,054次元）をILUC-BiCGSafe法で解く
- 倍精度では収束しないが、4倍精度なら収束



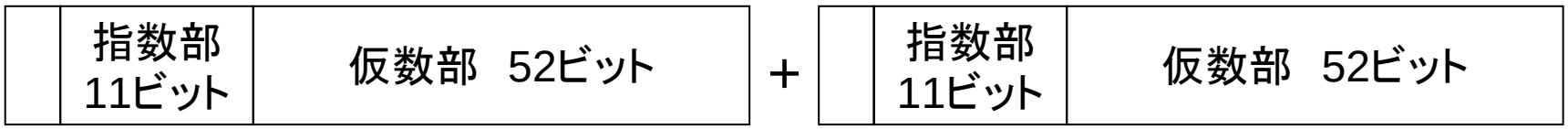
Lisでの4倍精度演算実装方針

- 同一インタフェース
  - 入力（係数行列A, 右辺ベクトル b, 初期ベクトル x_0 ）は倍精度、出力 解xは倍精度
  - 解法中の解 x, 補助ベクトル,スカラーは4倍精度
  - 前処理行列Mの生成部分は倍精度演算
    - 前処理行列Mは係数行列Aの近似
  - 反復中のMu = v の求解は4倍精度
- double-double精度演算を利用

double-double精度演算

- 倍精度浮動小数を2個用いて倍精度の四則演算の組み合わせで4倍精度を実現
  - FORTRAN REAL*16より高速
  - 仮数部がIEEE準拠より8ビット少ない
  - 有効桁数　　double-double精度　　約31桁
    IEEE準拠4倍精度　　約33桁

double-double精度



IEEE準拠の4倍精度

実装と高速化

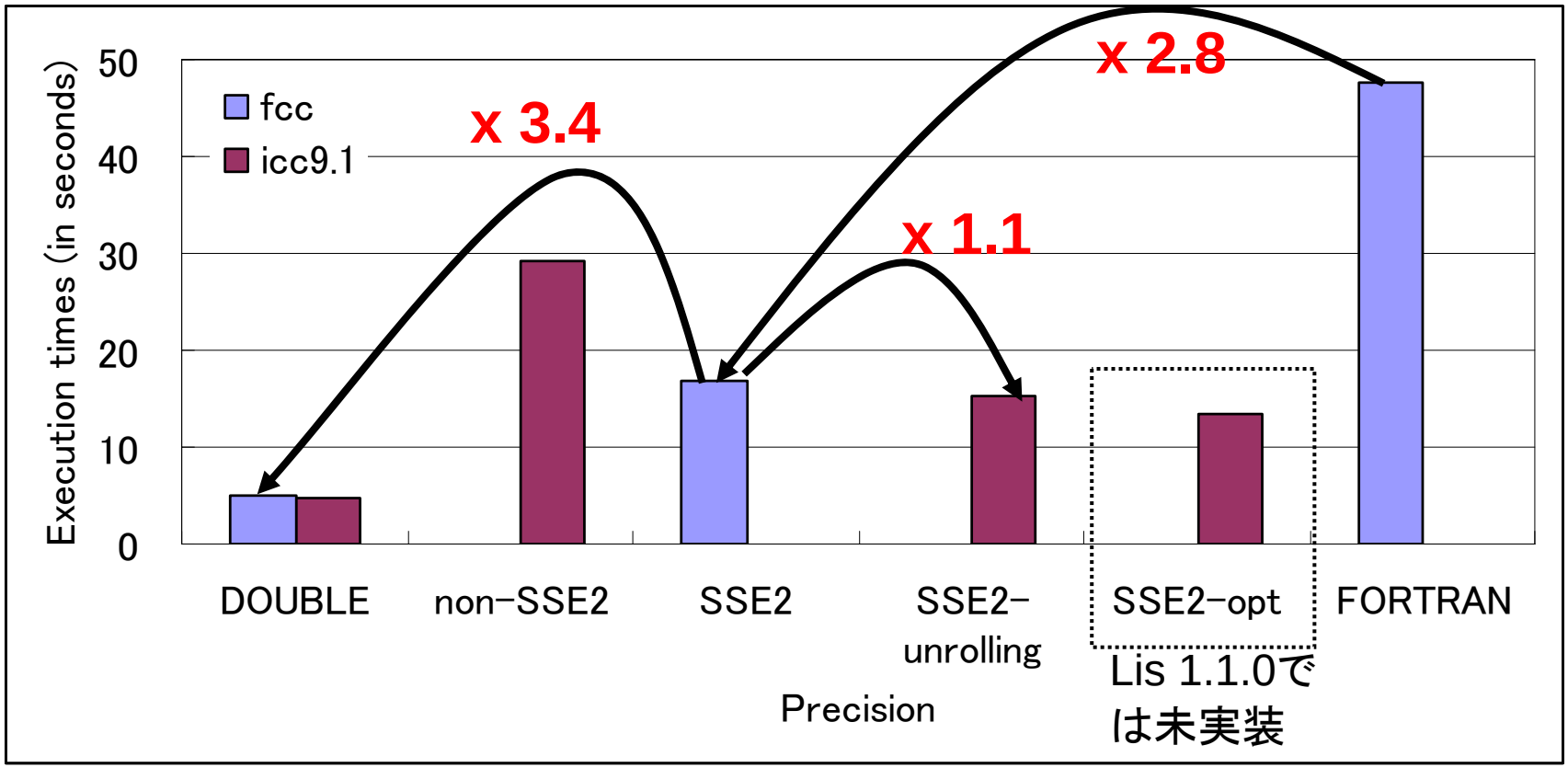
- 反復解法を4倍精度演算に置き換える
  - 行列ベクトル積（matvec）
  - ベクトルの内積（dot）
  - ベクトルおよびその実数倍の加減（axpy）
- matvec,dot,axpyの主な処理は積和演算
  - MULとADDの関数をまとめることでメモリストアを削減
- SSE2による高速化
- 2段のループアンローリング
  - すべてSSE2のpd命令で処理できる（理論的には2倍の高速化）

Lisの4倍精度演算の性能

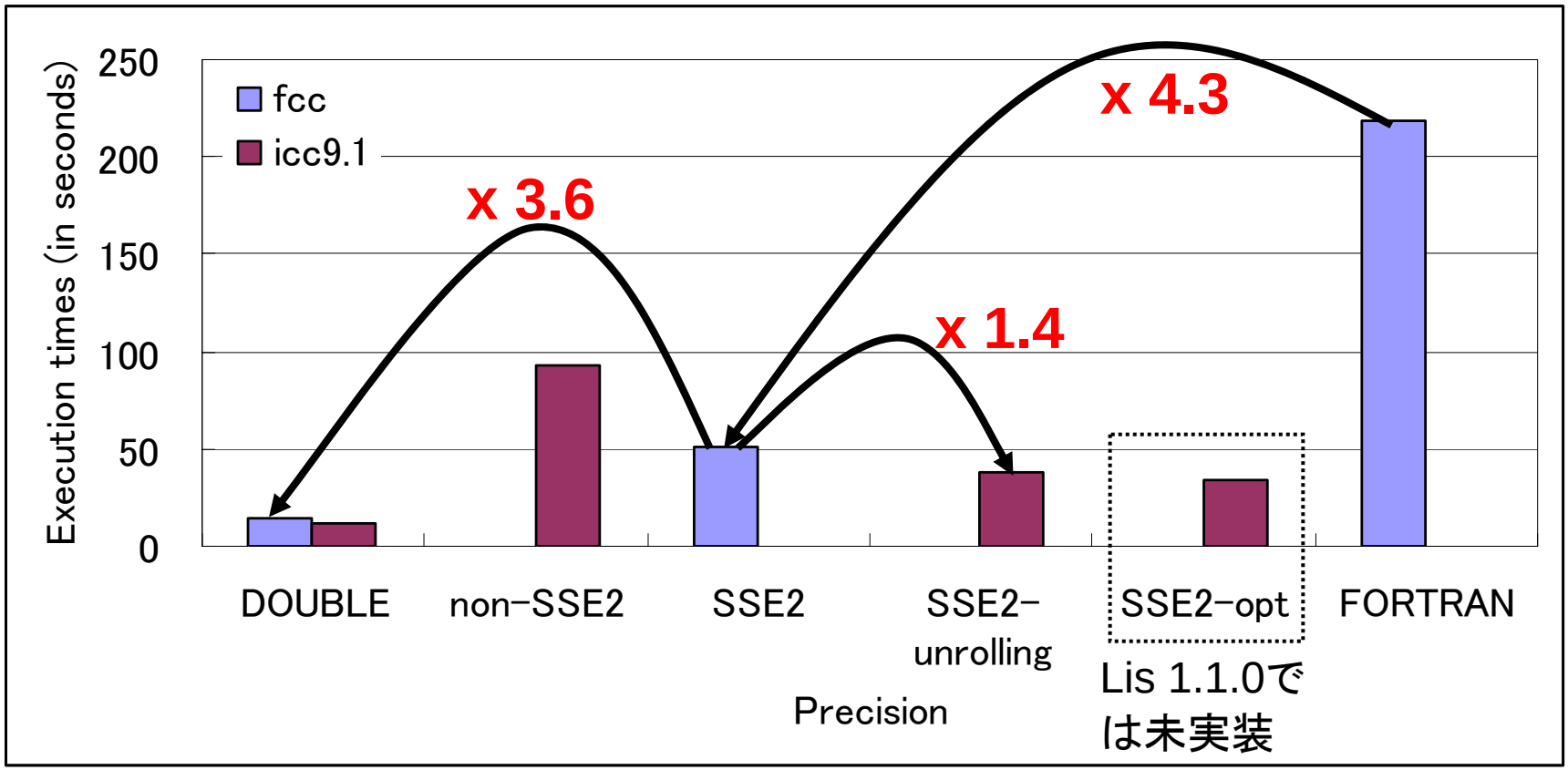
計算環境

- FUJITSU PRIMERGY RX200S3
  - 1ノード　Xeon 3.0GHz（2 Core）x 2
  - FUJITSU C Compiler
  - Intel C Compiler 9.1
    - SSE2組み込み関数利用のため

2次元ポアソン方程式を有限差分で離散化した行列(次数:100万)に対する実行時間 BiCG法50回反復



3次元ポアソン方程式を有限要素法で離散化した行列（次数：100万）に対する実行時間 BiCG法50回反復



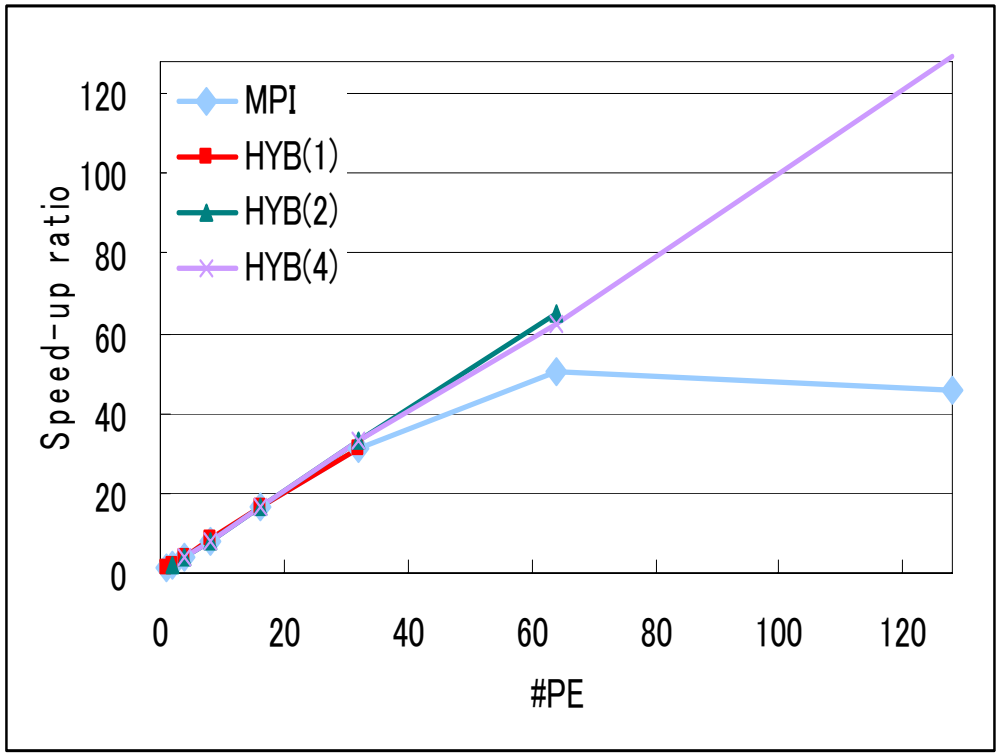
Lisの並列性能

実験条件

- 3次元ポアソン方程式を有限要素法で離散化
  - 次数：100万
  - 非零要素数：26,207,180
- 反復解法： CG
- 右辺ベクトルb= $(1,\ldots,1)^T$
- 初期ベクトル$x_0 = (0,\ldots,0)^T$
- 収束判定基準 $\|r_{k+1}\|_2 / \|r_0\|_2 \leq 10^{-12}$

Flat MPI vs MPI+OpenMP（倍精度）
CG法を50反復

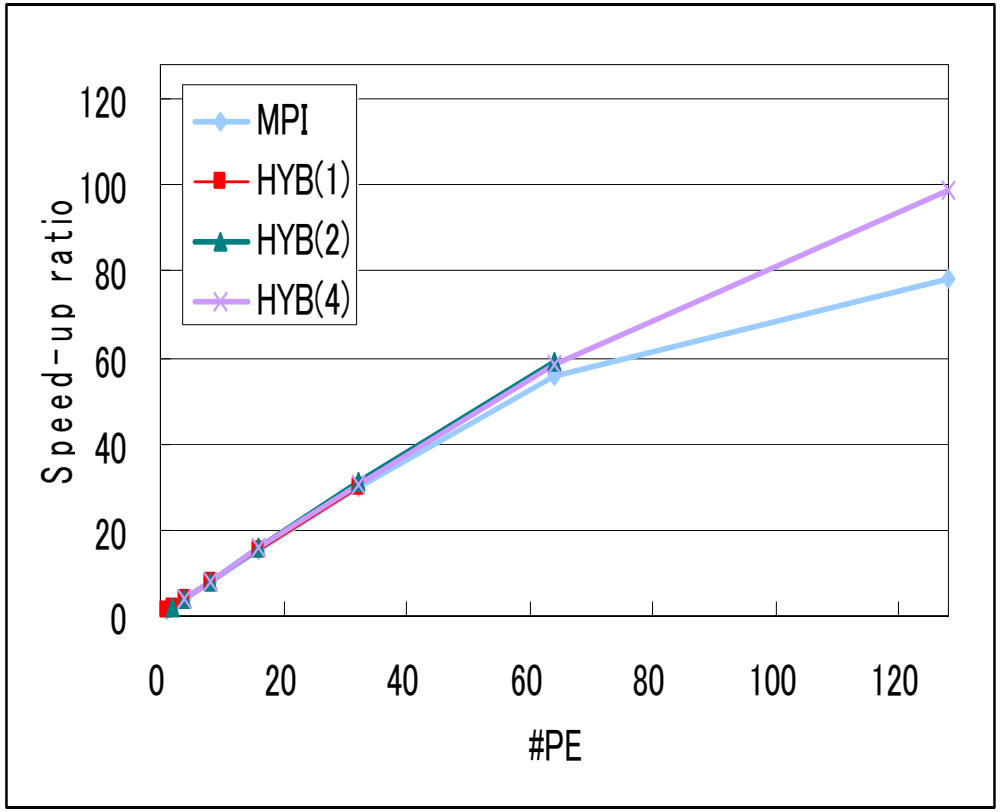
#PE	MPI	HYB (1)	HYB (2)	HYB (4)
1	7.16	7.74		
2	3.57	3.84	6.17	
4	1.76	1.88	3.08	3.88
8	0.87	0.93	1.50	1.88
16	0.44	0.47	0.74	0.92
32	0.23	0.25	0.37	0.47
64	0.14		0.19	0.25
128	0.16			0.12



- 64PEまではFlat MPIが高速
- 128PEではMPI+OpenMP(4)が高速
- 64PEまではOpenMPのスレッド数を増やすと性能低下

Flat MPI vs MPI+OpenMP（4倍精度）
CG法を50反復

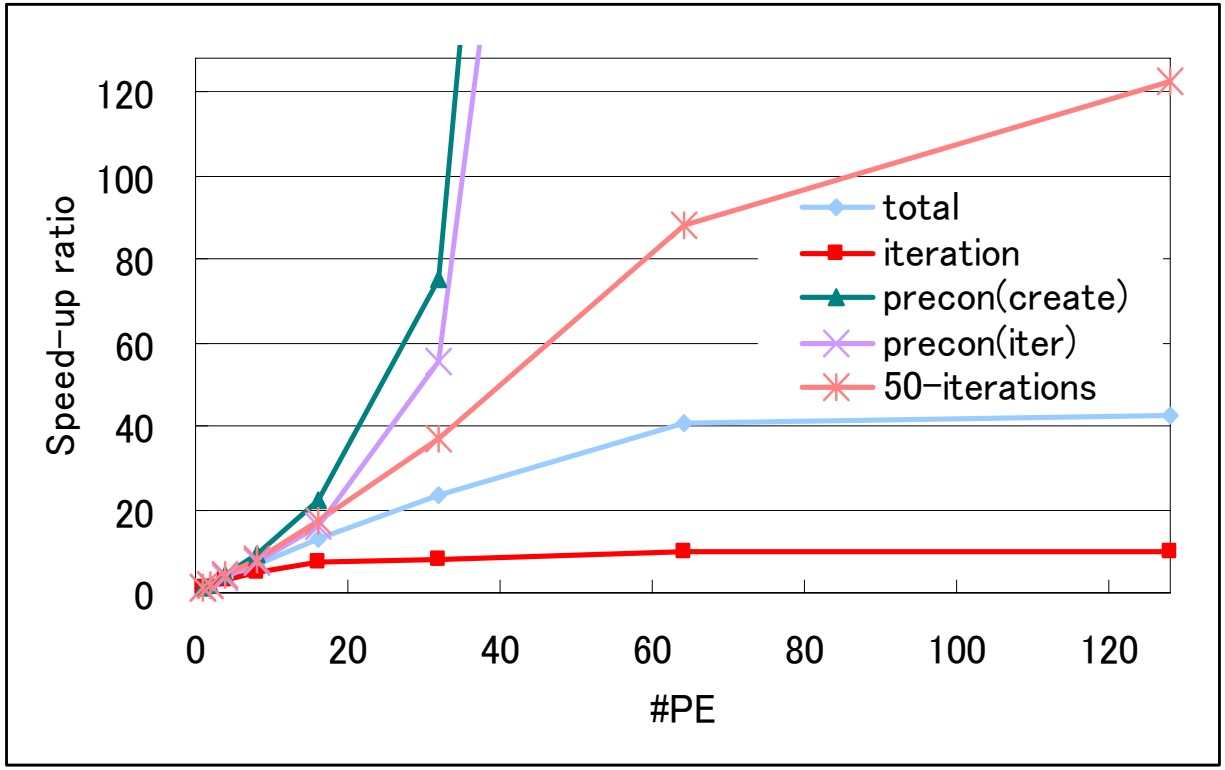
#PE	MPI	HYB (1)	HYB (2)	HYB (4)
1	27.43	28.38		
2	13.84	14.3	15.29	
4	6.97	7.23	7.70	7.92
8	3.51	3.64	3.87	3.96
16	1.78	1.84	1.94	2.00
32	0.92	0.95	0.99	1.03
64	0.50		0.52	0.54
128	0.35			0.32



- OpenMPのスレッド数を増やしても倍精度程の性能低下は発生していない
- 倍精度と比較して1PEで3.8倍、128PEで2.7倍の実行時間

局所ILU(0)前処理付CG法

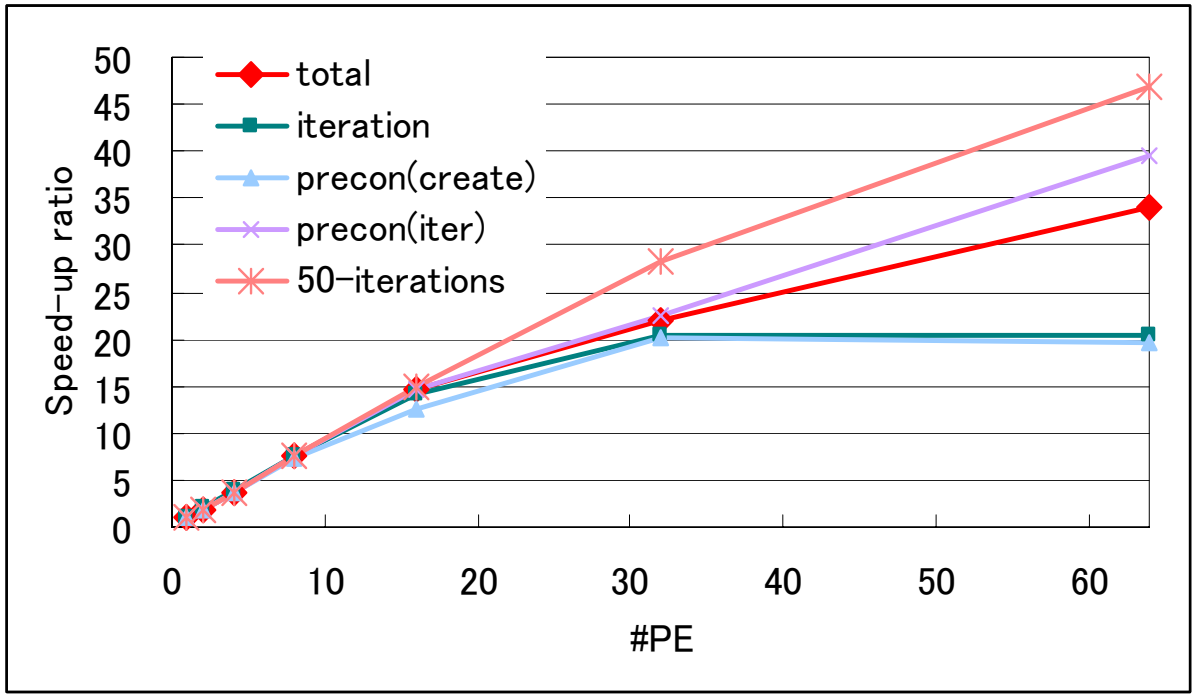
#PE	iter.	sec.
1	201	120.22
2	247	73.94
4	229	33.93
8	242	17.56
16	267	9.33
32	318	5.20
64	434	2.94
128	510	2.85



- 前処理は通信なしに処理できるため並列性は高い
- PE数増加にともない反復回数が大幅に増加

SA-AMG前処理付CG法

#PE	iter.	Sec.
1	133	103.18
2	136	55.07
4	137	28.12
8	130	13.39
16	135	7.04
32	174	4.67
64	185	3.02



- PE数増加による反復回数の増加は軽微
- 64PEでの速度向上率は34

まとめ

- Lisの4倍精度演算
  - FORTRAN REAL*16の4.3倍高速
  - 倍精度の3.6倍の実行時間
- Lisの並列性能
  - ノード間　32PEでの速度向上は22～32
  - ノード内　4スレッドでの速度向上は2

今後の展開

- 複素数への対応
- 行列のオーダリング

Lisのご利用お待ちしております

http://www.ssisc.org/lis