PCクラスタコンソーシアムHPCオープンソースソフトウェア普及部会ワークショップ
「高性能数値計算ライブラリLexADV」, 2021年9月10日

階層型領域分割法ライブラリ LexADV_lsDDMの概要と 並列反復法の実装

荻野正雄(大同大学)

自己紹介

- 所属
  - 大同大学情報学部　准教授
- 研究分野
  - 計算工学, 高性能計算
- プログラミング言語
  - C言語は20年以上, Fortranは読み書き程度, Pythonはそれなり
  - MPIとOpenMPはそれなり, Pthreadは読み書き程度
- ソフトウェア開発
  - ADVENTUREプロジェクト: AdvSolid, AdvThermal
  - HDDMPPSプロジェクト: LexADV_lsDDM, LexADV_TryDDM, LexADV_WOVis
- PC歴
  - 1997年に大学の授業でPCでのFortran 77プログラミングを学んでから.
  - 1998年から研究室でSun OSやLinux OSに触れる機会を得る. その後メイン機種で使ってきたLinux OSは, Kondara → Red Hat → SuSE, openSUSE → Ubuntu
  - 2000年代は研究室でよくPCクラスタ(数十台規模)を作っていた.

階層型領域分割法ライブラリの概要

- LexADV_IsDDMの概要
  - 特徴
    - 固体解析向けAdvSolidモジュールの線形方程式ソルバ部分を置換
    - Schur補元方程式を陰的に構築
    - AdvMetisによる領域分割メッシュに対応
    - C言語, MPI+OpenMPによるハイブリッド並列
  - 履歴
    - Ver.0.1b: 2016年8月25日【非公開】

高効率なDDMを実装したい人向け参照コード

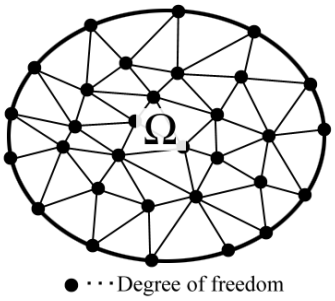
- LexADV_TryDDMの概要
  - 特徴
    - 領域分割法(DDM)に基づいて線形方程式を求解
    - Schur補元方程式を陽的に構築
    - AdvMetisによる領域分割メッシュに対応
    - C言語, MPIによるプロセス並列+並列BLAS
  - 履歴
    - Ver.0.1b: 2014年3月23日公開
    - Ver.0.2a: BDD-DIAG前処理法追加【公開前】

線形ソルバとしてのDDMの評価やDDM前処理の開発向け

このスライドはざっくりとした説明なので細かいことは考えないでください

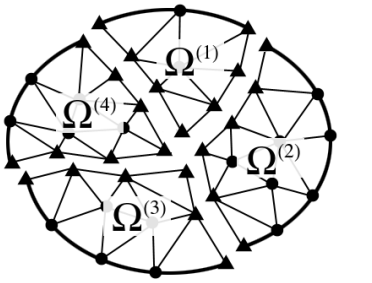
領域分割法による並列有限要素解析(1)

- 有限要素法(FEM)
  - 汎関数$A(x)$の最小化問題と考える
    $\min A(x)$ s.t. xの境界条件



メッシュの領域分割

- 領域分割法(DDM)
  - N個の部分領域に重なりがないよう分割する
  - 汎関数$A(x^{(1)}), \ldots, A(x^{(N)})$の最小化問題と考える
    $\min A(x^{(i)})$ s.t. $x^{(i)}$の境界条件と
    $x^{(i)}$の領域間境界条件



領域間境界条件は不明なので，適当な初期値から開始し，反復法で修正していく

領域分割法による並列有限要素解析(2)

- DDMの利点
  - 既存のFEMコードを内包する形で開発できる
  - (反復法の選び方によるが)部分領域ごとの計算は並列化できる
    - 例えば，OpenMP並列化されたFEMコードがあれば，部分領域ごとの計算をMPIで分散並列処理する，など
- DDMの欠点
  - FEMコードが数万行規模であることも多く，実際の開発は大変
  - 元のFEMコードの(データ構造などの)影響を受けるため，FEMコードごとに開発する必要がある
  - 反復法は元のFEMを考慮して構築するため，反復法の変更はDDMアルゴリズムの変更になる

ADVENTURE界隈の話を聞いてDDMを試したくても気軽にできない
ADVENTUREの中の人でもコード改修は大変な労力である

DDMを数値線形代数に帰着させる

- 解くべき線形方程式

$$A\mathbf{x} = \mathbf{b}$$

一般的な反復法ライブラリはこの式をそのまま解く

- N個の領域分割で未知数を領域内部(I)と領域間境界(B)で並び替え

$$\begin{bmatrix} A_{II}^{(1)} & & 0 & A_{IB}^{(1)} \\ & \ddots & & \vdots \\ 0 & & A_{II}^{(N)} & A_{IB}^{(N)} \\ A_{BI}^{(1)} & \cdots & A_{BI}^{(N)} & \sum A_{BB}^{(i)} \end{bmatrix} \begin{bmatrix} \mathbf{x}_I^{(1)} \\ \vdots \\ \mathbf{x}_I^{(N)} \\ \mathbf{x}_B \end{bmatrix} = \begin{bmatrix} \mathbf{b}_I^{(1)} \\ \vdots \\ \mathbf{b}_I^{(N)} \\ \sum \mathbf{b}_B^{(i)} \end{bmatrix}$$

- 式の分割
  - 未知数$\mathbf{x}_B$に関するSchur補元方程式

$$S\mathbf{x}_B = \mathbf{g}$$

これを反復法で解く

- $S = \sum_{i=1}^{N} \left\{ A_{BB}^{(i)} - A_{BI}^{(i)} \left(A_{II}^{(i)} \right)^{+} A_{IB}^{(i)} \right\}, \mathbf{g} = \sum_{i=1}^{N} \left\{ \mathbf{b}_B^{(i)} - A_{BI}^{(i)} \left(A_{II}^{(i)} \right)^{+} \mathbf{b}_I^{(i)} \right\}$

  - 未知数$\mathbf{x}_I$に関する領域内部方程式

$$A_{II}^{(i)} \mathbf{x}_I^{(i)} = \mathbf{b}_I^{(i)} - A_{IB}^{(i)} \mathbf{x}_B, \qquad i = 1, \ldots, N$$

$S\mathbf{x}_B = \mathbf{g}$に対する反復法の実装

- Sの成分を陽に計算しない陰的構築 (LexADV_IsDDM)
  - 行列ベクトル積$\mathbf{q} = S\mathbf{p}$の計算手順
    - step 1. 行列ベクトル積　$\mathbf{t}^{(i)} = A_{IB}^{(i)}\mathbf{p},\ \ i = 1, \ldots, N$
    - step 2. 線形方程式求解 $A_{II}^{(i)}\mathbf{u}^{(i)} = \mathbf{t}^{(i)}, i = 1, \ldots, N$
    - step 3. 行列ベクトル積　$\mathbf{q} = \sum\left(A_{BB}^{(i)}\mathbf{p} - A_{BI}^{(i)}\mathbf{u}^{(i)}\right)$
  - 部分領域単位で並列処理

行列Sの成分が不明のため，評価できる前処理法が限定される

- Sの成分を陽に計算する陽的構築 (LexADV_TryDDM)
  - 最初に$S = A_{BB} - A_{BI}(A_{II})^{+}A_{IB}$を計算する
  - 行列ベクトル積$\mathbf{q} = S\mathbf{p}$はそのまま計算する
  - 行列分散で並列処理

ほとんどの反復法・前処理法が評価可能である

LexADV_IsDDMの構成と利用方法

- ライブラリに含まれる主なファイル
  - `hddmpps_matrix.c` ... Schur補元行列とのベクトル積計算
  - `hddmpps_matrix_expansion` ... Schur補元からの展開
  - `hddmpps_matrix_reduction.c` ... Schur補元への縮約
  - `hddmpps_solver_cg.c` ... Schur補元方程式をCG法で解く
  - `hddmpps_solver_cr.c` ... Schur補元方程式をCR法で解く
  - `hddmpps_solver_minres.c` ... MINRES法で解く
  - `hddmpps_precon_bdd_1.c` ... BDD前処理法ステップ1
  - `hddmpps_precon_bdd_2.c` ... BDD前処理法ステップ2
  - `hddmpps_precon_bdd_3.c` ... BDD前処理法ステップ3
  - `hddmpps_precon_bdd_4.c` ... BDD前処理法ステップ4
  - `hddmpps_vector.c` ... ベクトル演算
- 利用方法
  - 自前アプリのソースファイルと一緒にビルドする方式
    - いわゆるライブラリ化すると，行列データのコピー作成が必要になり，時間・空間的にもコスト増になってしまうため
    - DDMソルバ向けのデータ構造が複雑で，API設計がうまくいっていないため・・・

LexADV_lsDDMの使い方

```
hddmpps_setup_options(&argc, &argv);
...
hddmpps_set_mesh(mynparts, pmesh);
hddmpps_allocate_matrix();
...
#pragma omp parallel for schedule(dynamic,1)
for (idom = 0; idom < pmesh->n_domain; idom++) {
  /* making coef. matrix of sub-domain in gk */
  /* making RHS vector of sub-domain in gf */
  hddmpps_set_matrix(idom, &gk);
  hddmpps_set_vector(idom, gf);
}
hddmpps_solver_cg(nparts, optsw, cgoption,
                  nr_norm, pmesh, pfield);
```

コマンドライン引数を渡す

領域分割メッシュ情報を渡す

係数行列のためのメモリ割り当て

部分領域ごとに係数行列$A^{(i)}$と右辺ベクトル$\mathbf{b}^{(i)}$を作成して渡す

$A\mathbf{x} = \mathbf{b}$をDDMに基づいてCG法で解く

CG法ソルバhddmpps_solver_cg.cの中身

```
...
hddmpps_reduction(ap_h, r_h);
hddmpps_vec_part_gather(p_hddm_npart[0], p_hddm_partid[0],
          p_hddm_outfree_num[0], p_hddm_outfree_list[0], r_h);
...
  /* CG loop */
  while (1) {
    for (i = 0; i < n_total; i++)
      ap_h[i] = 0.0;
    hddmpps_matvec(p_h, ap_h);
    hddmpps_vec_part_gather(p_hddm_npart[0], p_hddm_partid[0],
          p_hddm_outfree_num[0], p_hddm_outfree_list[0], ap_h);
    myprod = 0.0;
    for (i = 0; i < n; i++)
      myprod += p[i] * ap[i];
    COM_GrpSumDbl(&myprod, &pap, 1);
    alpha = rr / pap;
```

初期残差$\mathbf{r}_0 = g - S\mathbf{x}_0$を計算

ベクトル$\mathbf{r}$は分割されているので，袖領域通信が必要

行列ベクトル積の$\mathbf{q} = S\mathbf{p}$計算

ベクトル$\mathbf{q}$の袖領域通信

LexADV_IsDDMを導入したアプリの実行

- 基本的には元々のアプリの実行方法通り
- IsDDMオプションとして以下を用意している

```
Usage:
  mpiexec [mpi_options] app_name -iterate [solver] -precon [pc]

Options:
  solver: cg, cr, minres
  pc: jacobi, nn, bdd, bdd-diag
```

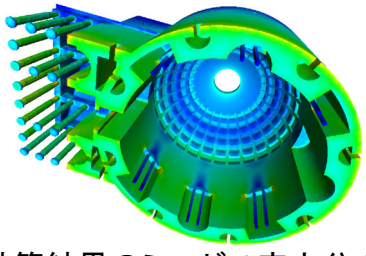
他に, Pipelined CG法, ドット積を倍々精度計算, なども実装済み

DDM専用の前処理

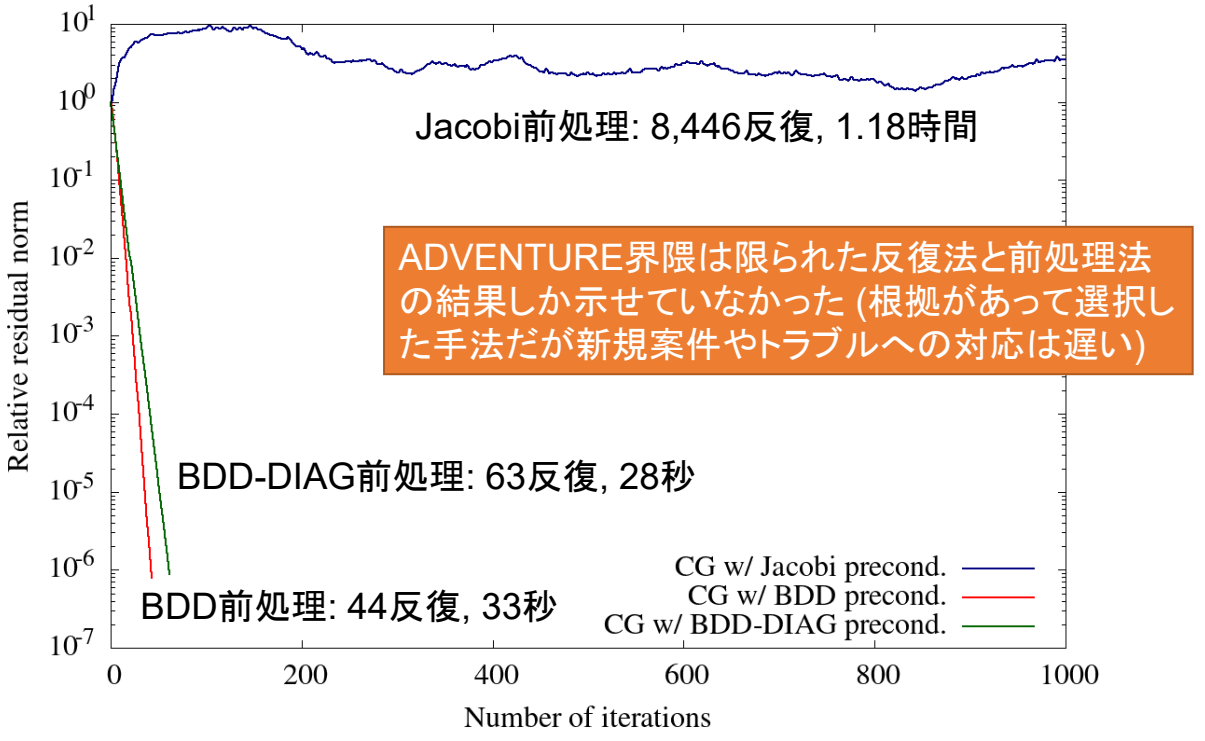
- nn: Neumann-Neumann [De Roeck & Le Tallec, 1991]
- bdd: Balancing Domain Decomposition [Mandel, 1993]
- bdd-diag: BDD w/ diagonal-scaling [Ogino et al., 2008]

LexADV_IsDDMの実行例(1)

線形弾性問題の有限要素解析, 未知数約100万
計算機: 仮想マシン@Intel i7-8560U, Ubuntu OS

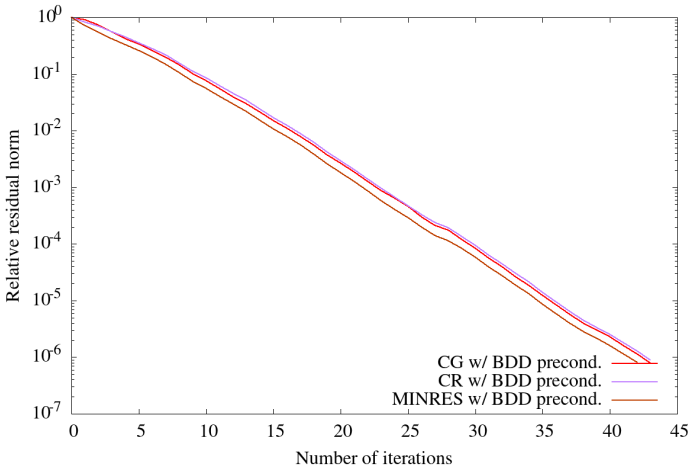


計算結果のミーゼス応力分布

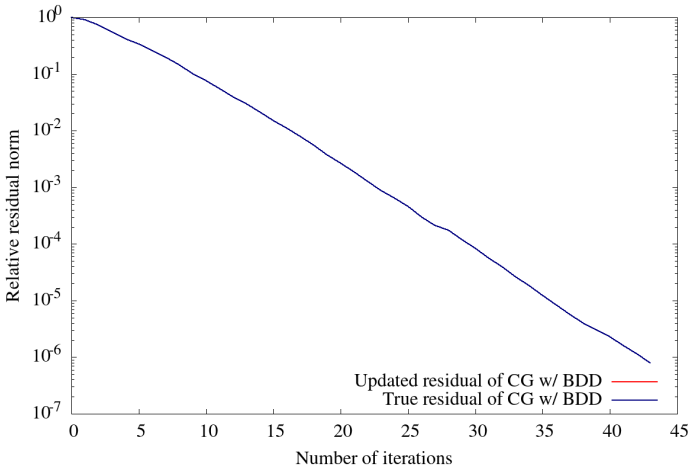


ADVENTURE界隈は限られた反復法と前処理法の結果しか示せていなかった (根拠があって選択した手法だが新規案件やトラブルへの対応は遅い)

LexADV_IsDDMの実行例(2)



反復法(CG, CR, MINRES)ごとの
収束傾向比較
(このケースはCG法で十分と確認)



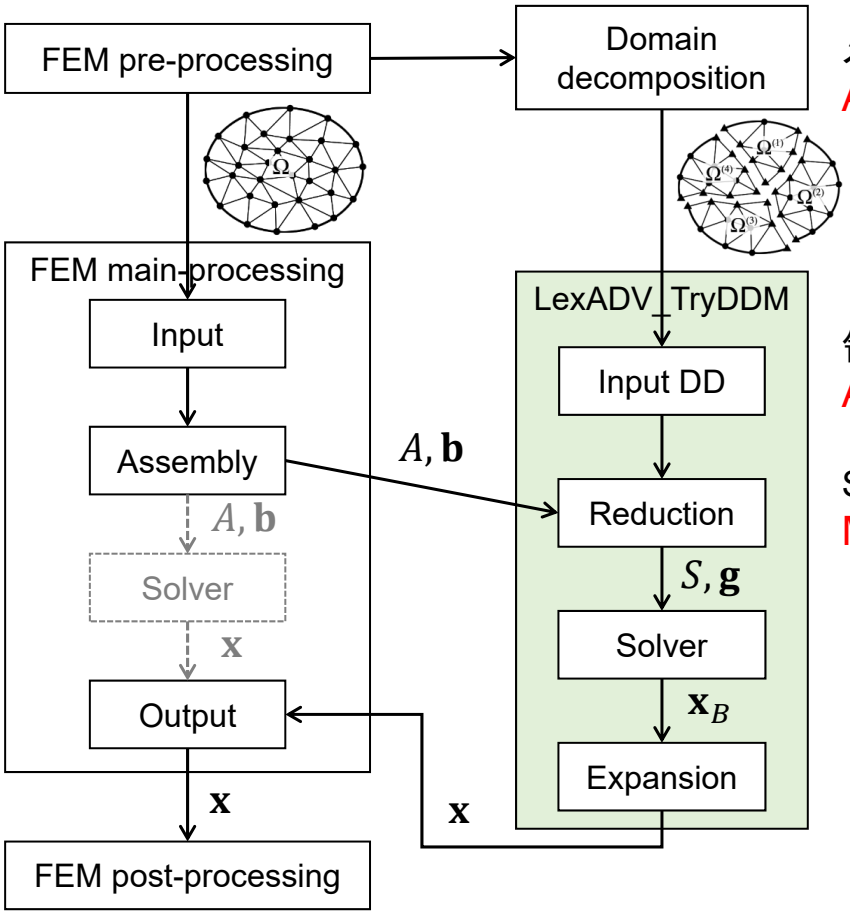
更新残差$\mathbf{r}_n$と真の残差$\mathbf{b} - A\mathbf{x}_n$の
ノルム比較
(一致しており偽収束でないことを確認)

LexADV_IsDDMによって，数万行規模で書かれたDDMコードであるAdvSolidで反復法の選択が容易になり，様々な評価や検討が可能になった．さらに，例えば係数が非対称になるなど新たな反復法が必要な場合の開発コストも低くなる．

LexADV_TryDDMディレクトリ構成

- ライブラリに含まれるディレクトリ構成
  - `doc` ... マニュアル
  - `lib` ... ビルドされたライブラリがコピーされる
  - `samples` ... サンプルデータ
  - `src`
    - `common` ... global/local共通のプログラムソース
    - `common-include` ... global/local共通のヘッダーファイル
    - `global-schur-ver` ... グローバル構築版
    - `local-schur-ver` ... ローカル構築版

TryDDM利用時の処理の流れ



グローバルS構築版とローカルS構築版

- グローバルSchur補元行列構築版(global-schur-ver)
  - $A \rightarrow [\mathrm{MUMPS}] \rightarrow S = A_{BB} - A_{BI}(A_{II})^{+}A_{IB}$
  - MUMPSがマルチフロンタル法による並列行列分解と求解で構築
    - 長所: 全体係数行列とベクトルを入力するのみで実装が容易
    - 短所: 台数効果が低い

- ローカルSchur補元行列構築版(local-schur-ver)
  - $A = \sum_{i=1}^{N} R^{(i)^T} \begin{bmatrix} A_{II}^{(i)} & A_{IB}^{(i)} \\ A_{BI}^{(i)} & 0 \end{bmatrix} R^{(i)} + \begin{bmatrix} 0 & 0 \\ 0 & A_{BB} \end{bmatrix} \rightarrow S = A_{BB} + \sum_{i=1}^{N} R^{(i)^T} S^{(i)'} R^{(i)}$

    $\rightarrow [\mathrm{MUMPS}] \rightarrow S^{(i)'} = -A_{BI}^{(i)} \left(A_{II}^{(i)}\right)^{+} A_{IB}^{(i)}$

  - MUMPSが局所Schur補行列を構築
    - 長所: 台数効果が高い
    - 短所: 実装が少し複雑になる

依存ライブラリ等

- LexADV_TryDDM依存ライブラリ
  - 入出力ライブラリ AdvIO
    - ADVENTUREプロジェクト
  - 反復法ライブラリ Lis
    - SSIプロジェクト
  - 直接法ライブラリ MUMPS
    - PARASOLプロジェクト
    - BLAS, BLACS, ScaLAPACKが必要 (Intel MKL等でも良い)
- その他
  - 領域分割メッシュ作成ツール AdvMetis
    - ADVENTUREプロジェクト
    - AdvIOが必要

LexADV_TryDDMのビルド例

```
download AdvIO-1.2.tar.gz from adventure project
tar xvfz AdvIO-1.21.tar.gz
cd AdvIO-1.2a
./configure
make install

git clone https://github.com/anishida/lis.git
cd lis
./configure --enable-mpi --enable-omp
make install

sudo apt install libmumps-dev

tar xvfz LexADV_TryDDM-0.1b.tar.gz
cd src/local-schur-ver
vi Makefile
make
```

AdvIO

Lis

MUMPS

TryDDM

TryDDM公開版はLis最新版への対応で少し修正が必要

Makefileの編集

```
#-----ADVENTURE------------------
ADVIO_DIR =  (AdvIOインストールディレクトリ)
ADVIO_CONFIG = $(ADVIO_DIR)/bin/advsys-config
ADVIO_INCLUDES = `$(ADVIO_CONFIG) --cflags`
ADVIO_LIBS = `$(ADVIO_CONFIG) --libs docio`
#-------MUMPS---------------------------------
MUMPS_INCLUDES =  -I$(MUMPSインクルードディレクトリ)
MUMPS_LIBS     =  -L$(MUMPSリンクディレクトリ) -ldmumps -lmumps_common -lpord
SCALAP_LIBS    = (ScaLAPACKへのリンク)
BLAS_LIBS      = (BLASへのリンク)
#-------Lis-----------------------------------
LIS_INCLUDE =  -I$(Lisインクルードディレクトリ)
LIS_LIBS =  -L$(Lisリンクディレクトリ) -llis

MPI_INCLUDE =  -I$(MPIインクルードディレクトリ) #必要に応じて
MPI_LIBS =  $(MPIへのリンク) #必要に応じて
DDM_INCLUDES = -I../common-include

CC = mpicc    (MPI Cコンパイラ)
CFLAGS = -O3 -DUSE_MPI -fopenmp  (コンパイラフラッグ)
INCLUDES = $(DDM_INCLUDES) $(ADVIO_INCLUDES) $(LIS_INCLUDE)  $(MUMPS_INCLUDES) ¥
           $(MPI_INCLUDE)
LIBS = $(ADVIO_LIBS) $(LIS_LIBS) $(MUMPS_LIBS) $(SCALAP_LIBS) $(BLAS_LIBS) $(MPI_LIBS) -lm
LINKER = mpicc    (MPIリンカ)

PROGRAMS = libtry_ddm_gl.a
...
```

利用方法

```
int try_ddm(int argc, char** argv, int *csr_ptr, int
*csr_idx, double *csr_val, int A_nnz, int A_matdim, int
b_dim, double *b_val, char *mesh_file, char *mesh_type);
```

	引数	説明
オプション指定	int argc	引数の数
	char** argv	引数リスト
係数行列	int* csr_ptr	各行開始位置リスト (CSR形式)
	int* csr_idx	非零要素の列番号リスト (CSR形式)
	double* csr_val	非零要素の値リスト (CSR形式)
	int A_nnz	非零要素数
	int A_matdim	行列次元数
右辺ベクトル	int b_dim	ベクトル次元数
	double* b_val	値リスト (dense形式)
領域分割メッシュ	char* mesh_file	領域分割ファイル名
	char* mesh_type	メッシュタイプ (4面体2次: tetra)

サンプルプログラムとデータ

ディレクトリ構造

- samples/
  - Makefile ... メイクファイル
  - sample.c ... サンプルプログラム
  - data/
    - Ab/
      - Ab-130k.dat ... 係数行列と右辺ベクトル
    - Mesh/
      - adv-metis/
        - 130k/p1d240/model/advhddm_in_0.adv ... 領域分割メッシュ
      - origin-mesh/
        - 130k.adv ... 領域分割前メッシュ(AdvIO形式)

サンプル実行例

```
mpiexec -n 2 ./sample ¥
-i cg -p ssor -maxiter 10000 -tol 1.0e-6 -print out ¥
tetra ./data/Ab/Ab-130k.dat ¥
./data/Mesh/adv-metis/130k/p1d240/model/advhddm_in_0.adv
####################################
Matrix A and Vector b
  matdim:128814
  nnz:9726120
DDM Meshfile
  SubDomains:240
  Global Schur matdim:36219
...
iteration:  1617  relative residual = 9.719264E-07
linear solver status  : normal end
...
log10(||b-Ax_n|| / ||b||) = -6.040980e+00
[0]*** All elapsed time :82.436755 ***
```

反復法ソルバLisへのオプション指定

TryDDMへのオプション指定

入力した係数行列Aの情報

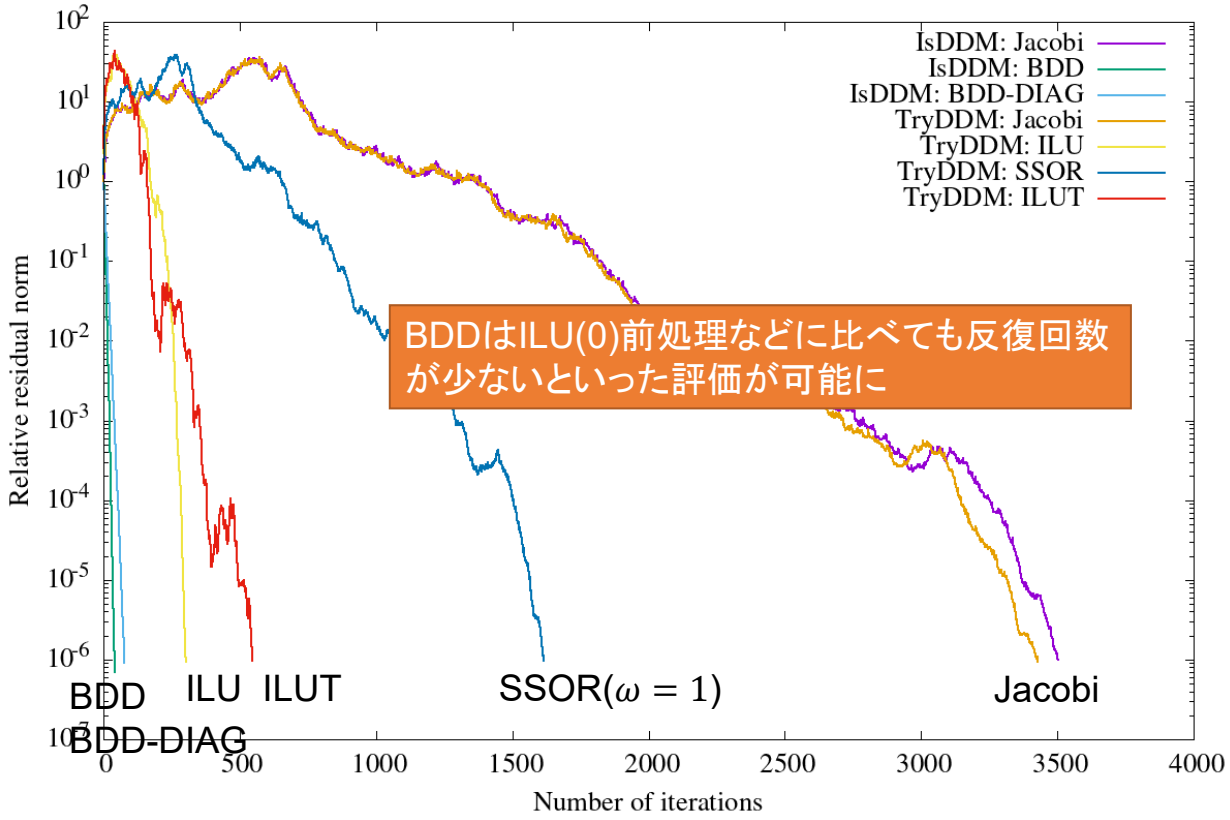
Schur補元行列Sの情報

反復法停止時の情報

得られた近似解の誤差

全体の計算時間

IsDDMとTryDDMを用いた前処理の性能評価例



領域分割数を変えたサンプル実行 (1)

- 同梱サンプルは領域分割数240 (p1d240)
- 領域分割数を変えた実験を行う例
  - AdvMetisの実行例(領域分割数120)

```
adventure_metis origin-mesh/130k.adv adv-metis/p1d120 120
```

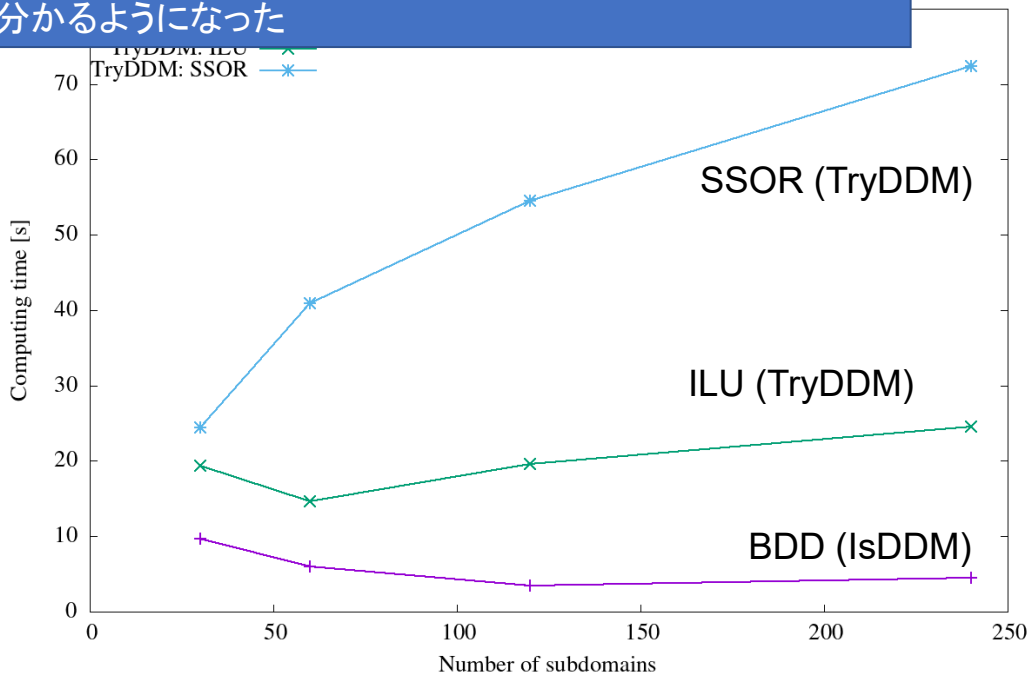
  - サンプルプログラムの実行

```
mpiexec -n 2 ./sample ¥
-i cg -p ssor -maxiter 10000 -tol 1.0e-6 -print out ¥
tetra ../Ab/Ab-130k.dat ¥
Mesh/adv-metis/130k/p1d120/model/advhddm_in_0.adv
###################################
...
DDM Meshfile
  SubDomains:120
  Global Schur matdim:25038
###################################
...
iteration:  1333  relative residual = 9.843737E-07
linear solver status  : normal end
...
[0]*** All elapsed time :60.299284 ***
```

DDMでは領域分割数を変えるとSchur補元行列のサイズも変わり，それが反復回数や計算時間に影響する

領域分割数を変えたサンプル実行 (2)

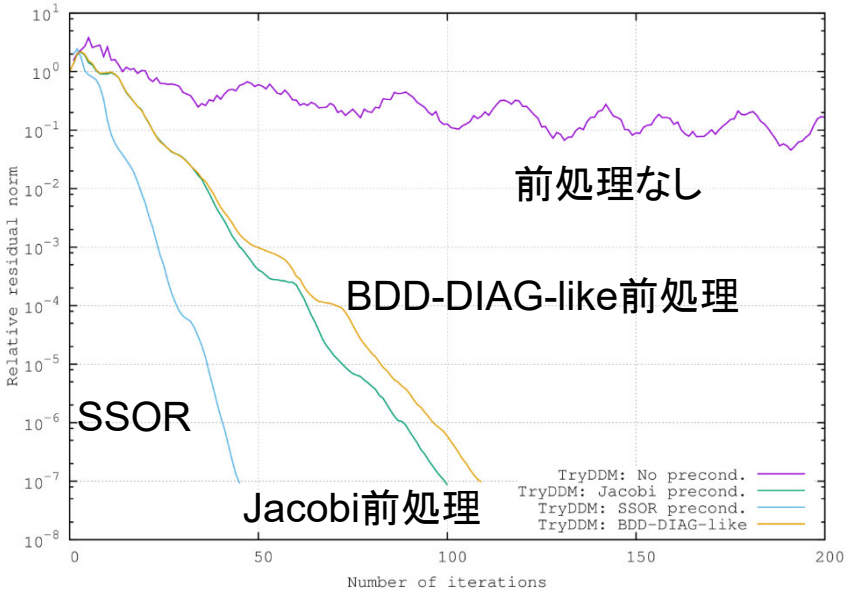
BDDやILUはSSORに比べて領域分割数に対してロバストだな，などが分かるようになった

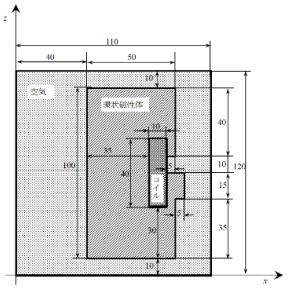


領域分割数 vs. 計算時間

LexADV_TryDDMの応用例
静磁場問題のBDD-DIAG前処理開発

- 静磁場の有限要素解析における線形方程式
  - 磁気ベクトルポテンシャルを未知数とするA法
- BDD前処理フレームワークの適用
  - Coarse空間の基底 $Z_P^{(i)} = [1]$ (BDDの理論は満たさない)





An axial model

まとめと今後の計画

- 階層型領域分割ライブラリの概要
  - LexADV_IsDDMは，
    - 高性能・高機能だが線形ソルバ変更が困難なADVENTUREモジュールにおいて，反復法の変更を容易に実現するもの
    - API設計が完了しておらず，現時点ではAdvSolidモジュールに直接組み込む形での利用となり，非公開扱いである
    - 名大「不老」やR-CCS「富岳」などでの最先端研究でも使われている
  - LexADV_TryDDMは，
    - DDM向けの前処理法開発を支援するもの
    - 自前アプリの線形方程式におけるDDM性能の予備調査にも
  - 今後の計画
    - LexADV_IsDDMの公開は，AdvSolid同梱という形で検討
    - LexADV_TryDDMはVer.0.2bを公開予定