ユーザズ・マニュアル

EigenExa version 1.0

2013 年 8 月 8 日改訂
2013 年 8 月 1 日初版
EigenExa チーム

1. はじめに

EigenExa は高性能固有値ソルバである．EigenExa は EigenK ライブラリ[1] の後継として開発され，ペタスケールコンピュータシステム同様に，将来登場するであろうポストペタスケール計算機システム(所謂「エクサ」または「エクストリーム」システム)でスケーラブルに動作するように設計されている．現リリースでは，全ての固有対(「固有値」と「対応する固有ベクトル」)を計算するという最もシンプルな機能を提供する．[1,2]で報告されているように, EigenK がそうであるのと同様 EigenExa もまた古典的なアルゴリズムと先進的なアルゴリズムの両者を採用して対角化に要する計算時間を削減している．

EigenExa は，MPI, OpenMP, 高性能 BLAS, 更に SIMD ベクトル化 Fortran90 コンパイラ技術など様々な並列プログラミング言語とライブラリを用いて開発されている．明らかに，次に挙げる項目が複数同時に機能して，高性能計算を実現することが期待される．

1) MPI による分散メモリ型のノード間並列性
2) OpenMP による共有メモリ型並列計算機もしくはマルチコアプロセッサでの並列性
3) ベンダにより高度に最適化された BLAS を用いた高い並列性
4) ベンダ提供の高性能コンパイラを用いた SIMD もしくは疎粒度並列性

Fortran90 のよい特徴も積極的に取り込まれている．EigenExa の API は Fortran77 による実装ライブラリよりも柔軟であり，モジュールインターフェイスや省略可能引数によりユーザーフレンドリーなインターフェイスが提供される．

並列性能の観点から見てみると, EigenK の通信オーバヘッドを削減することで高い性能向上を達しており，多くの場合に EigenExa は EigenK や ScaLAPACK などの最高水準の数値計算ライブラリより高性能であることが確認されている[3].

本ドキュメントは EigenExa version 1.0 のユーザズ・マニュアルである．導入開始から実際の使用までの内容を記している．特に，導入，コンパイル，クイックチュートリアル，API リスト，EigenK との互換性の注意が選択されている．EigenExa チームの全ての開発者は，本ドキュメントが多くの利用者に対して並列シミュレーションを効率よく走らせるための手助けになることを期待している．

2. 利用の前に

EigenExa のインストールのために必要なソフトウェア:

EigenExa ライブラリをコンパイルするためには幾つかのソフトウェアパッケージを準備しないといけない. BLAS, LAPACK, ScaLAPACK 更に MPI は EigenExa のコンパイルの前にシステムにインストールされていなくてはない.
現在のところ, EigenExa は以下に示すライブラリでコンパイルできることが確認されている.

BLAS	Intel MKL, GotoBLAS, OpenBLAS, ATLAS, Fujitsu SSL II
LAPACK	Version 3.4.0 以降
ScaLAPACK	Version 1.8.0 以降
MPI	MPICH2 version 1.5 以降, MPICH version 3.0.2 以降 OpenMPI version 1.6.4 以降

EigenExa:の入手方法

EigenExa に関する全ての情報は次の URL から入手可能である. :

http://www.aics.riken.jp/labs/lpnctrt/EigenExa.html

tarball の配布も上記サイトからされる. EigenExa に関するその他の情報も提供されていく予定である.

コンパイルとインストール手順

EigenExa ライブラリのコンパイルには幾つかの手順が必要である. 次のインストレーション手引きに従ってほしい.

まず, tarball をワーキングディレクトリ上で展開する. そして, ディレクトリ EigenExa-1.0 に移動する.

```
% tar zcvf EigenExa-1.0.tgz
% cd EigenExa-1.0
```

次に, Makefile と make_inc.xxx を使用者の環境に合わせて編集する. ここで, xxx には BX900, Intel, K_FX10, もしくは gcc の中から（特に）使用するコンパイラに応じた文字列が入る.
３番目に, make を実行する. その結果スタティック・ライブラリ libEigenExa.a が作成される.

```
% make
```

最後に, libEigenExa.a と eigen_libs.mod と eigen_blacs.mod をインストールディレクトリにコピーし終了である. 例えば /usr/local/lib にインストールする場合は次のようにする.

```
% cp libEigenExa.a eigen_libs.mod eigen_blacs.mod   /usr/local/lib/
```

3. クイックチュートリアル

ワーキングディレクトリに移り，‘make test’ を実行すると標準ベンチマークコードが得られる．ソースコード中の ‘main2.F’ と ‘Makefile’はコード作成に役立つはずである．

main2.f の核部分を取り出したものが次のようになる．

```
use MPI
use eigen_libs
...
call MPI_Init_thread( MPI_THREAD, MULTIPLE, i, ierr )
call eigen_init( )

N=10000; mtype=0

call eigen_get_matdims( N, nm, ny )
allocate ( A(nm,ny), Z(nm,ny), w(N) )
call mat_set( N, a, nm, mtype )
call eigen_sx( N, N, a, nm, w, z, nm, m_forward=32, m_backward=128)
deallocate ( A, Z, w )
...
call eigen_free( )
call MPI_Finalize( ierr )
end
```

上のコードは骨格部分を示したのみであり実際の動作はしないが，「初期化」→「配列確保」→「固有値計算」→「終了手続き」の流れを示すには十分なものである．

上例では初期化関数 eigen_init()を引数省略型呼び出しで実行している．eigen_init()には固有値計算を実施するグループをコミュニケータとして comm=XXX の形で指定できる．複数のグループで同時に固有値計算を並列実行したいときには MPI_Comm_split()等で作成されたコミュニケータを渡すことで並列計算可能となる．ただし，現実装では eigen_init()の内部でコミュニケータ MPI_COMM_WORLD に対する集団操作を行うため，eigen_init()は MPI_COMM_WORLD に属する全プロセスが同時に呼び出さなくてはならないという制約がある．そこで、個々のプロセスごとに異なるコミュニケータを指定できるので，固有値計算に参加しないプロセスは MPI_COMM_NULL もしくは MPI_COMM_SELF を eigen_init()に指定して，固有値計算ドライバ eigen_sx()自身の呼び出しをスキップさせることができる．つまり，eigen_sx()の処理以外に eigen_sx()を含む様々な演算を同時実行することができる．

EigenExa では eigen_init()で指定されたコミュニケータに属するプロセスを２次元プロセスグリッドに配置して使用する．できるだけ通信量が削減できるように正方に近い形のプロセスグリッドを採用するよう設計されている．また，EigenExa は利用者の便宜を高めるという

観点から，MPI で採用されている２次元カーテシアンを comm に指定することができるように開発されている．原理的にはカーテシアンの形状が２次元であれば任意のプロセス配置に対して EigenExa を呼び出して計算することができるので，上述の複数種類のコミュニケータとの組み合わせにより複雑な並列処理を実行することができる．
なお，カーテシアンは基本的にプロセスグリッドが Row-major になるため，order='C'指定と矛盾するときはカーテシアンを優先して扱うことになっている．なお，EigenExa は歴史的経緯からデフォルトのプロセスグリッドは Column-Major を採用している．

eigen_sx()の呼び出しの直前に呼び出している mat_set()において行列データの生成を行っている．行列データは指定された２次元プロセスグリッド上に２次元サイクリック分割のスタイルで分散されており，ローカル配列として各プロセスに格納されている．各プロセスは行列の一部のデータのみを格納するため，行列要素のアクセスをする場合にはグローバルインデックスとローカルインデックス間の変換ルールが必要である．

次のプログラムは mat_set()からの抜粋であり，グローバルカウンタループ構成の Frank 行列生成プログラムをローカルカウンタループに変換した両者を対比として示している．

```
! Global loop program to compute a Frank matrix
do i = 1, n
   do j = 1, n
      a(j, i) = (n+1-Max(n+1-i,n+1-j))*1.0D+00
   end do
end do
```

```
! Translated local loop program to compute a Frank matrix
use MPI
use eigen_libs
call eigen_get_procs( nnod, x_nnod, y_nnod )
call eigen_get_id   ( inod, x_inod, y_inod )

j_2 = eigen_loop_start( 1, x_nnod, x_inod )
j_3 = eigen_loop_end  ( n, x_nnod, x_inod )
i_2 = eigen_loop_start( 1, y_nnod, y_inod )
i_3 = eigen_loop_end  ( n, y_nnod, y_inod )
do i_1 = i_2, i_3
   i = eigen_translate_l2g( i_1, y_nnod, y_inod )
   do j_1 = j_2, j_3
      j = eigen_translate_l2g( j_1, x_nnod, x_inod )
      a(j_1, i_1) = (n+1-Max(n+1-i,n+1-j))*1.0D+00
   end do
end do
```

ループ範囲の変換は eigen_loop_start()もしくは eigen_loop_end()を使用する．第二，第三引数は分散方向を示すコミュニケータから派生されるプロセス数とプロセス ID を指定する．本ドキュメントでは常に「行」→”x”，「列」→”y”の対応になっている(参加プロセス全体のコミュニケータの場合は，”x”や”y”の部分を無文字にしている)．
ここで，重大な注意として「EigenExa ではプロセス ID を 1 から始まる整数で管理している」．そのため，問合せ関数 eigen_get_id()によって取得したプロセス ID は MPI のランクと 1 だけずれている．MPI のランクが必要な場合はプロセス ID から 1 減じる必要がある．

上記プログラムではローカルなループカウンタ値から対応するグローバルカウンタ値に変換して使用する．その変換には eigen_translate_l2g()を使用する．第二，第三引数は eigen_loop_start()などと同様に指定するとよい．逆に，グローバルカウンタ値をローカルカウンタ値に変換するには eigen_translate_g2l()を使用する．ただし，eigen_translate_g2l()はグローバルカウンタ値をループカウンタとして見たときに，オーナープロセス(そのグローバルカウンタ値に対応するローカルカウンタ値をループ内に含むプロセス)となるプロセス上で対応するローカルカウンタ値を返すこととする．指定したグローバルループカウンタのオーナープロセスを知るには eigen_owner_node()を使用する．特定の行ベクトルや列ベクトルの値を参照したりブロードキャストする際に利用するとよい．

更に，ScaLAPACK と連携した上級者向けの計算を進めたいときは，補助関数 eigen_get_blacs_context()により EigenExa で使用するプロセスグリッドコンテキストを取得すればよい．mat_set()関数の mtype=2 の部分を参照するとよい(次例は，行列 AS の転置を行列 A に格納する PDTRAN()呼び出しの核部分を取り出したものである)．

```
! Cooperation with ScaLAPACK
NPROW = x_nnod; NPCOL = y_nnod

ICTXT = eigen_get_blacs_context( )
CALL DESCINIT( DESCA, n, n, 1, 1, 0, 0, ICTXT, nm, INFO )

! A ← AS^T
CALL PDTRAN( n, n, 1D0, as, 1, 1, DESCA, 1D0, a, 1, 1, DESCA )
```

コンパイル時には mpif90 を使用するとともに，eigen_libs などモヂュールへのアクセスが必要となるためパスの設定をしておく必要がある（多くの場合は-I オプションである）．また、EigenExa ライブラリをリンクするには，MPI, OpenMP, ScaLAPACK(バージョンが 1.8 以前の場合は BLACS も)などを同時にリンクする必要がある．Intel コンパイラベースの MPI の場合は以下のようにする(ScaLAPACK や BLAS まわりライブラリ名は環境によって異なる)．

```
% mpif90 –c a.f –openmp –I/usr/local/include –I/usr/local/lib
% mpif90 –o exe a.o –openmp –L/usr/local/lib –lEigenExa –lscalapack –llapack –lblas
```

4. API

本節では‘eigen_libs.mod’中で public 属性を与えられた関数をリストアップする．始めの三ルーチンはメインドライバーであり，その他はユーティリティ関数である．Optional 属性のついた引数の場合は省略もできるし，FORTRAN フォーマット形式である *TERM=variable or constant value* でも指定することができる．

1. eigen_init

EigenExa の機能を初期化する．プロセスグリッドマッピングを引数 ‘comm’ や ‘order’ を通じて指定することができる．

subroutine eigen_init(comm, order)			
comm	基盤となるコミュニケータ	integer, optional	In
	省略時は MPI_COMM_WORLD		
order	Row もしくは Column	character*(*), optional	In
	省略時は ‘C’として扱われる．グリッドメジャーがカーテシンアン comm の指定と矛盾するときは‘R’が採用される．		

2. eigen_free

EigenExa の機能を終了する．

eigen_free(flag)			
flag	タイマープリンタのフラグ	integer, optional	In
	この引数は開発用途のためのものである．省略時は 0 である．		

3. eigen_sx

EigenExa の主たるドライバルーチンである．

subroutine eigen_sx(n, nvec, a, lda, w, z, ldz, m_forward, m_backward)			
n	行列・ベクトルの次元	integer	In
nvec	計算する固有ベクトルの本数	integer	In
	現在このオプションはサポートされていない．eigen_sx()は全固有ベクトルを計算する		
a	対角化の対称行列	real(8)	InOut
lda	配列 a の整合寸法（リーディングディメンジョン）	integer	In
w	昇順の固有値	real(8)	Out
z	行列 a の直交固有ベクトル	real(8)	Out
ldz	配列 z の整合寸法(リーディングディメンジョン)	integer	In
m_forward	ハウスホルダー変換のブロック係数(偶数でなければいけない)	integer, optional	In
m_backward	ハウスホルダー逆変換のブロック係数	integer, optional	In

4. eigen_get_matdims

本関数で取得した配列寸法(nx,ny)を使用してローカルな配列を動的に確保する．行列全体は(CYCLIC,CYCLIC)分割されている．

subroutine eigen_get_matdims(n, nx, ny)

n	行列の次元	integer	In
nx	配列 a ならびに z の整合寸法（リーディングディメンジョン）の下限値	integer	Out
ny	配列 a ならびに z の第二インデックスの下限値’	integer	Out

5. eigen_memory_internal

本関数は EigenExa が呼び出されている間に内部で動的に確保されるメモリサイズを返す．利用者は本関数の返却値を知り，メモリ不足に陥らないようにすべきである．

integer function eigen_memory_internal(n, lda, ldz, m1, m0)			
n	行列の次元	integer	In
lda	配列 a の整合寸法（リーディングディメンジョン）	integer	In
ldz	配列 z の整合寸法（リーディングディメンジョン）	integer	In
m1	ハウスホルダー変換のブロック係数(偶数でなければいけない)	integer	In
m0	ハウスホルダー逆変換のブロック係数	integer	In

6. eigen_get_comm

eigen_init()によって生成された MPI コミュニケータを返す．

subroutine eigen_get_comm(comm, x_comm, y_comm)			
comm	基盤となるコミュニケータ	integer	Out
x_comm	行コミュニケータ．行 id が一致する全プロセス所属する．	integer	Out
y_comm	列コミュニケータ．列 id が一致する全プロセスが所属する．	integer	Out

7. eigen_get_procs

eigen_init()によって生成されたコミュニケータに関するプロセス数情報を返す．

subroutine eigen_get_procs(procs, x_procs, y_procs)			
procs	comm 中のプロセス数	integer	Out
x_procs	x_comm 中のプロセス数	integer	Out
y_procs	y_comm 中のプロセス数	integer	Out

8. eigen_get_id

eigen_init()によって生成されたコミュニケータに関するプロセス ID 情報を返す．ここでプロセス ID は MPI ランクとは異なり 1 から開始する成数値で，MPI ランク=プロセス ID-1 の関係にある．

subroutine eigen_get_id(id, x_id, y_id)			
id	comm で定義されたプロセス ID	integer	Out
x_id	x_comm で定義されたプロセス ID	integer	Out
y_id	y_comm で定義されたプロセス ID	integer	Out

9. eigen_loop_start

指定されたグローバルループ開始値に対応するローカルなループ構造におけるループ開始値を返す.

integer function eigen_loop_start(istart, nnod, inod)			
istart	グローバルループ開始値	integer	In
nnod	プロセス数	integer	In
inod	プロセス ID	integer	In

10. eigen_loop_end

指定されたグローバルループ終端値に対応するローカルなループ構造におけるループ終端値を返す.

integer function eigen_loop_end(iend, nnod, inod)			
istart	グローバルループ終端値	integer	In
nnod	プロセス数	integer	In
inod	プロセス ID	integer	In

11. eigen_translate_l2g

integer function eigen_translate_l2g(ictr, nnod, inod)			
ictr	ローカルカウンタ	integer	In
nnod	プロセス数	integer	In
inod	プロセス ID	integer	In

12. eigen_translate_g2l

integer function eigen_translate_g2l(ictr, nnod, inod)			
ictr	グローバルカウンタ	integer	In
nnod	プロセス数	integer	In
inod	プロセス ID	integer	In

13. eigen_owner_node

指定されたグローバルループカウンタ値に対応するオーナープロセスの ID を返す.

integer function eigen_owner_node(ictr, nnod, inod)			
ictr	グローバルループカウンタ	integer	In
nnod	プロセス数	integer	In
inod	プロセス ID	integer	In

5. その他の注意事項

互換性の注意

EigenExa は EigenK の後継で多くの機能を継承している. しかしながら両者の間には完全な互換性を保証していない，それは内部実装の詳細が異なることに起因しており，主に関数や変数の命名則，コモン領域の管理方法の違いによるものである．また，同様の理由により EigenExa と EigenK を同時にリンクすることを勧めない.

他の言語との結合

Fortran90 以外からの EigenExa の呼び出し方法は利用者の環境に大きく依存する．コンパイラマニュアルを引き，「言語結合(language bindings)」や「複数プログラミング言語とのリンク方法」を参照されたい.

シェアード・ライブラリ

現バージョンの EigenExa ではシェアード・ライブラリをサポートしない．これは，現時点においてシェアード・ライブラリ使用時に関数名の解決が衝突なく完全にできることを保証できないからである (gcc のあるバージョンでは実行時に関数名の解決ができず異常終了する場合があった)．シェアード・ライブラリとして利用する場合はあくまでも利用者の責任の元実行して欲しい.

6. 参考文献

[1] Toshiyuki Imamura, Susumu Yamada, Masahiko Machida, “Development of a High Performance Eigensolver on the Peta-Scale Next Generation Supercomputer System”, Progress in Nuclear Science and Technology, the Atomic Energy Society of Japan, Vol. 2, pp.643-650 (2011) .

[2] Susumu Yamada, Toshiyuki Imamura, Takuma Kano and Masahiko Machida, “High-Performance Computing for Exact Numerical Approaches to Quantum Many-Body Problems on the Earth Simulator”, ACM/IEEE SC06, November 2006. Tampa USA.

[3] Toshiyuki Imamura, Susumu Yamada, Masahiko Machida, “Eigen-K: high performance eigenvalue solver for symmetric matrices developed for K computer”, 7th International Workshop on. Parallel Matrix Algorithms and Applications (PMAA2012), June 2012, London UK.

7. 謝辞

EigenExa チームの全員に彼らの真摯な協力と支援に対して感謝する．彼らの努力なしには EigenExa を公開することは叶わなかったであろう．また，EigenExa 開発には幾つかの外部資金の援助を受けている．それらは以下の通りである．併せてここに謝意を表したい.

- 科学技術振興機構　戦略的創造研究推進事業 CREST「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」 (平成 23 年度～27 年度)
- 文部科学省　科学研究費補助（科研費）:基盤研究(B)課題番号 21300013 (平成 24 年度), 基盤研究(A)課題番号 23240005 (平成 23 年度～25 年度)