Vol.2016-HPC-153 No.16
2016/3/2

電子動力学シミュレーションコードのSymmetricモードによるXeon Phiクラスタへの実装と性能評価

廣川 祐太[1,a)]　朴 泰祐[1,3]　佐藤 駿丞[2]　矢花 一浩[2,3]

概要：近年，Intel Xeon Phi プロセッサを搭載した PC クラスタが積極的に運用され．同プロセッサへのアプリケーション実装や最適化が強く求められている．本研究では，実時間密度汎関数理論に基づく電子動力学シミュレータ ARTED を Xeon Phi クラスタ上に実装し，最適化を行った．同アプリケーションの支配的な計算はステンシル計算で，本研究ではコンパイラによる自動ベクトル化，Intrinsics を用いた手動ベクトル化の両方で最適化を行い性能比較行った．特に，手動ベクトル化では連続方向のメモリアクセスについて最適化を行い，212.2 GFLOPS と Ivy-Bridge の 106.9 GFLOPS に対し約 2 倍の性能を達成した．また，我々はアプリケーション実行方法として Xeon Phi を 1 台のノードとみなし計算を行う Native 実行，Xeon Phi クラスタ上の全計算リソースが利用可能な Symmetric 実行を用いた．Native 実行では，CPU-only 実行に対し最大約 1.45 倍の性能を達成した．Symmetric 実行では，Xeon Phi と CPU 間の計算量を調整することで，CPU-only 実行に対し 2 倍以上の性能が得られている．

1. はじめに

Intel Xeon Phi プロセッサは，Many Integrated Cores (MIC) アーキテクチャに基づくアクセラレータで，現プロダクトは Knights Corner (KNC) と呼ばれる．Xeon Phi は GPU と同様に，ホスト CPU に対し PCI-Express によって接続される．Xeon Phi は x86 アーキテクチャに基づき，Linux カーネルが動作しているため，Xeon CPU で開発されたアプリケーションをコードの変更なく容易に実行可能である．しかしながら，Xeon CPU に対してその性能特性は大きく異なる．ここで重要なのは，単純な移植では実アプリケーションに Xeon Phi を適用し高い性能を得るのは非常に困難で，Xeon Phi に対する最適化が強く要求されることである．現在，次世代 Xeon Phi として Knights Landing (KNL) が開発されている．KNL プロセッサでは，一般的な Xeon CPU と同様にソケット接続による製品が提供される．KNC は PCI-Express で接続されるためホスト CPU を要求し，Xeon CPU と Xeon Phi のヘテロジニアスクラスタしか実装できなかった．KNL では，KNL 自体がホスト CPU として利用できるため，KNL を唯一の CPU としてクラスタを実装可能となった．

本研究では，Xeon Phi クラスタ COMA に対し実アプリケーションである電子動力学シミュレーションコードを実装，支配的となるステンシル計算の最適化を中心に行い Xeon Phi クラスタの性能評価を行っている．実行方法は，Xeon Phi を 1 台の計算ノードとしてアプリケーションを実行する Native 実行，Xeon CPU と Xeon Phi 間で MPI を用いて協調計算を行う Symmetric 実行を用いた．前者は KNL クラスタへの移行を見据え，後者は現在の KNC クラスタを有効利用するための性能評価と捉えることができる．

我々は，先行研究 [1] において，コンパイラの自動ベクトル化を活用したステンシル計算の最適化および，時間発展計算で計算量を調整した Symmetric 実行の性能評価を行った．本研究では，自動ベクトル化に加え，Intrinsics を用いた手動ベクトル化の実装も行い，Xeon Phi でのステンシル計算性能を更に向上させる．また，AVX2 を採用した Haswell プロセッサ，ARTED のオリジナルターゲットである京コンピュータの SPARC64 VIIIfx プロセッサとも比較を行う．最後に，時間発展計算について京コンピュータとの比較，COMA 全体の 2/3 を用いた実シミュレーションの評価を示す．

[1] 筑波大学大学院 システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
[2] 筑波大学大学院 数理物質科学研究科
Graduate School of Pure and Applied Sciences, University of Tsukuba
[3] 筑波大学 計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[a)] hirokawa@hpcs.cs.tsukuba.ac.jp

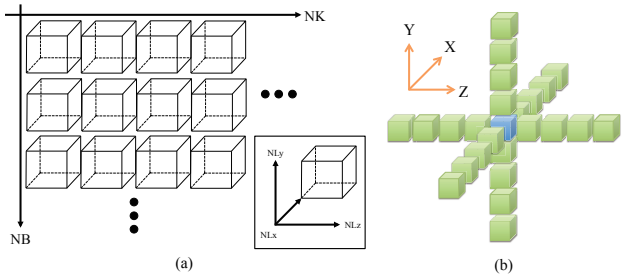

図 1 (a) マクロ格子点の計算領域のイメージ，(b) 25 点ステンシル計算のメモリアクセスパターン．

2. ARTED: 電子動力学シミュレータ

ARTED (Ab-initio Real-Time Electron Dynamics simulator) は，筑波大学計算科学研究センターにて開発されている実時間密度汎関数理論に基づくマルチスケール電子動力学シミュレータである [2]．時間依存密度汎関数理論の基礎方程式である時間依存 Kohn-Sham 方程式を実時間・実空間法を用いて解き，非常に短いパルス光と物質の相互作用をシミュレーションする．

ARTED は RSDFT (Real-Space Density Functional Theory) [3] と同様の方法を用いて基底状態を求め，時間発展計算を行い電子の時間変化を計算する．この際，時間依存 Kohn-Sham 方程式から電子の波動関数を導出し，波動関数に対して時間発展計算を行っている．RSDFT が，1000 から 10 万原子といった大規模な系を対象としているのに対し，ARTED は 10 から 100 原子程度の小規模なセルを非常に多くの個数計算する必要がある．したがって，ARTED は RSDFT と同様の計算が行われるが，時間発展計算が大部分を占めるため基底状態を求める計算時間は非常に短いものとなる．電子の波動関数は 4 次のテイラー展開で計算され，波動関数のハミルトニアンを計算する際にステンシル計算が必要となる．時間発展計算はおおよそ 1 万から 10 万ステップ行われるため，ARTED では時間発展計算が支配的であり，その大部分はステンシル計算に費やされる．RSDFT は実空間を分割し，MPI で並列計算を行うため隣接する MPI プロセス間で袖領域交換が必要となり，通信の隠蔽が大きな課題となっている．ARTED では，実空間ではなくより大きな並列空間である波数空間を分割し並列化を行う．波数空間の分割によって袖領域の交換が不要な代わりに，実空間について `MPI_Allreduce` 通信を行う必要がある．実空間のサイズは RSDFT に対し非常に小さいため通信コストが低く，大規模並列システム向けのアプリケーションであると言える．

ARTED は，電子の波動関数を表現するために，計算領域は下記の 4 つのパラメータで構成されている．

- マクロ空間格子点数 (NZ)
- Block wave number k (NK)
- Wave band (NB)
- 3 次元空間格子点 (NLx, NLy, NLz)

波動関数は，3 次元空間格子点を 1 次元 (NL = NLx×NLy×NLz) で示し，(NZ, NK, NB, NL) の配列で表現される．シミュレーションの際には，マクロ空間格子点が複数個設定されるが，1 個のマクロ空間格子点を計算する MPI プロセス数は固定される．したがって，Strong Scaling の性能評価では NZ= 1 とし，1 個のマクロ空間格子点の計算性能を評価する．性能評価では，入力データにシリコンを用い，パラメータを (NK, NB, NL) = (24^3, 16, $16 \times 16 \times 16$) と設定する．時間発展計算では，`MPI_Allreduce` が唯一の通信となり，最大でサイズ NL の倍精度浮動小数点数ベクトルの総和を行う．1 個のマクロ空間格子点内の MPI プロセス間での通信と，マクロ空間格子点間 (全 MPI プロセス間) での通信が必要となるが，全て `MPI_Allreduce` 通信である．Weak Scaling では，マクロ空間格子点を増やすことでその性能評価を行う．ARTED では，MPI と OpenMP によるハイブリッド並列化が行われており，NK が MPI によって各 MPI プロセスに分散される．NP を MPI プロセス数とすると，各 MPI プロセスは NK/NP×NB 個の 3 次元空間格子点を OpenMP を用いて並列に計算する．**図 1**-(a) に，本アプリケーションの時間発展計算時の並列化イメージを示す．各小領域 (NL) は独立に存在しており，またサイズは 16^3 と非常に小さいため，ステンシル計算は各 OpenMP スレッドが独立かつ逐次的に行う．

計算領域の波動関数配列は，倍精度複素数で表現され周期境界条件による 25 点ステンシル計算が行われる．図 1-(b) に，25 点ステンシル計算のメモリアクセスパターンを示す．非常にメモリバンド幅律速な問題となるが，次に示す通り一般的なステンシル計算とは異なる．本研究でのステンシル計算は，ハミルトニアン計算に用いられているため，1 回の時間発展で 1 個の小領域に対し 4 回のステンシル計算が必要となる．ハミルトニアン計算はステンシル計算と擬ポテンシャル計算で構成され [2]，擬ポテンシャル計算も同様に 4 回行われる．前述のとおり，1 個の小領域の計算は OpenMP の 1 スレッドで行われるため，各スレッドは 1 回の時間発展で 4 回のステンシル計算が含まれるハミルトニアン計算を逐次的に行い，複数個の小領域を計算する．各空間は独立，閉じた空間のため 1 回の時間発展で行われる 4 回のステンシル計算において OpenMP のスレッド同期または MPI による通信が発生しない．

3. 評価環境

本研究の評価は，筑波大学計算科学研究センターの Xeon Phi クラスタである COMA を用いる [4]．システムは 393 台のノードで構成され，各ノードには 2 台の Xeon Phi が接続されている．Xeon Phi の理論ピーク演算性能は 1074

表 1　本研究の評価環境 (COMA クラスタ).

# of Node	256 (System total : 393)
CPU	Intel E5-2670v2 2.5 GHz ×2 sockets
# of Cores	20 (10 ×2 sockets) / Node (disable HT)
Memory	64 GB (CPU) + 8GB ×2 (Xeon Phi)
Xeon Phi	7110P ×2 / Node
Interconnect	InfiniBand FDR (Mellanox Connect-X3)
OS	CentOS release 6.4
Software	Intel 15.0.2, Intel MPI 5.1.1
MPSS	3.4.2
OFED	1.5.4-1

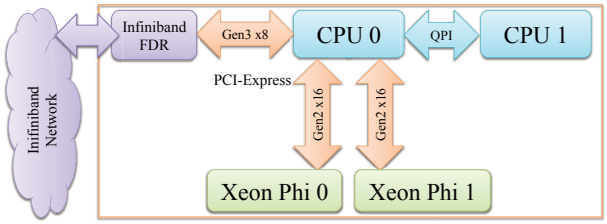

図 2　COMA のノード構成.

GFLOPS で，ノードあたり約 2.5 TFLOPS の理論ピーク演算性能となる．COMA の諸元について**表 1** に示し，ノード構成について**図 2** に示す．PCIe デバイス間通信では，Intel QPI (QuickPath Interconnect) を経由した場合に通信性能が大幅に低下することが知られている [5]．そのため，COMA の各ノードには 2 台の Ivy-Bridge Xeon CPU が接続されているが，Xeon Phi および InfiniBand HCA はすべて CPU-0 に接続し，この問題を回避している．

本研究では，CPU only 実行 (以下 CPU 実行)，Native 実行，Symmetric 実行の 3 つについて性能評価を行う．CPU 実行では，CPU の各ソケットに対し MPI プロセスを 1 個割り当て，各ノードあたり 2 個の MPI プロセスが割り当てられる．COMA の Xeon CPU は Intel Hyper Threading (Simultaneous Multithreading) が無効になっているため，各プロセスの OpenMP スレッド数は物理コア数と等しく 10 となる．Native 実行では，各 Xeon Phi に対し 1 個の MPI プロセスを割り当て，CPU 実行と同じく各ノードあたり 2 個の MPI プロセスが割り当てられる．OpenMP のスレッド数は，コア数の倍数となるように 60, 120, 180, 240 の 4 種類を評価する．一般的には，ステンシル計算はメモリバンド幅律速のため OpenMP スレッド数は物理コア数と一致することが望ましい場合が多い [6]．本研究のステンシル計算は，1 つのスレッドで 1 個の閉じた領域を計算するため，ステンシル計算は OpenMP での並列タスクといえる．Symmetric 実行では CPU 実行及び Native 実行を組み合わせ，各ノードに 4 個の MPI プロセスが割り当てられる．したがって，計算ノード単位では Symmetric 実行は CPU 実行に対して 2 倍以上の計算リソースを持つことになる．

```
integer,intent(in)     :: NL
integer,intent(in)     :: IDX(4,0:NL-1)
integer,intent(in)     :: IDY(4,0:NL-1)
integer,intent(in)     :: IDZ(4,0:NL-1)
real(8),intent(in)     :: A,B(0:NL-1)
real(8),intent(in)     :: Cx(4),Cy(4),Cz(4)
real(8),intent(in)     :: Dx(4),Dy(4),Dz(4)
complex(8),intent(in) :: E(0:NL-1)
complex(8),intent(out):: F(0:NL-1)
complex(8),parameter   :: zI = (0.d0, 1.d0)
integer     :: i
complex(8) :: v(3),w(3)

do i=0,NL-1
  ! x computation
  v(1)=Cx(1)*(E(IDX(1,i))+E(IDX(-1,i)))&
  &    +Cx(2)*(E(IDX(2,i))+E(IDX(-2,i)))&
  &    +Cx(3)*(E(IDX(3,i))+E(IDX(-3,i)))&
  &    +Cx(4)*(E(IDX(4,i))+E(IDX(-4,i)))
  w(1)=Dx(1)*(E(IDX(1,i))-E(IDX(-1,i)))&
  &    +Dx(2)*(E(IDX(2,i))-E(IDX(-2,i)))&
  &    +Dx(3)*(E(IDX(3,i))-E(IDX(-3,i)))&
  &    +Dx(4)*(E(IDX(4,i))-E(IDX(-4,i)))
  ! y computation
  ! z computation
  ! store
  F(i) = B(i)*E(i) + A*E(i)      &
  &     - 0.5d0*(v(1)+v(2)+v(3)) &
  &     - zI*(w(1)+w(2)+w(3))
end do
```

図 3　ステンシル計算のオリジナル実装.

4. ステンシル計算の最適化

本章では，ARTED で行っている波動関数のハミルトニアン計算で行われるステンシル計算についての最適化を行う．オリジナルの実装では，ステンシル計算を含むハミルトニアンの全計算が同じソースファイルに記述されている．そこで，まずステンシル計算のみ別のソースファイルとして抜き出し，コンパイラによる余計な最適化が行われないようにした．

図 3 に，オリジナルの実装を示す．ただし，Fortran では一般的に配列のインデックスが 1 で始まる 1-origin だが，最適化のため 0 始まりの 0-origin で受け取っている．ステンシル計算は，配列 v，w には係数が異なる近傍点の加減算結果が各次元毎に格納されており，各次元で 2 つのベクトル計算が行われている．また，周期境界領域のインデックス計算を省略するため，間接参照配列 IDX，IDY，IDZ に予め計算した近傍点のインデックスが格納されている．必要な演算を洗い出すと，倍精度複素数の加減乗算，倍精度浮動小数点数と倍精度複素数の乗算がこの計算では必要となる．

```
real(8),    intent(in) :: B(0:NLz-1,0:NLy-1,0:NLx-1)
complex(8),intent(in) :: E(0:NLz-1,0:NLy-1,0:NLx-1)
complex(8),intent(out):: F(0:NLz-1,0:NLy-1,0:NLx-1)
integer     :: ix,iy,iz
complex(8) :: v,w

#define IDX(dt) iz,iy,iand(ix+(dt)+NLx,NLx-1)
#define IDY(dt) iz,iand(iy+(dt)+NLy,NLy-1),ix
#define IDZ(dt) iand(iz+(dt)+NLz,NLz-1),iy,ix

do ix=0,NLx-1
do iy=0,NLy-1
!dir$ vector nontemporal(F)
do iz=0,NLz-1
  ! z computation
  v=(Cz(1)*(E(IDZ(1))+E(IDZ(-1))) &
  & +Cz(2)*(E(IDZ(2))+E(IDZ(-2))) &
  & +Cz(2)*(E(IDZ(3))+E(IDZ(-3))) &
  & +Cz(4)*(E(IDZ(4))+E(IDZ(-4))))
  ! y computation
  ! x computation
  ! store
  F(iz,iy,ix) = B(iz,iy,ix)*E(iz,iy,ix) &
  &           + A*E(iz,iy,ix) - 0.5d0*v - zI*w
end do
end do
end do
```

図 4 コンパイラによる自動ベクトル化に最適化したステンシル計算の実装.

4.1 コンパイラによる自動ベクトル化

まず，コンパイラによるベクトル化を考慮したコード修正 (Compiler Vectorization) を行う．コンパイラによるベクトル化は，あくまでも記述されたコードを基に最適な計算方法を推測し最適化を行うため，記述によってはベクトル化が複雑化し期待した性能が得られていない可能性がある．図 3 の通り，計算は各次元で別々に行われており，長さ 4 のベクトル計算が計 6 回行われていることになる．Xeon Phi が持つ 512bit SIMD では，倍精度複素数だと長さ 4 のベクトル長となり今回計算には都合の良い状態となっている．近傍点のアクセスに用いている間接参照配列 IDX，IDY，IDZ により，コンパイラは近傍点がメモリ上のどの位置に存在するか把握できず，ロードを効率的に行えていない可能性がある．また，24 点分の近傍点のアクセスを空間サイズの回数を行う必要があるため，4 Byte 整数配列とするとステンシル計算中でメモリから 96×NL [KB] の追加読み込みが必要となる．メモリアクセスの最適化と，キャッシュメモリの有効利用のため間接参照配列を除去する．

以上より，コンパイラによる自動ベクトル化を考慮した実装を行う．本来は 3 次元空間を NL の 1 次元配列として確保しているのを，カーネル中では (NLx, NLy, NLz) の 3 次元配列として扱うように変更した．この変更で，ステンシル計算時にインデックス計算が簡素化されるため，コンパイラはメモリアクセスパターンを把握しやすくなると予測される．また，その際に X, Y, Z の順に行っていた計算を，メモリ上距離が近い順となる Z, Y, X の順に計算するように変更した．NL は 16^3 で，データサイズにすると 64KB となり，タイリングを行って計算領域を L1 キャッシュに合わせて計算することも可能だが，2 次元タイリングの効果がなかったため使用していない．L2 キャッシュは各コアで 512KB 利用できる．各点に対する係数 B のサイズを考慮しても，各スレッドで 96 KB をメモリからロードするため，4 スレッド/コアでの実行でも計算領域は L2 キャッシュに収まる．この時点で L1 キャッシュにタイリングすると，オーバーヘッドが大きくなり性能低下につながっていると考えられる．前節で述べたとおり，F の各要素には一度の書き込みしか行わず，書き込んだデータを利用しない．通常のストア命令では，キャッシュに F のデータが保存されてしまうが，F のデータを利用しないためキャッシュを無駄にしている．そこで，キャッシュを経由せず直接書き込む non-temporal store を利用するようにコンパイラに対してヒントを与えた [7]．また，計算領域のサイズが全次元で 2 のべき乗のため，キャッシュスラッシングが多発する可能性がある．Z 次元のパディングを行った場合，ベクトル化時にループサイズがベクトル長で割り切れなくなってしまうため効率的なベクトル計算が難しくなる．またアドレスが 64 Byte 境界からずれてしまうため，ペナルティが大きい．したがって，Y 次元についてパディングを行い，キャッシュスラッシングを回避する．

本研究のステンシル計算は，周期境界条件を用いているため，ループ中でインデックス計算を行う場合には剰余を計算する mod 命令が必要となる．ステンシル計算の各次元サイズは，本研究で用いたパラメータでは 2 べきの 16 のため論理積演算で代替可能である．コンパイラは，インデックス計算中の mod もベクトル化して計算するが，mod は Short Vector Math Library (SVML) で実装されている．論理積演算はアセンブラ命令として実装されているため，mod よりも圧倒的に高速である．しかしながら，論理積演算では，被除数が 2 のべき乗である必要がある．格子サイズを増やして 2 のべき乗にすることも可能だが，領域サイズが飛躍的に増大し Xeon Phi での実シミュレーションが制限されてしまう可能性が高い．そこで，各次元で剰余テーブルを用意し，インデックス計算を省略するように実装した．インデックス計算を行う場合よりも余分なメモリを幾つか消費するが，間接参照配列を用いる場合よりもメモリ消費量は非常に少ない．

4.2 Intrinsics を用いた手動ベクトル化

次に，Knights Corner の Initial Many-Core Instructions (IMCI) を用いた明示的なベクトル化 (Explicit Vectoriza-

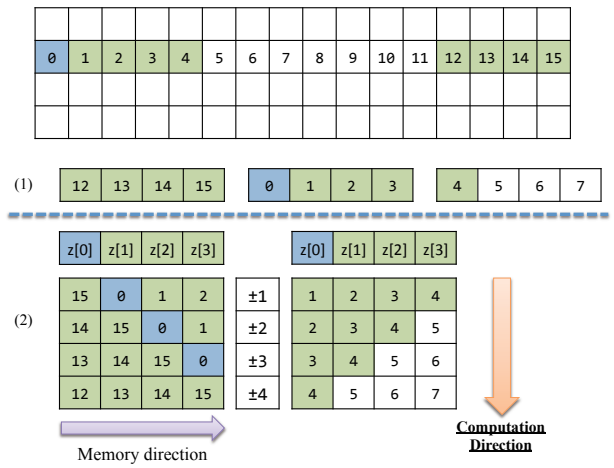

図 5 ベクトル演算での Z 次元の計算.

```
double complex const* e = E[ix*NLy*NLz + iy*NLz];

for(iz = 0 ; iz < NLz ; iz += 4) {
  __m512i v0, v1, v2;
  __m512d m[4], p[4];

  /* (1) */
  v0 = _mm512_load_pd(e + ((iz-4+NLz) % NLz));
  v1 = _mm512_load_pd(e + iz);
  v2 = _mm512_load_pd(e + ((iz+4+NLz) % NLz));

  /* (2) */
  m[0] = (__m512d)_mm512_alignr_epi32(v1,v0,12);
  m[1] = (__m512d)_mm512_alignr_epi32(v1,v0, 8);
  m[2] = (__m512d)_mm512_alignr_epi32(v1,v0, 4);
  m[3] = (__m512d)v0;
  p[0] = (__m512d)_mm512_alignr_epi32(v2,v1, 4);
  p[1] = (__m512d)_mm512_alignr_epi32(v2,v1, 8);
  p[2] = (__m512d)_mm512_alignr_epi32(v2,v1,12);
  p[3] = (__m512d)v2;
}
```

図 6 Z 次元のメモリアクセスを Intrinsics で実装したコード.

tion) を行う．本研究では，連続なメモリアクセスとなる Z 次元について格子サイズをベクトル長である 4 の倍数とし，下記の最適化を行った．

- 非アラインメモリアクセスの最適化
- 倍精度複素数積の最適化

また，ベクトル長に基き，4 点を同時に更新する．連続方向の次元がベクトル長の倍数であるため，全ての格子点のストアに対し non-temporal store 命令を発行できる．

4.2.1 非アラインメモリアクセスの最適化

連続方向となっている Z 次元の計算では，必ず非アラインメモリアクセスが発生するため，コンパイラは `gather` 命令を発行しデータを集める．`gather` 命令を回避するため，`alignr` 命令を用いて計算に必要なデータを揃える．`alignr` 命令は 2 つの 512-bit ベクトルを連結し 1024-bit ベクトルとし，32-bit 単位で右論理シフトを行う．シフト後，下位 512-bit を出力として返す．IMCI では，メモリのアラインメントが不要な `load` 命令もあるが，アラインをまたがないように 2 命令発行する必要がある．Z 次元のメモリアクセスを，ベクトル命令で実装すると**図 5** となる．これは Z 次元のサイズ (NLz) が 4 の倍数であれば正常に動作する．(1) まず，ロード命令を用いて更新対象の 4 点，その次の 4 点，後ろ 4 点をそれぞれベクトルレジスタにロードし，(2) `alignr` 命令を用いて，各点の更新に必要な近傍点ベクトルを生成する．(3) 各ベクトルの乗加算を行う．この計算で得られた結果が，評価点 4 点分の Z 次元の計算結果となる．`alignr` 命令で生成したベクトルは，それぞれ更新点 $\pm$ N で 1 つのベクトルを構成している．そのため，マイナス方向とプラス方向の近傍点で 4×4 の正方行列が 2 つできる形となる．このようにすると，ベクトル演算はメモリ方向に対して垂直方向に演算を行い，水平方向への演算が必要ない．ベクトルの水平方向加算は，AVX などでは実装されているが，IMCI には実装されておらず，また AVX の水平加算は演算コストが高い．このメモリアクセスを，Intrinsics を用いて実装すると**図 6** となる．`alignr` 命令の 32-bit シフト回数は即値で，コンパイル時には値が確定している必要がある．図 6 では，剰余を用いているが，実際には剰余テーブルを用いている．

4.2.2 倍精度複素数積の最適化

手動ベクトル化でもう 1 つ問題となるのは，倍精度複素数積である．図 3 では，配列 `w` と定数 `zI` で積が必要となる．IMCI では，Intel SSE/AVX が提供する命令セットの中で，複素数を高速に計算するための命令が実装されていない [8]. Xeon Phi で複素数演算を行う際には，masked 命令を使って実部あるいは虚部のみを計算し，ベクトル演算ユニットの半分が使われないといった状況が発生する．これは Intel コンパイラによる自動ベクトル化，組み込み関数を用いたハンドコーディングによるベクトル化の両方で問題となる．

この問題から，Xeon Phi 上の複素数演算で高い性能を得るのは実数演算の場合よりも困難であると考えられる．しかしながら，本研究でのコードの場合，定数 `zI`$= (0, -i)$ との積であるため式を展開して計算した．複素数積 $(a, bi)(0, -i)$ を展開すると $(b, -ai)$ となり，(1) 実部と虚部を入れ替える，(2) 虚部の符号を反転させる，の 2 ステップで計算可能となる．

5. 性能評価

5.1 ステンシル計算性能

5.1.1 演算性能比較

COMA の Ivy-Bridge CPU および Xeon Phi に加え，Haswell CPU，京コンピュータの SPARC64 VIIIfx プロセッサ [9] との性能比較を行う．各プロセッサの諸元を**表 2** に示す．本研究のターゲットシステムは，Ivy-Bridge

表 2 各プロセッサ諸元.

	Haswell	Ivy-Bridge	KNC	SPARC64
CPU	E5-2670v3	E5-2670v2	Xeon Phi 7110P	SPARC64 VIIIfx
Core	2.3 GHz ×12	2.3 GHz ×10	1.1 GHz ×60	2.0 GHz ×8
AVX base clock	2.0 GHz	–		
L1 Data Cache/Core	32 KB			
L2 Cache	256 KB/Core		512KB/Core	6MB
L3 Cache	30 MB	25 MB	–	
Instruction Set	AVX2	AVX	IMCI	HPC-ACE
FMA unit	2	0	1	2
FLOP/Cycle	16	4	16	8
GFLOPS	384	200	1074	128
Memory Bandwidth	68 GB/s	59.7 GB/s	352 GB/s	64 GB/s
Byte/FLOP	0.177	0.2985	0.3277	0.5
Compiler	Intel 16.0.0		Intel 15.0.2	Fujitsu K 1.2.0-19

表 3 各プロセッサでのステンシル計算性能とピーク性能比.

Processor	Vectorization	GFLOPS	ピーク性能比 [%]
KNC	Original	29.0	2.70
	Compiler	132.2	12.30
	Explicit	212.2	19.75
Ivy-B	Original	26.2	13.08
	Compiler	112.3	56.14
	Explicit	114.6	57.32
Haswell	Original	55.2	14.38
	Compiler	170.7	44.46
	Explicit	145.0	37.75
SPARC64	Original	15.9	12.42
	Compiler	26.8	20.93

プロセッサを搭載した COMA クラスタである．しかし Haswell では 2 基の FMA 演算ユニットが追加され，命令セットが AVX2 となり 16 FLOP/Cycle と性能が大幅に向上したため，Haswell との比較は重要である．コンパイラについては，Haswell, Ivy-Bridge では Intel Compiler 16.0.0 を使用し，Xeon Phi では Intel Compiler 15.0.2, SPARC64 VIIIfx では Fujitsu K-1.2.0-19 を用いた．また最適化オプションは，Intel CPU には`-O3 -restrict -ansi-alias -fno-alias` を設定し，Xeon Phi は加えて`-opt-assume-safe-padding` を設定した．SPARC64 VIIIfx は，`-O3 -Kfast,ocl` を設定した．Xeon CPU の手動ベクトル化は，Xeon Phi のコードを基に実装した．Haswell では，FMA 命令への積極的な置換により最適化を行った．自動ベクトル化コードについては，Ivy-Bridge と同じ実装を用い，コードの修正は行わない．SPARC64 VIIIfx では，Xeon CPU 向けに最適化した自動ベクトル化コードをほぼそのまま利用している．Intel コンパイラ用の最適化ディレクティブを，Fujitsu コンパイラ用のものに入れ替える以外に大きな変更点はない．

各プロセッサのステンシル計算の性能について**表 3** に示す．Ivy-Bridge と比較すると，Haswell はコードを修正していない自動ベクトル化コードでもおよそ 1.75 倍ステンシル計算の性能が向上している．また自動ベクトル化コードの性能が高く，Haswell 用に再実装した手動ベクトル化コードに対し約 1.17 倍の性能となっている．一方 Xeon Phi の場合，約 1.6 倍程度，手動ベクトル化コードが高速である．Intel コンパイラは，SSE と AVX，128/256-bit 幅でのベクトル演算については非常に高レベルな最適化が行われていると考えられるが，512-bit 幅のベクトル演算については，IMCI の機能が不足していることも関係するが，効率の高いベクトル化を提供できていないのではないかと考えられる．Haswell の場合，演算性能自体は Ivy-Bridge に比べ向上したが，Byte/FLOP が Ivy-Bridge に比べ低くピーク性能比は低下する傾向にある．また，Ivy-Bridge と Haswell のピーク性能比が高くなっているが，各計算領域である空間格子点のサイズは L2 キャッシュに収まる程度であるため，次に計算する領域が L3 キャッシュには入っているためと考えられる．SPARC64 VIIIfx の Byte/FLOP 値は 0.5 で，メモリバンド幅律速であるステンシル計算性能で最も良い性能が得られると考えられる．京コンピュータ上のプロファイラで確認すると，最適化後の L1 キャッシュミスはロードストア比で 7%から 2%まで削減され，キャッシュメモリのアクセス待ち時間は約 1/7 まで減少している．しかしながら，ピーク性能比で見ると 20% 程度しか得られておらず，Xeon Phi とほぼ同等で非常に低い．

表 4 手動ベクトル化コードでの最適化の効果.

	Complex Mult.	Unit-Stride Load	Rel. Perf
(a)	Expand	`load+alignr`	155.45 %
(b)	Natural	`load+alignr`	152.55 %
(c)	Expand	`gather`	103.47 %
(d)	Natural	`gather`	102.76 %
(e)	Compiler Vectorization		100.00 %

5.1.2 Xeon Phi の手動ベクトル化コードの最適化

次に，Xeon Phi 向け手動ベクトル化コードの最適化の

効果について述べる．Xeon Phi では，倍精度複素数積と連続方向のメモリアクセス最適化を行った．**表 4** に，それぞれの効果について示している．倍精度複素数積は，展開しなかった場合に 2 %程度の性能低下にとどまった．これは，倍精度複素数積が各点の更新に一度しか必要とせず，大きな影響を与えなかったと考えられる．連続方向のメモリアクセス最適化は，適用せず `gather` 命令を用いた場合には Compiler Vectorization とほぼ同等性能まで低下した．`gather` 命令のコストが非常に高く，`load` と `alignr` 命令の組み合わせにより大幅に性能が向上したと言える．

ARTED のステンシル計算は，テイラー展開を行った 4 次差分のため，同様の差分計算では同じようなメモリアクセスが必要である．この連続方向へのメモリアクセスの最適化は，他のアプリケーションにおけるステンシル計算にも応用可能と考えられる．

5.1.3 Knights Landing への移行について

Knights Landing では，Knights Corner とは異なり AVX-512 が SIMD 命令として提供される．したがって，本研究で実装した手動ベクトル化コードの実装を修正する必要がある．しかしながら，IMCI と AVX-512 には不完全ではあるが下位互換性があり，ほとんどの命令は同じフォーマットで利用できる．

本研究では，四則演算命令を除外すると 13 命令を使用しており，そのうち 3 命令が IMCI と AVX-512 でフォーマットが異なる．修正は，単純な命令置換，または数行のインライン化可能な関数レベルでの実装の置き換えにとどまっており，修正コストは非常に小さい．現在の IMCI コードをベースに，命令またはインライン関数レベルの置換を行い，Intel のエミュレータ [10] 上で動作することを確認している．また，AVX-512 では機能ごとに複数のサブ命令セットが用意されているが，本研究の実装では AVX-512 を提供する全プロセッサで利用可能な AVX-512F (Foundation) のみが必要で，AVX-512 のプロセッサであれば性能評価が可能である．

5.2 時間発展計算性能

5.2.1 CPU/Xeon Phi 間の負荷分散

時間発展計算では，CPU 実行と Native 実行，Symmetric 実行の性能評価を行う．Native 実行と Symmetric 実行では，Xeon Phi の OpenMP スレッド数を 60, 120, 180, 240 スレッドで評価し最速値を用いた．

ここで，MPI プロセス単位で比較した場合の Symmetric 実行の性能についての予測を行う．Symmetric 実行は CPU および Xeon Phi 両方を計算リソースとして用いるため，問題を CPU と Xeon Phi 間で均等に割り当てた場合，計算時間は全体通信によって性能が低い方に律速される．したがって，負荷分散を考慮し CPU と Xeon Phi の計算時間が同一となるように計算量を調整する必要がある．ステンシル計算の性能評価から，Xeon Phi に多く計算領域を割り当てることになる．

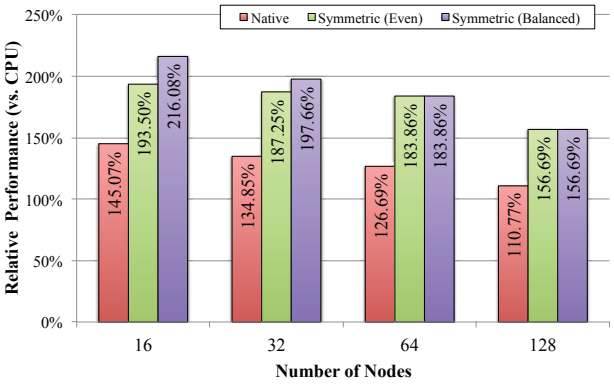

図 7 時間発展計算の Strong Scaling での性能評価．

本研究では，実空間のサイズよりも波数空間のサイズが非常に大きいため，波動関数の波数空間パラメータである Bloch wave number k (NK) を MPI で分散する．また，この並列化により NK 間では袖領域の交換ではなく，`MPI_Allreduce` を用いた総和演算を行うため，並列化は粗粒度かつ非常に容易である．そのため，NK の割当量を変更することがそのまま負荷分散を適用することになり，極めて容易に負荷分散を実現可能である．本評価では，性能に合わせて Xeon Phi と CPU の計算量を調整したものを “Symmetric (Balanced)” と記載し，均等割当は “Symmetric (Even)” と記載する．しかしながら，計算量調整により演算順序が変更され計算結果が変わる可能性がある．既に検証を行ったが，計算量調整によるシミュレーション結果への影響は確認されていない．

5.2.2 Strong Scaling

Strong Scaling の評価で，CPU 実行に対する相対性能を**図 7** に示す．利用する計算リソースから，Symmetric 実行は CPU 実行に対して 2 倍以上の性能が予測されるが，16 ノードの “Symmetric (Balanced)” のみ 2.16 倍の性能が得られた．計算リソースが 2 倍以上になったため，強スケーリング性が増加し Symmetric 実行での性能が得にくくなっているのではないかと考えられる．また別の原因として，Native 実行と Symmetric 実行での最適な Xeon Phi の OpenMP スレッド数が異なる問題が考えられる．Native 実行では 16 ノードと 32 ノード実行時には 240 スレッドが最適であったが，Symmetric 実行では全てのケースで 180 スレッドが最適であった．

Symmetric 実行は，計算ノード単位で見た場合には CPU 実行に対し計算リソースが 2 倍以上となる．したがって，CPU 実行と同等の MPI プロセスで計算を行う場合，必要な計算ノード数は半分となる．本研究では，16 ノードまでは $N \times 2$ 台の計算ノードを確保し CPU 実行で計算するよりも，N 台の計算ノードを確保して Symmetric 実行で計

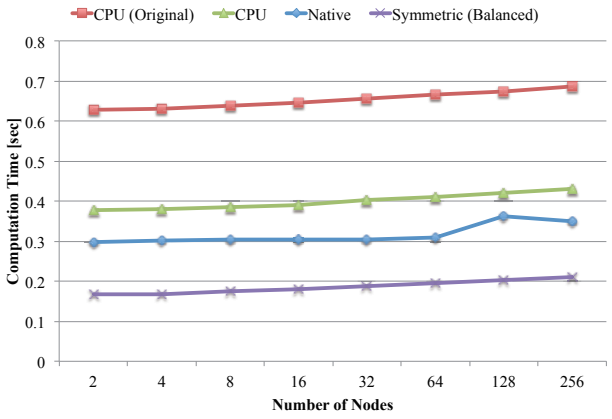

図 8 時間発展計算の Weak Scaling での性能評価.

算する方が高速となっている.

5.2.3 Weak Scaling

時間発展計算の 1 反復あたりの，Weak Scaling による実行時間を**図 8** に示す．前節では，マクロ格子点 (NZ) 1 個の計算性能について評価を行ったが，Weak Scaling では NZ を 2 以上に設定し性能評価を行う．ARTED は，本評価で用いているシリコンが持つダイヤモンド構造を利用し計算量を 1/8 まで減らすことが可能で，2 ノード (4 Xeon Phi) で 1 個のマクロ格子点を計算可能である．CPU, Native 実行では，2 ノード (4 MPI プロセス) で 1 個のマクロ格子点を計算し，Symmetric 実行では，2 ノード (8 MPI プロセス = CPU + Xeon Phi) で 1 個のマクロ格子点を計算する．すなわち，図 7 中の 16 ノード時の性能で，Weak Scaling の性能評価を行うことと同義である．また，Native, Symmetric 実行共に Xeon Phi は 180 スレッドで計算している．特に Xeon Phi では，240 スレッドで計算した場合に通信時間が増大する問題が発生しており，原因は調査中である.

図 8 から，Weak Scaling は概ね達成できている．Native 実行では，128 ノード実行時に実行時間が増加しており，`MPI_Allreduce` のプロトコルスイッチが入っている可能性がある．しかしながら，実シミュレーションの際は Native 実行ではなく，Symmetric 実行で全ての計算リソースを用いて計算すると考えられる．Symmetric 実行では，Native 実行のような問題は発生しておらず，CPU-only 実行と同様のスケーリングを達成している．また，ここでは最大 256 ノードで評価を行っているため，NZ は最大 128 であるが，実シミュレーションでは最大 100 程度を NZ に設定する．したがって，アプリケーションが要求する計算サイズを COMA クラスタで十分計算可能であるといえる.

5.2.4 京コンピュータとの比較

京では，各ノードに 1 台の CPU が接続されているため，1 ノードあたり 1 MPI プロセスを割り当て，OpenMP スレッド数は 8 となる．MPI プロセス数を一致させる場合，

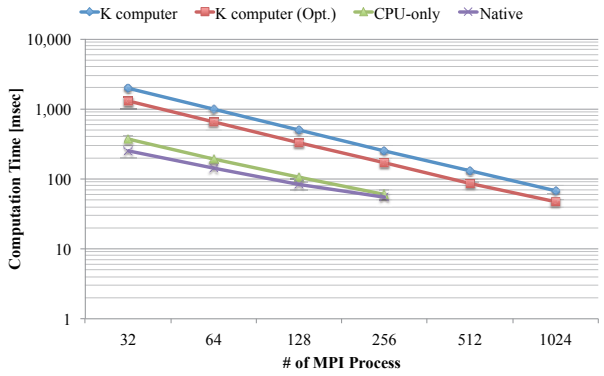

図 9 京コンピュータとの比較，時間発展 1 反復あたりの計算時間.

COMA では N ノード利用し，京では $N \times 2$ ノードを用いて実行する．すなわち，MPI プロセス数は CPU ソケット数または Xeon Phi 台数と等しい.

Strong Scaling での京の時間発展 1 反復あたりの計算時間を，**図 9** に示す．オリジナルコードによる実行に比べ，COMA の Xeon CPU 向けコードを実行した “K Computer (Opt.)” は約 1.5 倍程度高速である．“K Computer (Opt.)” に対し “CPU-only” は最大 3 倍程度高速で，Xeon Phi と比較すると最大 4.5 倍程度の性能差となっている．ここから，京で N ノードを用いて計算する場合，COMA では $N/6$ 程度の計算ノードの Xeon CPU を用いて計算するとほぼ同等性能が得られるということが言える．Xeon Phi は，$N/9$ 程度の計算ノードでほぼ同等性能が得られる．Xeon CPU 向けの最適化により性能低下している可能性もあるが，少なくとも元実装よりも 1.5 倍高速化されている．また，ステンシル計算よりも性能差が広がっており，利用しているノード数の差が影響していると考えられるが，現在調査中である．京の場合，1024 ノードまで理想的な線形スケールを達成しているが，CPU-only や Native 実行に比べ実行時間が長くなっている．Native 実行は，Xeon Phi 256 台で既に飽和状態になっており，やはり高い並列性によって Strong Scaling の達成が困難であると考えられる.

6. まとめ

本研究では，電子動力学シミュレーションコードを Xeon Phi 上に実装し，支配的な計算となっているステンシル計算の最適化を行った．コンパイラによる自動ベクトル化のためにコードを修正し，計算領域の配列の次元と計算ループの変換，non-temporal store の利用，計算順序の変更などを行うことで 132.2 GFLOPS と元の 29.0 GFLOPS から大幅に改善した．手動ベクトル化では，近傍点へのアクセスの最適化，倍精度複素数積の省略などにより最終的に 212.2 GFLOPS の演算性能が得られた．Ivy-Bridge でも同様の最適化を行い，手動ベクトル化コードで 106.9 GFLOPS の演算性能が得られた．また，Haswell プロセッサでは自動ベクトル化によって 170.7 GFLOPS の演算性能が得られ

ている．

時間発展計算全体では，CPU 実行に対し Native 実行は 16 ノード実行時に約 1.45 倍の性能を達成し，CPU 実行よりも高い性能が得られている．Symmetric 実行は，Xeon Phi と CPU 間の計算量を調整することにより N 台の計算ノードによる CPU-only 実行に対し，$N \div 2$ 台の計算ノードで 2 倍以上の性能を達成した．Weak Scaling の評価によって，アプリケーションが要求する計算サイズを Xeon Phi クラスタで十分計算可能であることを示した．また，元のターゲットシステムである京コンピュータでの評価も行い，Xeon CPU 向けに最適化したコードを変更なく実行することで，オリジナル実装に対し 1.5 倍の性能向上が得られた．しかしながら，ステンシル計算の実効性能が Xeon CPU に比べて低く，ソケットあたりの計算量を同等にして比較すると理論ピーク性能よりも大きな性能差となった．今後の課題として，ステンシル計算を用いているハミルトニアン計算全体の最適化と，Knights Landing での性能評価が挙げられる．

謝辞 本研究の評価環境は，筑波大学計算科学研究センターの平成 27 年度学際共同利用プログラム課題「時間依存密度汎関数理論によるパルス光と物質の相互作用」および HPCI の平成 27 年度「京」一般利用課題「極限的パルス光と物質の相互作用を記述するマルチスケール第一原理計算」による．本研究の性能評価にあたり東京大学情報基盤センターの塙敏博准教授には様々なご助言を頂きました．ここに感謝申し上げます．

参考文献

[1] 廣川 祐太，朴 泰祐，佐藤 駿丞，矢花 一浩：Xeon Phi クラスタにおける Symmetric 並列実行による電子動力学シミュレーションの性能評価，情報処理学会研究報告，Vol. 2015-HPC-151, No. 18 (2015).

[2] S. A. Sato and K. Yabana: Maxwell + TDDFT multiscale simulation for laser-matter interactions, *J. Adv. Simulat. Sci. Eng.*, Vol. 1, No. 1, pp. 98–110 (2014).

[3] Y. Hasegawa, J. Iwata, M. Tsuji and *et al.*: First-principles Calculations of Electron States of a Silicon Nanowire with 100,000 Atoms on the K Computer, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, ACM, (online), DOI: 10.1145/2063384.2063386 (2011).

[4] 筑波大学計算科学研究センター：スーパーコンピュータ COMA (PACS-IX) について，`http://www.ccs.tsukuba.ac.jp/files/coma-general/coma_outline.pdf`.

[5] 小田嶋 哲哉，塙 敏博，児玉 祐悦，朴 泰祐，村井 均，中尾 昌広，佐藤三久：HA-PACS/TCA における TCA および InfiniBand ハイブリッド通信，情報処理学会研究報告，Vol. 2014-HPC-147, No. 32 (2014).

[6] 松田 元彦，丸山 直也，滝沢 真一郎：Xeon Phi (Knights Corner) の性能特性とステンシル計算の評価，情報処理学会研究報告， Vol. 2014-HPC-143, No. 32 (2014).

[7] R. Krishnaiyer, E. Kultursay, P. Chawla, S. Preis, A. Zvezdin and H. Saito: Compiler-Based Data Prefetching and Streaming Non-temporal Store Generation for the Intel(R) Xeon Phi(TM) Coprocessor, *IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pp. 1575–1586 (2013).

[8] D. Takahashi: Implementation and Evaluation of Parallel FFT Using SIMD Instructions on Multi-core Processors, *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, IWIA 2007*, pp. 53–59 (2007).

[9] T. Maruyama: SPARC64(TM) VIIIfx: Fujitsu's New Generation Octo Core Processor for PETA Scale Computing, *Hot Chips: A Symposium on High Performance Chips 21* (2009).

[10] Intel Software Development Emulator: `https://software.intel.com/en-us/articles/intel-software-development-emulator`.