Vol.2015-HPC-151 No.18
2015/10/1

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

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ており，同プロセッサを用いた実アプリケーションの最適化や性能向上が求められている．本研究は，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 (Ab-initio Real-Time Electron Dynamics simulator) を Xeon Phi クラスタ上に実装し，その最適化を行った．同アプリケーションの主計算はステンシル計算だが，1 つの大領域を MPI と OpenMP を用いて並列計算するのではなく非常に多くの小領域を MPI で分散し，個々の小領域を OpenMP の 1 スレッドで逐次的に計算する必要がある．そのため，本研究ではステンシル計算のシングルスレッドレベル最適化を中心に行った．また，我々は実行方式として Xeon Phi を GPU と同様な方法で用いる Offload 実行ではなく，Xeon Phi を 1 台のノードとみなし計算を行う Native および Symmetric 実行に着目した．特に Symmetric 実行は CPU と Xeon Phi にそれぞれ MPI のプロセスを割り当て，ヘテロジニアス実行でアプリケーションを動作させるため，計算ノード上の全計算リソースを有効利用できると考えられる．本研究では，Native 実行で CPU に対し最大約 1.32 倍の性能向上を達成した．この結果，Xeon Phi を 1 台の Ivy-Bridge Xeon CPU 以上の性能を持つ計算ノードとみなすことが可能となり，問題を均等分割した Symmetric 実行では性能が低い CPU 側に律速されている．また，CPU と Xeon Phi 間の計算量を調整することで，CPU 実行に対し半分の台数の計算ノードで同程度以上の性能を得ることが可能となり，Symmetric 実行の有効性を示した．

1. はじめに

Intel Xeon Phi プロセッサは，MIC (Many Integrated Cores) アーキテクチャに基づくアクセラレータで，現在のプロダクトは KNC (Knights Corner) と呼ばれる．本研究では，Xeon Phi は現行プロダクトである KNC を指す．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なことである．

Xeon Phi には，Offload, Native, Symmetric と 3 種類の並列実行方法が用意されている．まず，Offload 実行は GPU のプログラミングモデルと同様で，オフロード対象のコードを切り出し，CPU と Xeon Phi 間で計算データのやりとりが必要となる．この実行方法には，2 つのモデルが用意されている．1 つ目は Explicit Offload で，ディレクティブを用いた方法である．一般的に，Offload 実行はこの Explicit Offload を指している．2 つ目は Implicit Offload で，C および C++言語で利用可能な Intel CilkPlus 拡張を用いるモデルである [1]．このモデルでは，CPU と Xeon Phi 間で明示的なデータのやりとりが不要となるが，1 つの大きな問題は CilkPlus が Fortran では利用できない点である．多くの計算科学アプリケーションは Fortran で実装されており，本研究の対象アプリケーションも Fortran90 で実装されている．もし Implicit Offload を用いる場合，Fortran で実装されたコードのほとんどを C, C++に書き換える必要があり現実的ではない．Native 実行は，アプリケーションをホスト CPU を介さずに Xeon Phi 上で実行する．この実行方法では，すべてが Xeon Phi 上で計算され基本的に CPU は用いられない．最後に，Symmetric 実行はアプリケーションを CPU と Xeon Phi 両方で実行し，

[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

表 1 各並列実行方法の比較.

	Offload	Native	Symmetric
Target Resources	Xeon Phi with CPU	Xeon Phi	Xeon Phi and CPU
Programming Model	Similar to GPU	Similar to CPU	
Implementation Cost	Relatively High	Low	Relatively High
Optimization Cost	High		
Communication	High	Low	Relatively Low

MPI を用いて協調計算を行う．この実行方法は，CPU と Xeon Phi の両方を用いるため理論的にはノードの全計算リソースを使い切れる．しかしながら，CPU と Xeon Phi のヘテロジニアス実行であるため，load-imbalance が大きな課題となりやすい．Xeon Phi は，比較的性能が低いコアを多数接続し，512bit 幅の SIMD 命令が十分利用できることを条件として高い理論演算性能を確保している．したがってこれらの実行方法すべてに共通するのは，演算性能を得るためには Xeon Phi への最適化が必須であり，Xeon Phi の高い並列性能を活用するためにより大きな問題でなければならないということである．

実行方法の比較を**表 1** に示す．計算リソースの活用という点でみると，Symmetric 実行は最良の選択肢だが，実アプリケーションを実装する際に 2 つの問題が考えられる．

- Xeon Phi に対する最適化をどのように行うか (すべての実行方法において共通)
- Xeon Phi と CPU 間での負荷分散の問題

本研究では，実アプリケーションを Xeon Phi 上に実装し，Xeon Phi クラスタ上にて Symmetric 実行の有効性を検証する．対象アプリケーションには，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の ARTED を用いる．同アプリケーションは，大規模システム向けに OpenMP と MPI のハイブリッド並列を用いて実装されている．

2. ARTED: 電子動力学シミュレータ

2.1 概要

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]．ARTED は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．

基となっている RSDFT (Real-Space Density Functional Theory) が電子の基底状態を計算するのに対して [3]，ARTED は RSDFT と同様の計算を用いて基底状態を求めた後，時間発展計算によって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 の対象が，1000 から 10 万原子といった大規模な

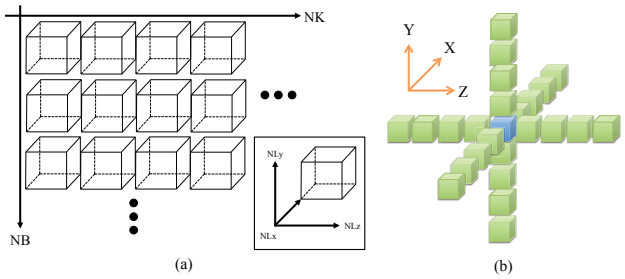

図 1 (a) マクロ格子点の計算領域のイメージ，(b) 25 点ステンシル計算のメモリアクセスパターン.

系であるのに対し，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 では 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．

近年の主な成果では，ARTED を利用したシミュレーションによりアト秒光科学実験で見られるシリコンのバンドギャップの超高速変化の特徴を再現し，電子の励起プロセスが量子トンネル効果により起きていることを明らかにしたものがある [4]．また，ARTED は京コンピュータ [5] における性能評価が 11,520 ノードまで行われており，大規模計算のための並列最適化は十分行われていると言える．

2.2 並列性と計算量

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)

波動関数の配列は (NZ, NK, NB, NLx, NLy, NLz) で表されるが，3 次元空間格子点は ARTED 上では NL で表される (NL = NLx × NLy × NLz)．また，本研究はマクロ空間格子点上の演算について強スケーリング性能の評価を行うため，常に NZ= 1 としている．したがって，本研究では波動関数の配列は (NK, NB, NL) と 3 つのパラメータ

で表される．性能評価では，シミュレーション対象の物質としてダイアモンド構造を持つシリコンを用い，パラメータを (NK, NB, NL) = (24^3, 16, 4096 = (16, 16, 16)) と設定する．実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．またマクロ空間格子点間では，非常に小さな通信は発生するが，時間発展計算全体から見れば無視できるレベルとなっている．したがって，ARTED では 1 個のマクロ空間格子点上の演算性能が重要となる．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．Byte/Flop 値は 20 となり，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特徴がある．本研究でのステンシル計算は，4 次のテイラー展開を行った式の中でハミルトニアンを計算する際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．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．また各空間は独立で存在しているため，OpenMP スレッド間でキャッシュは共有されない．そのため，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．また，ハミルトニアンの計算にはステンシル計算以外の計算も含まれるため純粋なメモリバンド幅律速の問題とはならず，実効メモリバンド幅に近い値を得るのは困難であると予想される．本研究では，各小領域において逐次的に行われるハミルトニアンの計算を Xeon Phi 向けに最適化できるかが大きな課題となる．

2.3 最適化および性能目標

我々は，既に [6] にて ARTED のステンシル計算の最適化を中心に性能評価を行っているが，Xeon Phi の時間発展計算全体の性能は CPU よりも低いものとなっていた．本稿でも，ステンシル計算を中心にさらなる最適化を行っている．

まず，時間発展計算において支配的となっている波動

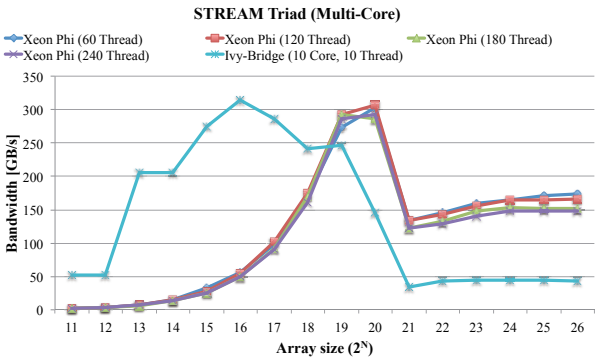

図 2 Ivy-Bridge (Xeon E5-2670v2) と Xeon Phi (7110P) の実効メモリバンド幅．

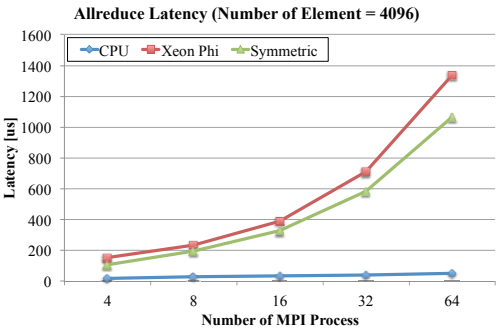

図 3 倍精度浮動小数点数ベクトルの `MPI_Allreduce` の通信性能．

関数のハミルトニアン計算を Xeon Phi 向けに最適化しなければならない．ここで，Ivy-Bridge Xeon CPU および Xeon Phi での実効メモリバンド幅を**図 2** に示す．実効メモリバンド幅では，Xeon Phi は Ivy-Bridge に対して約 3 倍のバンド幅を持つため，メモリ律速なステンシル計算では約 3 倍程度の性能が得られると期待できる．しかしながら，実アプリケーションであるためステンシル以外の計算も含まれる．ハミルトニアン計算中にも，メモリバンド幅律速でない計算が含まれるため，全体性能において実効メモリバンド幅に近い性能差を得るのは非常に困難が予想される．また，[7] では既に単精度浮動小数点数を対象とした Xeon Phi 上での組み込み関数を用いたステンシル計算の最適化が行われているが，本稿での計算は倍精度複素数が用いられる．後述するが，Xeon Phi では複素数に関連する組み込み関数が不足しており，ベクトルレジスタを完全に利用することは困難である．

時間発展計算中で発生する通信は `MPI_Allreduce` による倍精度浮動小数点数，および小領域のサイズ NL と同じ長さの倍精度浮動小数点数ベクトルの総和演算のみである．サイズ NL での `MPI_Allreduce` の通信性能を**図 3** に示す．Xeon Phi や Symmetric 実行での性能が CPU に対し非常に悪いが，通信時間のオーダはマイクロ秒からミリ秒オーダであるのに対し，計算時間はミリ秒から秒オーダであるため，計算最適化に注力すれば良い．Native 実行では，理論性能差から CPU 実行よりも高い性能となることが期待

表 2 本研究の評価環境 (COMA クラスタ).

Number of Node	128 (System total : 393)
CPU	Intel E5-2670v2 2.5 GHz ×2 sockets
Number of Cores	20 (10 cores ×2 sockets) / Node (disable Hyper Threading)
Memory	64 GB (DDR with CPU) + 8GB ×2 (Xeon Phi)
Xeon Phi	7110P ×2 / Node
Interconnect	InfiniBand FDR (Mellanox Connect-X3 (56 Gbit/s)) with OFED 1.5.4-1
Compiler and Software	Intel 15.0.2, Intel MPI 5.0.3 and Intel MKL 11.2.2
OS (Compute Node)	CentOS release 6.4
MPSS	3.4.2

表 3 各実行方法での MPI プロセスおよび OpenMP スレッドの割り当てルール.

	CPU	Native	Symmetric
MPI Process / Node	2 (each CPU sockets)	2 (each Xeon Phis)	4 (each CPU sockets and Xeon Phis)
OpenMP Threads / Process	10	60, 120, 180 or 240	10 (CPU). 60, 120, 180 or 240 (Xeon Phi)

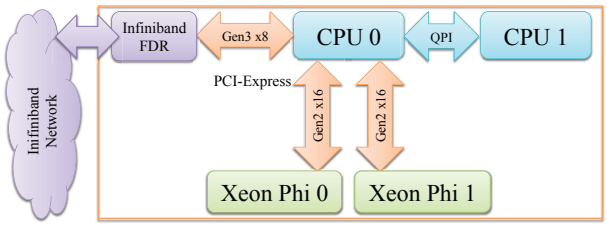

図 4 COMA のノード構成.

される．Symmetric 実行では，Native 実行の期待性能を踏まえ，2 つの指標が考えられる．1 つ目に MPI プロセス単位，計算リソースが CPU 実行と Native 実行に対し同等以上であった場合，Symmetric 実行の性能は CPU 実行性能に対し同等以上であることが期待される．2 つ目に計算ノード単位，計算リソースが CPU 実行と Native 実行に対し 2 倍以上であった場合，Symmetric 実行の性能は CPU 実行性能に対し 2 倍以上であることが期待される．

本研究での最適化および性能目標を以下にまとめる．

- ARTED で支配的な計算であるステンシル計算を含む波動関数のハミルトニアン計算を高速化する．
- Native 実行において，計算ノード単位で比べた際に CPU 実行以上の性能を達成する．
- Xeon Phi と CPU を併用し協調計算を行う Symmetric 実行において，より多くの CPU ノードを利用するよりも高い性能を達成する．

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 [8]．COMA は約 1PFLOPS の理論ピーク演算性能を持ち，2015 年 9 月現在，日本最大の Xeon Phi クラスタである．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．各 Xeon Phi の理論ピーク演算性能は 1TFLOPS で，ノードあたり 2.5TFLOPS の理論ピーク演算性能となる．COMA の諸元について**表 2** に示し，ノード構成について**図 4** 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 [9], [10]．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．

各実行方法の，MPI プロセスおよび OpenMP スレッドの割り当てについて**表 3** にまとめる．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 Hyper Threading が無効になっているため，各プロセスの OpenMP スレッド数は 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．COMA に接続されている Xeon Phi 7110P は 61 個の物理コアを持ち，最大で 244 スレッド並列が可能である．しかしながら，そのうち 1 コアは OS のプロセス制御に用いられているため，本研究では 1 コアを除外し 60 コアでの並列化を考える．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 実行および Native 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．

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，ハミルトニアン計算で必要となっているステンシル計算を含む全ての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

```
integer,intent(in)      :: NL,IDX(4,NL-1)
integer,intent(in)      :: IDY(4,NL-1),IDZ(4,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-dimension
  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-dimension
  v(2)=Cy(1)*(E(IDY(1,i))+E(IDY(-1,i))) &
  &    +Cy(2)*(E(IDY(2,i))+E(IDY(-2,i))) &
  &    +Cy(3)*(E(IDY(3,i))+E(IDY(-3,i))) &
  &    +Cy(4)*(E(IDY(4,i))+E(IDY(-4,i)))

  w(2)=Dy(1)*(E(IDY(1,i))-E(IDY(-1,i))) &
  &    +Dy(2)*(E(IDY(2,i))-E(IDY(-2,i))) &
  &    +Dy(3)*(E(IDY(3,i))-E(IDY(-3,i))) &
  &    +Dy(4)*(E(IDY(4,i))-E(IDY(-4,i)))

  ! z-dimension
  v(3)=Cz(1)*(E(IDZ(1,i))+E(IDZ(-1,i))) &
  &    +Cz(2)*(E(IDZ(2,i))+E(IDZ(-2,i))) &
  &    +Cz(3)*(E(IDZ(3,i))+E(IDZ(-3,i))) &
  &    +Cz(4)*(E(IDZ(4,i))+E(IDZ(-4,i)))

  w(3)=Dz(1)*(E(IDZ(1,i))-E(IDZ(-1,i))) &
  &    +Dz(2)*(E(IDZ(2,i))-E(IDZ(-2,i))) &
  &    +Dz(3)*(E(IDZ(3,i))-E(IDZ(-3,i))) &
  &    +Dz(4)*(E(IDZ(4,i))-E(IDZ(-4,i)))

  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 5 ステンシル計算のオリジナル実装.

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 5** 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，コード最適化のため 0 始まりの 0-origin に変換している．図 5 に示したステンシル計算では，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，2 種類の 25 点ステンシル計算が行われている．また，周期境界領域を考慮したインデックス計算を省略するため，IDX，IDY，IDZ という間接参照配列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．v，w に格納された値は総和され，評価点の更新が行われる．必要な演算を洗い出すと，倍精度複素数の加減乗算および総和，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．

4.1 Xeon Phi での複素数演算について

はじめに，Xeon Phi での複素数演算について述べる．Intel SSE および AVX では，複素数演算を高速に行うために SSE3 から下記の命令が追加されている [11].

MOVSLDUP, MOVSHDUP, MOVDDUP
: MOVSLDUP は単精度複素数ベクトルのうち実部データを虚部にコピーする．MOVSHDUP は逆に単精度複素数ベクトルのうち虚部データを実部へコピーする．MOVDDUP は倍精度複素数ベクトル用で，MOVSLDUP および MOVSHDUP の両方の機能が利用できる．

ADDSUBPS, ADDSUBPD
: 複素数ベクトルで下位ビット (実部相当) では減算，上位ビット (虚部相当) では加算を行う．この命令で複素数乗算を容易に実装することができる．

これらの命令セットは AVX で 256bit 命令が用意され，倍精度複素数については 2 個の要素を同時演算できるようになっている．しかしながら，Xeon Phi に実装されている 512bit 命令である Intel IMCI (Initial Many Core Instructions) ではこれらの命令セットは実装されていない [12]. そのため，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するようなアセンブリが作成され，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．これらの問題から，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．

本研究では，これらの複素数向け命令が存在しない状態で組み込み関数をハンドコーディングし高い性能を得るのは非常に困難であると考え，組み込み関数を用いない最適化を考える．ただし，Xeon Phi ではベクトル化が非常に重要であるため，Intel コンパイラによる自動ベクトル化を最大限活用する必要がある．

4.2 最適化オプションの検証

オリジナル実装に対し，まずコードを修正しない最適化として最適化オプションの効果を検証した．Intel コンパイラには多くの最適化オプションが用意されているが，中には Xeon Phi 専用のオプションがある [13]. 今回はステ

ンシル計算で効果が期待されるオプションを取り上げ，それらの効果について検証する．ただし，現在検証中のためXeon Phi で重要となるソフトウェアプリフェッチについては取り上げない．

`opt-assume-safe-padding`
: 変数と動的確保されたメモリがパディングされていると仮定した最適化をコンパイラに行わせる．動的確保されたオブジェクトには，メモリパディングが行われ，より効率的なベクトル化が行われる．本研究では，アプリケーション全体に対しこのオプションを常に指定する．

`opt-ra-region-strategy=name`
: レジスタの割り当てアルゴリズムの単位を指定する．`name` には `routine`，`block`，`trace`，`loop` が指定できる．1 個のループで各次元の計算が行われているため，このオプションでレジスタの割り当て効率が改善し性能向上が期待できる．

`opt-streaming-stores always`
: キャッシュを介さず計算結果を直接メモリへ書き込むストリーミングストア命令を発行させる．ストリーミングストア命令によってキャッシュ汚染を回避できる．図 5 のコードでは出力先配列 `F` には一度しか書き込まないため効果が期待できる [14]．

`opt-streaming-cache-evict=n`
: ストリーミングロード・ストア命令で，キャッシュからデータが溢れた際のキャッシュ退避命令の挙動を示す．`n` に 0 を指定するとキャッシュ退避命令を発行せず，1 を指定すると L1 キャッシュへの退避命令が，2 を指定すると L2 キャッシュへの退避命令がコンパイラによって発行される．`F` の各要素には一度しか書き込まないため，キャッシュへの退避命令を抑制することで性能改善につながる可能性がある．

`opt-gather-scatter-unroll=n`
: Intel コンパイラにおいて，Gather/Scatter 命令は独立なループを用いて発行されるが，この時に `n` に Gather/Scatter 命令の同時発行数を指定できる．ステンシル計算では各次元の計算でのベクトル長が 4 となり，512bit SIMD 1 回で計算できる．ただし 1 回の計算には `E` から 8 個の要素を取得する必要があるため Gather 命令が 2 回必要となる．そのため `n` には 2 の倍数を指定した．

4.3 コンパイラによるベクトル化コストの低減

次に，コンパイラによるベクトル化を考慮しコードの修正を行った．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するため，記述によってはベクトル化が複雑化し期待した性能を得られていない可能性がある．そのため，コンパイラによるベクトル化で最適となるようにコード修正を行った．図 5 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．しかしながら，メモリアクセスから考えると間接参照配列は各次元で別々にあり，1 回の計算においてメモリを読む回数が多くなってしまう．また `v`，`w` は最後に総和されるため，各次元で個別に計算する必要はない．そこで，ベクトル化を考慮し最小のコード修正を行った．

図 5 では `v`，`w` に対する 2 種類の 25 点ステンシル計算が行われているが，`v` の計算を取り上げ**図 6** にベクトル化のイメージについて示す．図 6-(a) は元々の計算の流れだが，各次元で別々に和を計算したあとに総和を計算しており余分な計算が行われている．そのため図 6-(b) では `IDX`，`IDY`，`IDZ` を 1 つの配列にまとめすべての次元をまとめて計算し，総和を計算するステップが減らした．また，長さ 4 の 3 回のベクトル演算が，長さ 12 の 1 回のベクトル演算になっている．3 回の SIMD 演算が必要となり両者に違いはないが，連続な演算となるためベクトル化が容易になると考えられる．

しかしながら，上記のコード修正でも不十分と考えられる．1 点目に，近傍点のアクセスには `IDX`，`IDY`，`IDZ` の間接参照配列を用いている．このため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ないため，ベクトルレジスタへのロードを効率的に行えない可能性がある．2 点目に，近傍点のアクセスのためにこれらの配列へアクセスする必要がありメモリアクセスが増加してしまう．そこで，`NL` のループを `NLx`，`NLy`，`NLz` の 3 重ループに変換して間接参照配列を使用しないようにコードを修正し，ステンシル計算時に都度インデックスを計算することでコンパイラがメモリアクセスパターンを把握できるようにした．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．このループ変換で，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，Y, X 方向の 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．周期境界上の計算を抜き出し，周期境界条件を考慮せずとも計算できる領域を先に計算するループ分割も考えられるが，コンパイラによる自動ベクトル化が複雑となりかえって性能が低下することがわかったため，ここでは適用していない．

4.4 周期境界条件を考慮した最適化

本研究で取り扱う 1 個のステンシル計算の計算サイズは 16^3 と非常に小さく，データサイズにすると 64KB となる．Xeon Phi の L2 キャッシュは物理コアに対し 512KB なので，1 個の物理コア上で 4 つのスレッドが同時実行された場

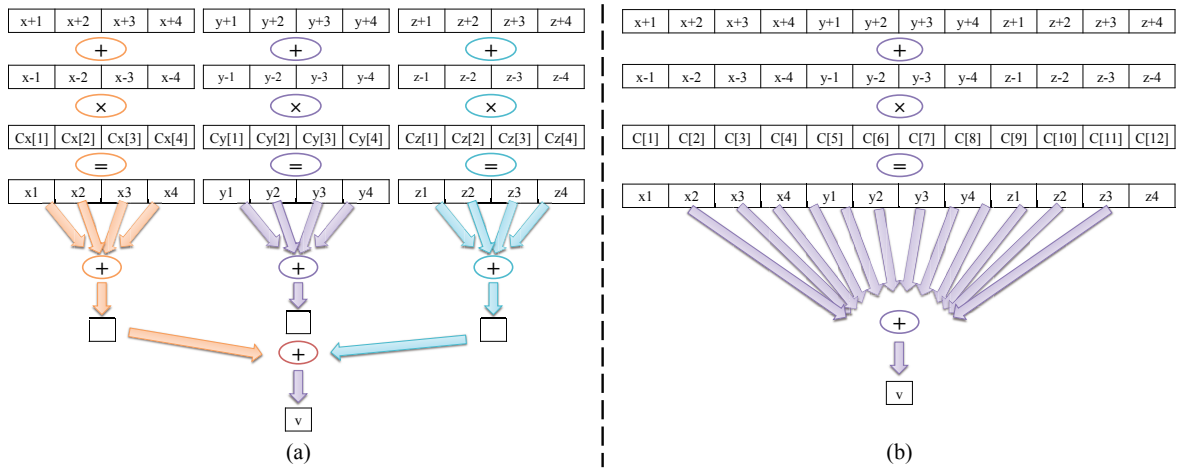

図 6 (a) オリジナル実装でのベクトル計算イメージ，(b) コンパイラによる最適化を意識したベクトル計算イメージ．

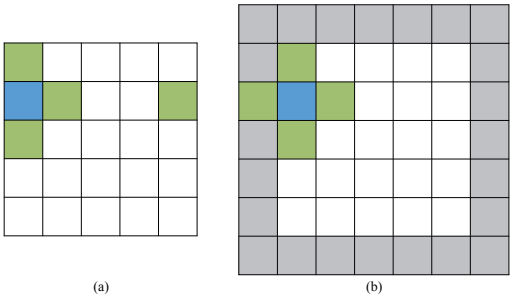

図 7 周期境界条件による 2 次元 5 点ステンシル計算でのメモリアクセス，(a) 元々のメモリアクセス，(b) 周期境界領域をコピーした場合，白色が計算領域で灰色が周期境界領域のコピーである．

合でも 1 個の計算領域は L2 キャッシュに収まる．しかしながら周期境界条件を考えると，最大で $16 \times 16 \times 15$ 離れた要素にアクセスする必要があり，メモリアクセスのレイテンシは無視できないレベルと考えられる．ここで，周期境界領域にあたる領域をコピーすれば，小領域上のすべての点においてメモリアクセスコストを同等にできる．**図 7** に，2 次元 5 点ステンシル計算でのメモリアクセスについて示す．元々の実装だと，Y 方向へのアクセスの場合 16×15 離れたメモリへのアクセスが必要となるが，図 7-(b) のように周期境界をコピーすることで，計算領域のどの点を更新する場合でも最大 16×4 離れたメモリへのアクセスとなりアクセスコストが均一となる．しかしながら，各方向に 4 点ずつアクセスするため，周期境界領域をコピーすると 16^3 だった小領域が 24^3 に拡張される．拡張した小領域のサイズは 216KB になるため，L2 キャッシュからも計算領域が溢れ期待した性能向上が得られない可能性がある．

5. 性能評価

5.1 ハミルトニアン計算性能

ハミルトニアン計算の性能評価では，16 ノードで実行した場合の性能を記載する．この際，32 台の Xeon Phi で実行するため MPI プロセス (Xeon Phi) あたりに 6912 個の小領域を計算する．また，OpenMP のスレッド数はすべてのケースにおいて 240 スレッドが最適化だったため，すべて 240 スレッドでの実行結果を示している．

表 5 に最適化オプションのステンシル計算に対する効果を示す．Xeon Phi では，全体の最適化オプションに`-mmic -O3 -opt-assume-safe-padding` を付与している．顕著な結果として，レジスタ割り当てアルゴリズムの単位を `block` とすることで性能が 5.47%向上した．また，ストリーミングストア命令で `F` への書き込みを行わせることで 3.32%性能が向上し，期待通りとなった．キャッシュ退避命令の抑制は，単独で指定すると性能低下につながっているが，`opt-streaming-stores` と組み合わせることで多少の効果が得られている．

`opt-gather-scatter-unroll` では，6 が最適となった．`v`，`w` の計算で使っている `E` の値は同じで，各点の更新で 24 個の近傍点が必要となる．ベクトル長は 4 なので，すべての近傍点を集めるには 6 回の Gather 命令が必要となる．つまり `n=6` とした場合は Gather 命令をすべて同時に発行するため，最適な性能が得られたと考えられる．`n=8` とすると余分な Gather 命令が 2 回発行されてしまうため，7%近くの性能低下につながっている．

最後に，特に性能の良かった最適化オプションである “RA Block”，“Store Evict 2”，“Gather 6” を組み合わせることで約 8%の性能向上となった．単純な加算であれば 12%程度の性能向上が期待できるが，それぞれが影響し約 8%の性能向上にとどまったと考えられる．

図 8 に，各最適化を行った Xeon Phi でのハミルトニアンの計算時間を示す．“Original” がコードの修正または最適化オプションの付与などを行わず，オリジナル実装のまま実行した場合の計算時間である．“Optimize Options” が最適化オプションで表 5 の一番下に示した組み合わせでの

表 4 “Tuned” の性能比較．() 内は Gather を使用する演算数．

	Load	index	Rel. Perf.	Gather
(1)	aligned	logical	100.00 %	Z (8)
(2)	aligned	mod	48.64 %	Z, Y, X (24)
(3)	unaligned	logical	99.64 %	Z (8)
(4)	unaligned	mod	57.49 %	Z (8)

実行時間，“Vectorize” がコンパイラのベクトル化を意識した最小のコード修正を行った場合の実行時間，“Tuned” は “Vectorize” に対して間接参照配列を使用せずに都度計算し，ロードするメモリの位置をコンパイラが把握できるようにしたものである．最後に “Shadow Region” は周期境界領域をコピーしメモリアクセスコストを均一にしたものである．比較用に，“Original”, “Vectorize”, “Tuned” の実装での Ivy-Bridge の実行時間を破線で示している．“Tuned” が最も高速で，“Original” に対して約 2.96 倍，“Ivy (Original)” に対して約 1.77 倍，“Ivy (Vectorize)” に対して約 1.36 倍の性能となっている．“Optimize Options” が示す通り，最適化オプションの適用はアプリケーションの Xeon Phi への移植時に一考の余地があると考えられるが，それでも従来の CPU での性能に比べると非常に低い性能となっている．最小のコード修正を行った “Vectorize” でも “Original” に対し約 1.52 倍の性能となっており，コードの修正が必要不可欠であると言える．また，“Ivy (Tuned)” は性能が低下してしまっており，Xeon Phi と CPU で最適なコードが異なり書き分けが必要となっている．

“Vectorize” と “Tuned” の結果が示すように，やはりコンパイラのベクトル化を意識してコードを修正する必要があるが，“Vectorize” の場合，連続でベクトル演算ができるようにはなったが，メモリアクセスをコンパイラが把握できず性能が “Original” に対し約 1.52 倍にとどまった．“Tuned” では，メモリアクセスを下記のコード例のように `assume` ディレクティブを用いてコンパイラにメモリアクセスがアラインされていることを教えたため最適なロード命令が用いられたと考えられる．**表 4** の (1) と (3) に，アラインされていないロード命令を利用した場合にどの程度の性能低下が起きるかを示す．

```
#define IDX(n) \
  ((ix-(iand(ix+(n)+NLx,NLx-1))*NLy*NLz)
sx1 = IDX( 1)
sx2 = IDX( 2)
...
sx5 = IDX(-1)
...
!dir$ assume (mod(sx1,4) == 0)
!dir$ assume (mod(sx2,4) == 0)
...

idx=ix*NLy*NLz + iy*NLz + iz
v=v&
&+(Cx(1)*(E(idx-sx1)+E(idx-sx5))+ ... )*-0.5d0
&+(Dx(1)*(E(idx-sx1)-E(idx-sx5))+ ... )*-zI
```

`assume` は，コンパイラに対してキーワード内の評価式が常に真であることを仮定できることを伝える．今回の場合，各次元のサイズを示す `NLx`，`NLy`，`NLz` はすべて 16 で，512bit SIMD での倍精度複素数ベクトル長である 4 の倍数となる．そのため，X と Y 次元のストライドメモリアクセスは常にキャッシュラインをまたがないアラインされたアクセスであり，コンパイラは効率の良いメモリアクセスを利用できると考えられるが，非アラインなロード命令を用いたことによる性能への影響はわずかで，インデックスの計算の方に問題があることがわかった．上記のコード中で，`NLx`，`NLy`，`NLz` は 2 のべき乗サイズであるためインデックス計算に論理積演算を用いているが，これを下記のように `mod` 演算にすると，コンパイラは異なるアセンブリを生成する．

```
#define IDX(n) \
  ((ix-(mod(ix+(n)+NLx,NLx)))*NLy*NLz)
```

この場合，コンパイラは Gather 命令を使って計算に必要なデータを収集するが，前述の論理積演算の場合は，一般的なロード命令を用いている．ただし連続な Z 次元では，キャッシュラインをまたぐか，周期境界条件によって非連続なアクセスが発生するため，どちらでも Gather 命令が利用されている．表 4 を見ると，`assume` の有無による性能差は限定的で，`mod` 演算の利用で 2 倍近い性能差が生じている．除数が 2 のべき乗の場合，剰余は論理積で代替可能だが，(2) および (4) でコンパイラは `mod` 演算を Xeon Phi の 512bit SIMD 命令にある SVML (Short Vector Math Library) を用いて計算している．表中の (3) と (4) の大きな違いはその点のみであったため，`mod` 演算が用いられたことによって約 57%まで性能が低下し，さらに (2) ではすべての近傍点の読み込みに Gather 命令が用いられたため約 48%まで性能が低下したと考えられる．これらの結果から，Gather 命令よりもロード命令を使用した方が高速なストライドアクセスが可能と考えられ，またインデックス計算では，`mod` 演算の使用を抑制したほうが良いのではないかと考えられる．

また，“Shadow Region” はメモリアクセスコストが均一になったため性能が向上すると考えられたが，“Tuned” に対して約 95%程度の性能となっている．ハミルトニアン計算での実行時間内訳を**図 9** に示す．“Stencil” はステンシル計算のみの時間，“Init” はステンシル計算のために実空間である小領域を一時領域へコピーする必要があるためそのコピーに要する時間，“Update” は 1 回のハミルトニアン計算の結果を用いた実空間の更新，“Other” はハミルトニアン計算中のステンシル以外の計算時間である．ステンシル計算だけで見れば，“Ivy (Vectorize)” に対し “Tuned” は約 1.57 倍の性能だが，“Shadow Region” は約 2.42 倍の性能向上が得られている．しかしながら，ステンシル以外の

計算時間が大幅に増加してしまっている．特に “Update” の計算時間が 5 倍近く増加している．“Update” では，実空間の更新の他にハミルトニアン計算によって得たデータを別の領域にコピーする必要がある．同計算が 4 回のテイラー展開された式中で行われているため，n 回目の計算を行う場合には，$n-1$ 回目に計算したデータが必要となるからである．16^3 の領域を，24^3 の領域に拡張したため，メモリコピーの回数は領域のサイズと同じく約 3 倍に増加する．全体性能の評価では，CPU では “Vectorize” を，Xeon Phi では “Tuned” のコードを用いて性能評価を行う．

5.2 時間発展計算性能

次に，時間発展計算全体での性能評価を行う．時間発展計算では，CPU 実行と Native 実行に加え，Symmetric 実行の評価も行う．また実行性能については，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．調整後の性能では，CPU か Xeon Phi どちらか性能が低い方の計算量を少なく割り当てるため，当然性能が高い方は計算量が増えその分計算時間は増加する．そのため，CPU の実行時間を T_{CPU}，Xeon Phi の実行時間を T_{Native} とすると，計算量の割り当てを最適値に変更した Symmetric 実行の実行時間 T_{Sym} は概ね $(\mathrm{T}_{CPU}+\mathrm{T}_{Native})\div 2$ となる．ただし，実際には CPU の方が遅い計算，Xeon Phi の方が遅い計算などが存在するため，それをオーバーヘッド $\mathrm{T}_{overhead}$ として $\mathrm{T}_{Sym}\approx(\mathrm{T}_{CPU}+\mathrm{T}_{Native})\div 2\pm\mathrm{T}_{overhead}$ と考えられる．

本研究では，波動関数のパラメータである Bloch wave number k (NK) を MPI で分散する．そのため，NK の分散方法を変更することで，Symmetric 実行時に CPU と Xeon Phi での大まかな計算量が調整可能である．均等割当を行っている元々のコードが約 10 行程度であるのに対し，20 から 30 行程度で計算量を調整可能なコードに変更可能なため，負荷分散を極めて容易に実現できる．しかしながら，CPU と Xeon Phi での計算量が変化するため加算順序等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，計算量の調整による計算結果への影響は確認されていない．

MPI プロセス単位で比較した場合の CPU 実行に対する相対性能を**図 10** に示す．ここで，“Symmetric (Tuned)” は計算量の割り当てを CPU と Xeon Phi で変更し，最適値にした場合の性能である．“Symmetric” と同一性能となっている場合，割り当てを変更しても性能が変わらなかったことを示す．“Symmetric” の場合，計算量は均等割当となっているため性能が低い CPU 実行に性能が律速されている．256MPI プロセスとなると，強スケーリングの結果並列性が低下し，Native 実行の性能が CPU 実行の性能よりも低くなったため，Symmetric 実行の性能が CPU 実行よりも低くなってしまっている．“Symmetric (Tuned)” は概ね予測に近い性能となっているが，CPU と Xeon Phi の中間よりも低い性能となっている．

表 3 で示したとおり，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 実行や Native 実行と同等の計算リソースで実行する場合，必要な計算ノード数は半分となる．今回の結果では，少なくとも 128MPI プロセスまでは，N 台の計算ノードを確保し CPU 実行で計算するよりも，$N\div 2$ の計算ノードを確保して Symmetric 実行で計算する方が高速となっている．

次に計算ノード単位で比較した場合の CPU 実行に対する相対性能を**図 11** に示す．ここで，Native 実行は MPI プロセス数が CPU 実行と同一で，相対性能は図 10 と同様となるため省略した．また，“Symmetric (Tuned)” や “Symmetric” と同一性能となっている箇所は図 10 と同様である．計算ノード単位で比較した場合，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，予測と当てはまったのは 16 ノードの “Symmetric (Tuned)” のみとなった．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 180 スレッドが最適であった．原因は不明だが，Symmetric 実行で Xeon Phi の OpenMP スレッド数を 240 で実行した場合に通信時間が増加してしまうことがわかっている．そのため，図 10 でも同様だが，Symmetric 実行の性能が予測よりも低くなってしまっていると考えられる．この問題については，引き続き調査を行っている．

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である ARTED を Xeon Phi 向けに最適化し，Xeon Phi を 1 台の計算ノードと捉え CPU との協調計算を行う Symmetric 実行の有効性を検証した．

まず，コードを Xeon Phi に対し最適化を行い，時間発展計算中で支配的な計算となっているステンシル計算を中心に最適化を行った．ステンシル計算は，時間依存

表 5　ステンシル計算に対する最適化オプションの効果.

	Applying Options	Performance Gain [%]
RA Routine	-opt-ra-region-strategy=routine	+2.67
RA Block	-opt-ra-region-strategy=block	+5.47
RA Trace	-opt-ra-region-strategy=trace	+3.39
RA Loop	-opt-ra-region-strategy=loop	-0.02
Cache Evict 0	-opt-streaming-cache-evict=0	-0.50
Cache Evict 1	-opt-streaming-cache-evict=1	-0.29
Cache Evict 2	-opt-streaming-cache-evict=2	+1.24
Stream Store	-opt-streaming-stores always	+3.32
Store Evict 0	-opt-streaming-stores always -opt-streaming-cache-evict=0	+3.32
Store Evict 1	-opt-streaming-stores always -opt-streaming-cache-evict=1	+3.71
Store Evict 2	-opt-streaming-stores always -opt-streaming-cache-evict=2	+4.18
Gather 2	-opt-gather-scatter-unroll=2	-0.32
Gather 4	-opt-gather-scatter-unroll=4	+2.74
Gather 6	-opt-gather-scatter-unroll=6	+2.95
Gather 8	-opt-gather-scatter-unroll=8	-6.80
Combination	-opt-gather-scatter-unroll=6 -opt-ra-region-strategy=block -opt-streaming-stores always -opt-streaming-cache-evict=2	+8.04

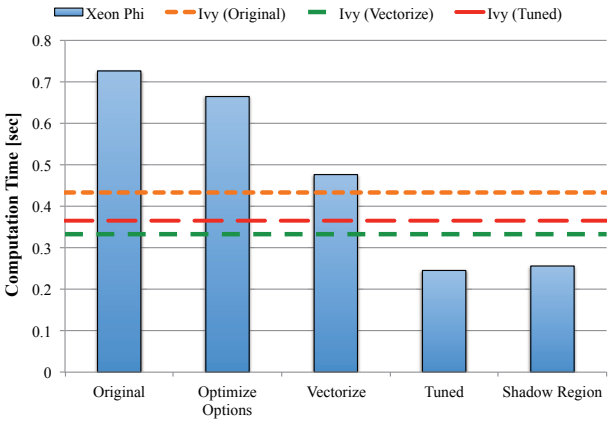

図 8　時間発展 1 回あたりのハミルトニアンの計算時間.

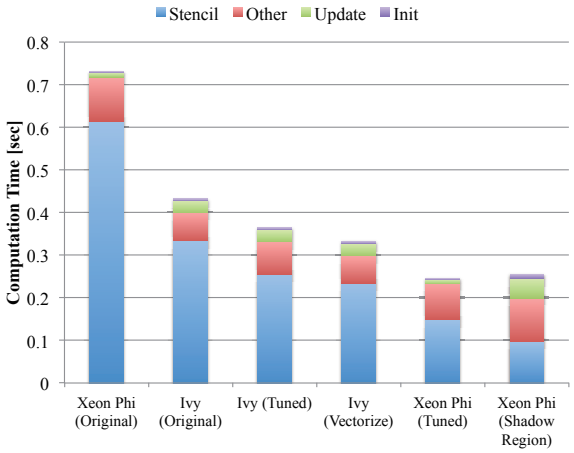

図 9　ハミルトニアン計算の実行時間内訳.

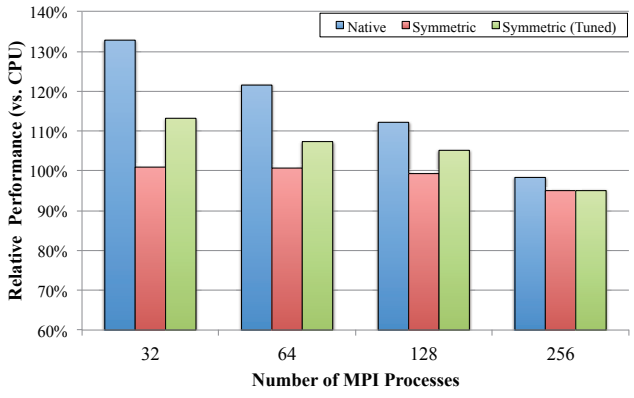

図 10　時間発展計算の相対性能 (MPI プロセス).

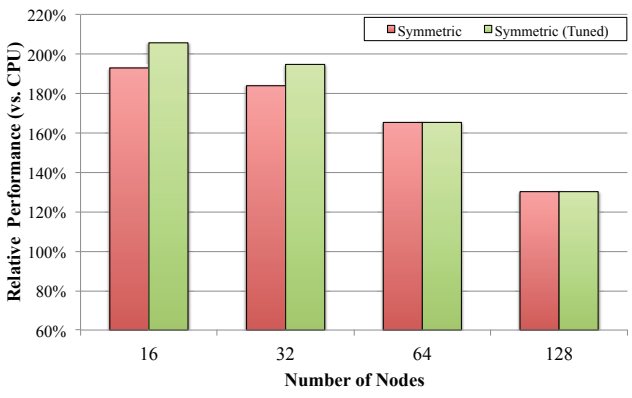

図 11　時間発展計算の相対性能 (計算ノード).

Kohn-Sham 方程式から導出される電子の波動関数でハミルトニアンを計算する際に用いられている．ARTED では一般的なステンシル計算と異なり，メモリ上独立で存在する多数の小領域を OpenMP で並列に計算する必要があるため，シングルスレッドレベルでの最適化が必要となっていた．ステンシル計算コードを抜き出し，コンパイラの最適化オプションを適用することで約 8%性能が向上し，最適化オプションの適用も一考する必要があると考えられる．しかしながら，コードをコンパイラがベクトル化しやすいよう修正を行うことで，約 52%性能が向上したため，コードの修正は必要不可欠であるといえる．大幅にコードを修正することで，組み込み関数を用いずに Ivy-Bridge Xeon CPU に対しステンシル計算では約 2.42 倍の性能向上が得られたが，ハミルトニアン計算全体では計算時間が増加し性能が低下してしまった．ハミルトニアン計算全体では，最適化後の CPU に対し約 1.35 倍の性能向上となっている．また，ハミルトニアン計算全体でステンシル計算に費やされる時間はおよそ 50 から 60%程度となってきたため，今後はハミルトニアン計算全体の最適化が必要である．

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 1.32 倍の性能を達成し，CPU 実行よりも高い性能となりつつある．これらの実装を用いて，Symmetric 実行の性能評価を行った．Symmetric 実行は，デフォルトでは問題を均等分散するため，全体通信によって性能が低い CPU 側に律速された．しかしながら，計算量を調整することにより N 台の計算ノードを確保して CPU のみで計算する場合に対し，CPU と Xeon Phi の両方を利用し $N \div 2$ 台の計算ノードで Symmetric 実行を行うことで同等以上の性能が得られるようになった．同じ N 台のノードで Symmetric 実行を行った場合には，計算リソースが 2 倍以上になるため性能も 2 倍以上得られるのが期待されるが，強スケーリング性が増加し期待した性能よりも低いものとなっている．Symmetric 実行の性能は，CPU 実行と Native 実行性能の中間付近となることが予測されるため，引き続き Xeon Phi での最適化が必要である．

本研究の今後の課題を以下にまとめる．

- Xeon Phi のハミルトニアン計算のさらなる最適化，今後は組み込み関数を用いた最適化と，ステンシル計算だけでなくハミルトニアン計算全体の最適化を考える必要がある．
- 時間発展計算全体の性能向上，現在でもステンシル計算が支配的となっているため，ステンシル計算の最適化により全体性能の向上が期待できる．

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」による．本研究の性能チューニング，評価にあたり東京大学情報基盤センターの塙敏博特任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] Intel: CilkPlus, `https://www.cilkplus.org/`.
[2] Shunsuke A. Sato and Kazuhiro Yabana: Maxwell + TDDFT multi-scale simulation for laser-matter interactions, *J. Adv. Simulat. Sci. Eng.*, Vol. 1, No. 1, pp. 98–110 (2014).
[3] Hasegawa, Y., Iwata, J.-I., Tsuji, M., Takahashi, D., Oshiyama, A., Minami, K., Boku, T., Shoji, F., Uno, A., Kurokawa, M., Inoue, H., Miyoshi, I. and Yokokawa, M.: 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] Schultze, M., Ramasesha, K., Pemmaraju, C., Sato, S., Whitmore, D., Gandman, A., Prell, J. S., Borja, L. J., Prendergast, D., Yabana, K., Neumark, D. M. and Leone, S. R.: Attosecond band-gap dynamics in silicon, *Science 12 December 2014*, Vol. 346, No. 6215, pp. 1348–1352 (online), DOI: 10.1126/science.1260311.
[5] RIKEN Advanced Institute for Computational Science: K Computer, `http://www.aics.riken.jp/en/k-computer/`.
[6] 廣川 祐太，朴 泰祐，佐藤 駿丞，矢花一浩：実時間実空間密度汎関数理論による電子動力学シミュレーションの Xeon Phi クラスタ向け最適化，情報処理学会研究報告，Vol. 2015-HPC-148, No. 19 (2015).
[7] 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).
[8] 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`.
[9] S. Potluri, D. Bureddy, K. Hamidouche, A. Venkatesh, K. Kandalla, H. Subramoni, D. Panda: MVAPICH-PRISM: A Proxy-based Communication Framework using InfiniBand and SCIF for Intel MIC Clusters, *SC '13 Proceedings* (2013).
[10] 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).
[11] Takahashi, D.: 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).
[12] Intel: Intrinsics Guide, `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.
[13] Intel: User and Reference Guide for the Intel Fortran Compiler 15.0, `https://software.intel.com/en-us/compiler_15.0_ug_f`.
[14] Krishnaiyer, R., Kultursay, E., Chawla, P., Preis, S., Zvezdin, A. and Saito, H.: 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).