Vol.2016-HPC-155 No.39
2016/8/10

階層型行列ベクトル積のメニーコア向け最適化

大島 聡史[1,a)]　伊田 明弘[1]　河合 直聡[1]　塙 敏博[1]

概要：階層型行列とは密行列を多数の小密行列と低ランク行列を用いて近似した行列であり, これを用いることで密行列をそのまま用いるよりも大規模な計算を行うことが可能となる. 我々はこれまで階層型行列を用いた境界要素法による静電場解析問題の実装と評価をマルチコア CPU 環境において実施してきた. 現在はこの計算をメニーコアプロセッサや GPU などでも活用することを目指している. 本稿では特に境界要素法に含まれる計算の中でも実行時間の長い階層型行列ベクトル積について, メニーコアプロセッサ Intel Xeon Phi 5110P 上で実行し性能を評価した. さらに既存の様々な最適化手法についての検討と実装を行った. 実験の結果, 幾つかのケースにて性能向上を確認した. 行列 human_1x1 行列に対する行列ベクトル積においては,180 スレッド同士の比較でオリジナルの実装に対して動的なスケジューリングの利用により最大 41%の実行時間が削減された. また小密行列ベクトル積に対してブロック化を行うことで, 行列 108kp22 行列において最大 6.1%の実行時間が削減された.

1. はじめに

高速な計算や大規模な計算への要求にともない, 様々な計算機環境が活用されている. 特に, 従来は半導体プロセスの微細化に基づき動作周波数を向上させることでプロセッサの性能を上げることができたのに対して, 今日では物理的な大きさの限界に起因する微細化の難しさやリーク電流の増大が問題となり計算コア単体の性能向上が難しくなってきていることから, 多数の計算コアを搭載しプロセッサ全体で高性能を達成するプロセッサの普及が進んでいる. 我々の所属する東京大学情報基盤センターにおいても, 2016 年から 2017 年にかけて, Intel 社製の最新メニーコアプロセッサ（Intel Xeon Phi, 開発コードネーム Knights Landing）を搭載した Oakforest-PACS[1] や, NVIDIA 社製の最新 GPU（Pascal アーキテクチャ）を搭載した Reedbush[2] の稼働が予定されている. これらのハードウェアにおいては従来の並列計算機環境と同じ MPI や OpenMP, またはそれに近い感覚で利用可能な OpenACC などが利用可能であるものの, 十分な性能を引き出すためにはハードウェアの特徴に合わせた最適化が必要である. そのため, 従来のマルチコア CPU 向けに開発されたアルゴリズムやアプリケーションに対して最新ハードウェア向けに性能評価や最適化を行うことは急務である.

一方, 大規模な密行列を用いて計算を行いたいという需要は大きい. 密行列を用いた計算についてはキャッシュの活用などの手法を用いた高速計算に関する研究が多く行われている. しかし大規模な密行列を扱うには多くのメモリ容量が必要であるという根本的な問題がある. 今日の計算機環境においては, 演算性能 (FLOPS) の向上に対してメモリ転送性能 (Byte/sec) やメモリ容量 (Byte) の向上の速度が遅い. そのため大規模な密行列を高速に計算するための演算性能が確保できても, それに十分なデータを提供できるだけのメモリ転送速度や, 行列を保持するためのメモリが足りないという問題が発生する. そこで我々は, 密行列を多数の小密行列と低ランク行列を用いて近似した行列 (Hierarchical matrix ($\mathcal{H}-matrix$), 以降では**階層型行列**と呼ぶ)[3] を用いた計算に着目している. この方法を用いればより大きな規模の密行列を扱うことが可能となるため, 将来の大規模計算機環境でも様々な計算問題にて有効に活用されることが期待される.

本稿では特に階層型行列を用いた行列ベクトル積を扱う. 今日の数値シミュレーションにおいては CG(Conjugate Gradient) 法に基づく計算が広く利用されており, 行列ベクトル積はこれらの計算を律速する計算であるため高速化が重要である. 階層型行列を用いて CG 法に基づく計算を行う研究も既に実施されており, 我々も ppOpen-HPC プロジェクト [4] において対応するライブラリを公開している. しかし従来はマルチコア CPU が主な対象計算機となっており, メニーコアプロセッサや GPU での性能評価や最適化は十分に行われていなかった. そこで本稿ではメニーコアプロセッサを対象として性能評価と最適化を実施する.

本稿の構成は以下の通りである. 2 章では対象とする計

[1] 東京大学 情報基盤センター
[a)] ohshima@cc.u-tokyo.ac.jp

算の内容について述べる．3 章では性能評価および最適化について述べ, 4 章はまとめの章とする．

2. 階層型行列を用いた計算

2.1 階層型行列

本論文では N 次元実正方行列 $\overline{A} \in \mathbb{R}^{N \times N}$ について考える．本論文で扱う階層型行列（A と表記する）とは，$\overline{A}$ を部分行列に分割した上で，それら部分行列の大半を低ランク行列で近似したものである．ここで，N 次元正方行列の行に関する添え字の集合を $I := \{1, \cdots, N\}$, 列に関する添え字の集合を $J := \{1, \cdots, N\}$ と表す．直積集合 $I \times J$ を重なりなく分割して得られる集合の中で, 各要素 m が I と J の連続した部分集合の直積であるものを M とする．すなわち任意の $m \in M$ は $s_m \subseteq I, t_m \subseteq J$ を用いて $m = s_m \times t_m$ と表される．ある m に対応する $\overline{A}$ の部分行列を

$$A|^m_{s_m \times t_m} \in \mathbb{R}^{\#s_m \times \#t_m} \tag{1}$$

と書く．ここで # は集合の要素数を与える演算子である．階層型行列では, 大半の m について $A|^m_{s_m \times t_m}$ の代わりに以下の低ランク表現 $\tilde{A}|^m$ を用いる．

$$\begin{aligned} &\tilde{A}|^m := V_m \cdot W_m \\ &V_m \in \mathbb{R}^{\#s_m \times r_m} \\ &W_m \in \mathbb{R}^{r_m \times \#t_m} \\ &r_m \leq min(\#s_m, \#t_m) \end{aligned} \tag{2}$$

ここで $r_m \in \mathbb{N}$ は行列 $\tilde{A}|^m$ のランクである．すなわち, 低ランク行列 $\tilde{A}|^m$ とは, 密行列 $A|^m_{s_m \times t_m}$ を V_m と W_m の積により近似した行列である．**図 1** に階層型行列の一例を示す．図 1 の濃く塗りつぶされた部分行列が $A|^m_{s_m \times t_m}$ に, 薄く塗りつぶされた部分行列が $\tilde{A}|^m$ に対応する．

本稿では階層型行列に関する演算，特に行列ベクトル積に関して論じる．ある階層型行列 A に関するデータ量 $N(M)$ は，m に対応する部分行列に関するデータ量 $N(m)$ を用いて以下のように表される．

$$N(M) = \sum_{m \in M} N(m) \tag{3}$$

$$N(m) := \begin{cases} \#s_m \times \#t_m & \text{m が密行列の場合} \\ r_m \times (\#s_m + \#t_m) & \text{m が低ランク行列の場合} \end{cases} \tag{4}$$

r_m が $\#s_m$ や $\#t_m$ と比べて十分に小さい場合，$r_m \times (\#s_m + \#t_m)$ は $\#s_m \times \#t_m$ と比べて小さい値となり，低ランク行列による表現がデータ量の点で有利となる．その結果，密行列を用いる場合と比べ，行列ベクトル積等の行列演算に必要な演算量や行列の保持に必要なメモリ量を低減することができる．

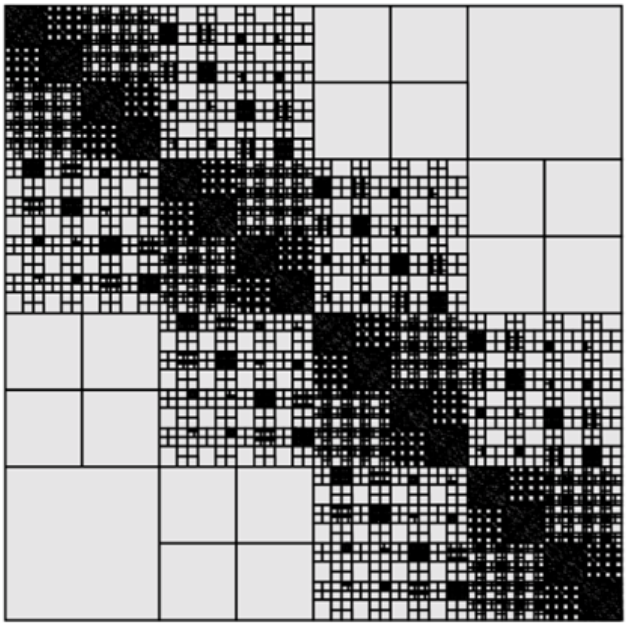

図 1 階層型行列

2.2 階層型行列ベクトル積

本稿では, 以下のような階層型行列ベクトル積を扱う．

$$\begin{aligned} &Ax \to y \\ &x, y \in \mathbb{R}^N \end{aligned} \tag{5}$$

本論文ではこの演算の実施手順として，各部分行列毎に行列積を実行し，その結果を統合することで最終的な結果 y を得るという，最も自然でかつ効率的と考えられる方法を採用する．密行列により表現されている部分行列 $A|^m_{s_m \times t_m}$ については

$$A|^m_{s_m \times t_m} \cdot x|_{t_m} \to \hat{y}|_{s_m} \tag{6}$$

を計算する．ここで,$x|_{t_m}$ は x の各要素のうち tm に対応する要素のみを抜き出して生成した $\#tm$ 個の要素からなるベクトルである．$\hat{y}|_{s_m}$ は $\#s_m$ 個の要素を持つベクトルであり, 各要素は y の s_m に対応する要素の部分積の一つとなる．

次に, 低ランク表現が用いられている行列 $\tilde{A}|^m$ に関しては,$c \in \mathbb{R}^{r_m}$ として, まず

$$Wm \cdot x|_{t_m} \to c|_{r_m} \tag{7}$$

を計算し, さらに

$$Vm \cdot c|_{r_m} \to \hat{y}|_{s_m} \tag{8}$$

を計算することで, $\tilde{A}|^m \cdot x|_{t_m} = Vm \cdot Wm \cdot x|_{t_m} \to \hat{y}|_{s_m}$ を得る．

それぞれの部分行列について $\hat{y}|_{s_m}$ を計算した後,

$$\sum_{m \in M} \hat{y}|_{s_m} \to y \tag{9}$$

のようにこれらを統合し, 最終的な結果とする．

2.3 対象とするプログラムと並列化手法

本稿では階層型行列に関する計算の実装として ppOpen-APPL/BEM ver.0.4.0 に含まれる HACApK 1.0.0 のソースコードを用いる. ppOpen-APPL/BEM は, JST CREST「自動チューニング機構を有するアプリケーション開発・実行環境: ppOpen-HPC」[4] の構成要素の 1 つであり, 境界要素法 (Boundary Element Method, BEM) の実装において HACApK[5] を用いた階層型行列による計算を行っている. この中に階層型行列を用いて BiCGSTAB 法を行うコードが含まれており, 本稿では特にその中における行列ベクトル積計算部分を性能評価と最適化の対象とする. ただし, 低ランク近似アルゴリズムについてはライブラリに含まれている ACA 法を使用せず, 新たに実装した ACA+ 法 [6] を用いている.

上記のプログラムには様々なパラメタが存在し, それらを調整することで生成される階層型行列の形状をある程度変更することができる. 例えば小密行列の大きさをある程度大きくまたは小さくすることや, 低ランク行列作成時の近似精度を変更することもできる. これらの調整は階層型行列に含まれる要素数などに影響するため, 性能評価や最適化に大きな影響を及ぼす可能性がある. 本稿では現状のデフォルト設定にて評価を行い, 行列形状の違いと性能の関係やその最適化について検討や評価は今後の課題とする.

対象プログラムは MPI による並列化と OpenMP による並列化の両方に対応している. 本稿では OpenMP のみで並列化を行い,1 プロセッサで実行する. 並列化手法の詳細は説明は文献 [5] に記載されているが, 特に以下のような特徴がある. スレッド毎の負荷をある程度均一にするためのアルゴリズムが含まれており, これによる割り当ての結果として得られる低ランク行列や小密行列の大きさや開始位置は様々である. また元の密行列の一行に影響を与える低ランク行列や小密行列は複数のスレッドにまたがって配置される可能性が高い. そのためスレッド毎に割り当てられる低ランク行列や小密行列の数は均一ではなく, また全体としては行単位などの明確な単位での分割 (並列化) は行われない. そのため OpenMP 版の実装では atomic 指示文を用いた足しあわせが含まれている. これらの負荷分散アルゴリズムや atomic 指示文が性能に与える影響についても次章にて調査する.

3. 性能評価と最適化

3.1 評価環境

本章では**表 3.1** に示す計算機環境を用いて性能評価を行う. 対象とするメニーコアプロセッサは Intel Xeon Phi 5110P (開発コードネーム Knights Corner, 以下 MIC と呼ぶ) である. ホスト CPU として接続された同世代のマルチコア CPU Intel Xeon E5-2680 v2 と比べると, 計算コアは多いが動作周波数は低い, 総理論演算性能は高いが計算コア単体の性能は低い, メモリ転送性能は高いがメモリ容量は小さい, といった特徴を持つ. コンパイラとしては ifort (IFORT) 16.0.3 20160415 を使用し, 主なコンパイラオプションは-mmic -qopenmp -O3 -align array64byte -align rec32byte を指定した. また, 比較用にホスト CPU にて実行した部分については-qopenmp -O3 -align array32byte -align rec32byte を指定した.

3.2 対象とする行列

本稿では**表 3.2** に示す 5 つの階層型行列を扱う. これらの行列はいずれも境界要素法を用いた静電場解析においてあらわれる行列である. 表中のリーフ数とは低ランク行列による近似行列の組数および小密行列数の合計数である. 階層型行列における近似行列または小密行列はツリーにおける葉 (リーフ) に相当するため, 本稿ではリーフという呼称を用いる. スレッド並列化を行った場合は各スレッドにリーフが割り当てられるが, 2.3 節にて述べたように, その割り当て数量は均等とは限らない.

3.3 性能評価と最適化

MIC を用いた最適化プログラミングについては様々な解説記事・書籍・研究発表などが存在する [7][8]. それらにて明らかにされている主な最適化の指針や注意点としては, Hyper-Threading 機能を用いて 120 スレッド以上の多数のスレッドを使うこと, 多数のコアを活用しきれるだけの高い並列度が対象計算に存在すること, スレッド間の負荷を均等にすること, 512bit 長の SIMD 演算を活用すること, 単純な計算内容にすること, プリフェッチ命令を用いてメモリアクセスを加速すること, などが挙げられる. そこで本節では, これらのいくつかについて評価と検討を行い, 可能であれば性能改善を目指すことにする.

3.3.1 実行時間と割合の確認

BiCGSTAB 法において行列ベクトル積が多くの実行時間を占めることは既に述べたとおりである. 具体的にどの程度の割合を占めるのかを実測にて確認した. **図 2** は BiCGSTAB 法のイタレーションループ部の実コードである. 配列計算に OpenMP を活用するため,workshare 指示文と single 指示文が用いられていることが確認できる. イタレーション内で二度実行されている HACApK_adot_lfmtx_hyp 関数は実際に行列ベクトル積を行う HACApK_adot_body_lfmtx_hyp 関数 (**図 3**) と MPI 通信により構成されている.

以上を踏まえて, まずは BiCGSTAB 法全体の実行時間および BiCGSTAB 法 1 イタレーションにおける HACApK_adot_lfmtx_hyp 関数の実行時間の割合を確認した. **図 4** に BiCGSTAB 法全体の実行時間を示す. このグラフから, 対象とする行列によって実行時間に大きな差があること, MIC の性能は 120 スレッドまたは 180 スレッドでの

表 1　実験環境

	ホスト CPU	メニーコアプロセッサ (MIC)
プロセッサ名称	Xeon E5-2680 v2	Xeon Phi 5110P
動作周波数	2.8 GHz	1.053 GHz
コア数 (有効スレッド数)	10 (20*)	60 (240)
メモリ種別	DDR3	GDDR5
キャッシュ構成	L1 32KB+32KB/core	L1 32KB+32KB/core
	L2 256KB/core	L2 512KB/core
	L3 25MB/socket	L3 N/A
理論演算性能	224 GFLOPS	1010.88 GFLOPS
理論メモリ性能	59.7 GB/s	320 GB/s
TDP	130W	300W

* 本実験環境では Hyper-Threading 機能を無効化しているため 10 まで

表 2　対象とする階層型行列の構成

行列名	100ts	108kp12	108kp22	216h	human_1x1
行数	101250	108000	108000	21600	19664
総リーフ数	222274	243970	243934	50098	46618
近似行列数	89534	86770	86734	17002	16202
小密行列数	132740	157200	157200	33096	20416

```
 do in=1,mstep
   if(zrnorm/bnorm<eps) exit
!$omp workshare
   zp(:nd) =zr(:nd)+beta*(zp(:nd)-zeta*zakp(:nd))
   zkp(:nd)=zp(:nd)
!$omp end workshare
   call HACApK_adot_lfmtx_hyp(zakp,st_leafmtxp,st_ctl,zkp,wws,wwr,isct,irct,nd)
!$omp barrier
!$omp single
   znom=HACApK_dotp_d(nd,zshdw,zr); zden=HACApK_dotp_d(nd,zshdw,zakp);
   alpha=znom/zden; znomold=znom;
!$omp end single
!$omp workshare
   zt(:nd)=zr(:nd)-alpha*zakp(:nd)
   zkt(:nd)=zt(:nd)
!$omp end workshare
   call HACApK_adot_lfmtx_hyp(zakt,st_leafmtxp,st_ctl,zkt,wws,wwr,isct,irct,nd)
!$omp barrier
!$omp single
   znom=HACApK_dotp_d(nd,zakt,zt); zden=HACApK_dotp_d(nd,zakt,zakt);
   zeta=znom/zden;
!$omp end single
!$omp workshare
   u(:nd)=u(:nd)+alpha*zkp(:nd)+zeta*zkt(:nd)
   zr(:nd)=zt(:nd)-zeta*zakt(:nd)
!$omp end workshare
!$omp single
   beta=alpha/zeta*HACApK_dotp_d(nd,zshdw,zr)/znomold;
   zrnorm=HACApK_dotp_d(nd,zr,zr); zrnorm=dsqrt(zrnorm)
   nstp=in
   call MPI_Barrier( icomm, ierr )
   en_measure_time=MPI_Wtime()
   time = en_measure_time - st_measure_time
   if(st_ctl%param(1)>0 .and. mpinr==0) print*,in,time,log10(zrnorm/bnorm)
!$omp end single
 enddo
```

図 2　BiCGSTAB 法のイタレーションループ部の実コード

```
ith = omp_get_thread_num()
ith1 = ith+1
nths=ltmp(ith); nthe=ltmp(ith1)-1
ls=nd; le=1
do ip=nths,nthe
  ndl   =st_leafmtxp%st_lf(ip)%ndl   ; ndt   =st_leafmtxp%st_lf(ip)%ndt   ; ns=ndl*ndt
  nstrtl=st_leafmtxp%st_lf(ip)%nstrtl; nstrtt=st_leafmtxp%st_lf(ip)%nstrtt
  if(nstrtl<ls) ls=nstrtl; if(nstrtl+ndl-1>le) le=nstrtl+ndl-1
  if(st_leafmtxp%st_lf(ip)%ltmtx==1)then
    kt=st_leafmtxp%st_lf(ip)%kt
    zbut(1:kt)=0.0d0
    do il=1,kt
      do it=1,ndt; itt=it+nstrtt-1
        zbut(il)=zbut(il)+st_leafmtxp%st_lf(ip)%a1(it,il)*zu(itt)
      enddo
    enddo
    do il=1,kt
      do it=1,ndl; ill=it+nstrtl-1
        zaut(ill)=zaut(ill)+st_leafmtxp%st_lf(ip)%a2(it,il)*zbut(il)
      enddo
    enddo
  elseif(st_leafmtxp%st_lf(ip)%ltmtx==2)then
    do il=1,ndl; ill=il+nstrtl-1
      do it=1,ndt; itt=it+nstrtt-1
        zaut(ill)=zaut(ill)+st_leafmtxp%st_lf(ip)%a1(it,il)*zu(itt)
      enddo
    enddo
  endif
enddo

do il=ls,le
!$omp atomic
  zau(il)=zau(il)+zaut(il)
enddo
```

A
B
C

図 3　行列ベクトル積部の実コード
(HACApK_adot_body_lfmtx_hyp 関数の抜粋)

実行で最大化されていること, ホスト CPU よりやや低いことが確認できる. また実行時間の割合を確認したところ, どの行列においても HACApK_adot_lfmtx_hyp 関数の実行時間がイタレーション全体の実行時間の 80%以上を占めており, HACApK_adot_lfmtx_hyp 関数の高速化が重要であることが確認できた.

3.3.2　並列度

メニーコアプロセッサを用いた多並列の実行において, 並列化対象となる計算が十分に高い並列度を持つことは非常に重要である. これは, 並列度が不足している場合にはプロセッサ上の全ての計算コアに計算を割り当てることができないためである. 本稿で対象としている行列ベクトル積の並列度は, 階層型行列のリーフ数によって左右される. 既に示したように, 今回対象としている階層型行列のリーフ数は十分に大きいため, 並列度不足により高性能が得られないという問題は生じない.

一方, 並列度が高いのに対して実行時間が長くはないこ

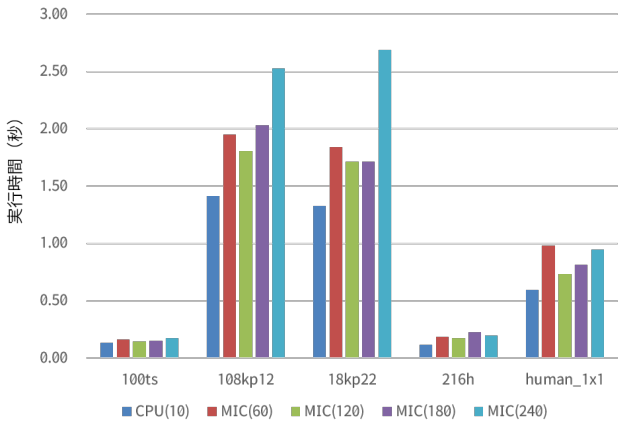

図 4　BiCGSTAB 法全体の実行時間 (凡例の括弧内の数値はスレッド数)

とは, 1 イタレーションあたりの実行時間が短いことを意味する. 極端に 1 イタレーションあたりの実行時間が短いプログラムの最適化は, 動的なスケジューリングや一時配列の初期化などの僅かな処理時間が性能ペナルティとして影響しやすくなるという点で容易ではないと言える.

3.3.3　スレッド間の負荷バランスと atomic 指示文の影響

行列ベクトル積部のソースコードは既に図 3 に示したとおりであるが, このプログラムは大きく 2 つの do ループによって構成されている. 1 つ目は各スレッドが自分の担当するリーフの計算を行う do ループであり, このループはさらに各リーフが低ランク行列の積により近似されている場合の計算部 (図中の A 部) と小密行列により構成されている場合の計算部 (図中の B 部) に別れている. もう 1 つの do ループは計算結果の部分ベクトルを全体ベクトルに足し合わせる処理を行う do ループ (図中の C 部) である. リーフ毎の計算のスレッド並列化においては, 生成自体は本関数の呼び出し側 (BiCGSTAB 法イタレーション部の外側) の omp parallel 指示文で行われており, またループの並列化は omp do 指示文ではなく図 3 の第一行で取得しているスレッド毎の ID を用いて各スレッド自身が計算範囲を取得する方式となっている.

スレッドへの計算範囲の割り当てについては 2 章にて述べた通りであり, スレッド間の負荷バランスを考慮した割り当てが行われている. ただしこの割り当て処理はマルチコア CPU 上で用いられてきた方法であり, MIC においてもスレッド毎の計算時間が概ね均等となるかはわからない. またこの並列化実装においては複数のスレッドがオリジナル行列の同一の行に影響を及ぼすデータを持つ可能性があるため, 単純に結果を足し合わせようとするとスレッド間で計算が衝突する可能性がある. そこで atomic 指示文を用いて計算の衝突を防いでいる. atomic 指示文を用いると複数スレッドによる同一メモリの変更を防ぐことができるが, この処理にはオーバーヘッドがあるため, 多用すると性能が大きく低下する可能性がある.

以上を踏まえて, 各スレッドによる HACApK_adot_body_lfmtx_hyp 関数の実行時間のばらつき (負荷バランス) と, atomic 指示文の影響について確認を行った. atomic 指示文の影響の度合いについては, atomic 演算自体の所要時間を直接測定するのは難しいため, HACApK_adot_body_lfmtx_hyp 関数全体の実行時間および HACApK_adot_body_lfmtx_hyp 関数から atomic 指示文を含むループ (図 3 の C 部) を除外した場合の実行時間を測定し, その差を比較した.

まずは参考として測定したホスト CPU における結果を**図 5** および**図 6** に示す. いずれのグラフも横軸には対象とする行列名とスレッド番号をとっており, 図 5 の縦軸には実行時間, 図 6 の縦軸には atomic 指示文を含むプログラムに対する atomic 指示文を含まないプログラムの実行時間の割合を示している. 図 5 を見る限り,atomic 指示文の影響は軽微であり, またスレッド間の実行時間のばらつきも特に大きくはないことがわかる. 図 6 では縦軸の値が小さいほど atomic 指示文による影響が大きいことを意味している. わずかに 1.0 を越えてしまっている, すなわち atomic 処理部を排除した方が実行時間が長くなってしまっているスレッドも存在するが, その増加本数や増加具合は僅かであることから, 誤差の範囲と考えて問題無いだろう. atomic 演算を除外したことによる実行時間の減少分は, 一番影響の大きな human_1x1 行列でも 4%程度, その他の行列ではせいぜい 2%程度であった. 以上から, ホスト CPU 上での実行においては atomic 指示文を用いたことによる影響はほとんどないと考えて良いだろう.

一方で MIC における結果を確認したところ, スレッド数が 60 や 120 の場合には CPU と似た傾向の結果が得られた一方, スレッド数が 180 や 240 になると性能比のばらつきが大きくなった. **図 7** から**図 11** にはスレッド数 240 の場合の各行列における実行時間比を示す. グラフが激しく上下しているのみならず, グラフによっては非常に大きな時間比となっている線も見受けられる. これらの結果から, 少なくとも MIC においてはスレッド間の負荷の均等化がうまく行えていないことがわかる. 全体の実行時間という観点からは全てのスレッドの実行時間が均一である必要性はなく, 他のスレッドよりも実行時間が非常に長いスレッドがなければ良いことも事実であるが, スレッド毎の実行時間のばらつきが大きいことから性能改善の余地があることが期待される. MIC を用いた 120 スレッド以上での実行では状況に応じてスレッドが動的に切り替わって実行されることを考慮すると, スレッドの切り替わりのタイミングなどによって実行時間がばらついてしまうために実行毎のスレッド単位の実行時間が安定せず, このような結果となってしまった可能性がある.

3.3.4　ループスケジューリングの最適化

OpenMP におけるスレッド間の実行時間のばらつきを

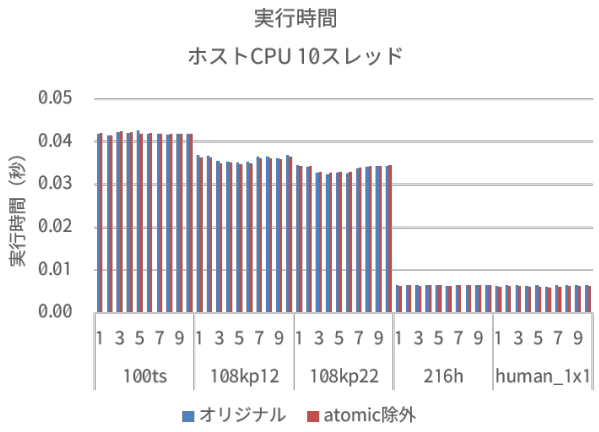

図 5　atomic 指示文の影響 (ホスト CPU, 実行時間)

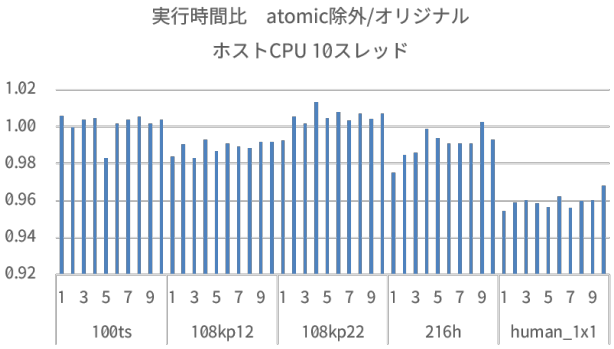

図 6　atomic 指示文の影響 (ホスト CPU, 実行時間比)

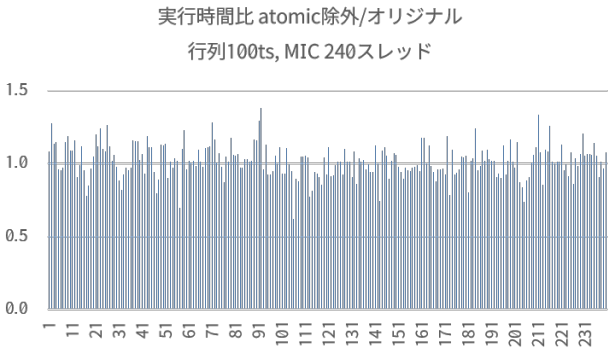

図 7　atomic 指示文の影響 (MIC, 行列 100ts)

解消する一般的な手段としては, 動的なスケジューリングの活用が知られている．これは OpenMP のループ並列化 (omp for 節および omp do 節) において schedule 節を指定する, 特に dynamic または guided のスケジューリングを指定することで実現が可能である．ところが, 今回の対象プログラムの場合, 前項にて示したようにそもそも omp do 節を用いた並列化を行っていない．しかし, あらかじめ行われていたスレッドへの計算割り当ての情報を無視して単純な do ループ構造に変更することは容易に可能である．

そこで, 単純な do ループ構造に変換したうえで omp do 節を用いた並列化を行い, さらに schedule 節を用いていく

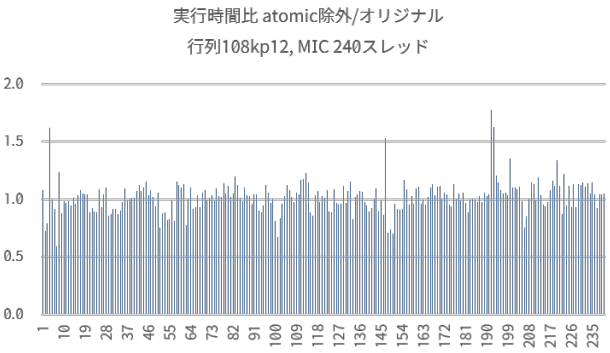

図 8　atomic 指示文の影響 (MIC, 行列 108kp12)

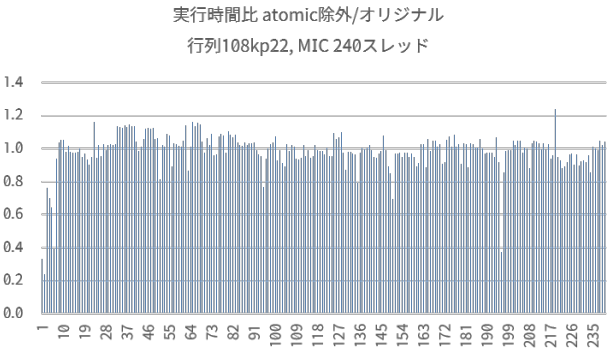

図 9　atomic 指示文の影響 (MIC, 行列 108kp22)

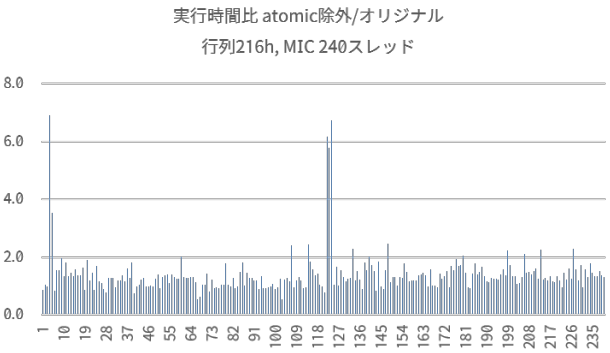

図 10　atomic 指示文の影響 (MIC, 行列 216h)

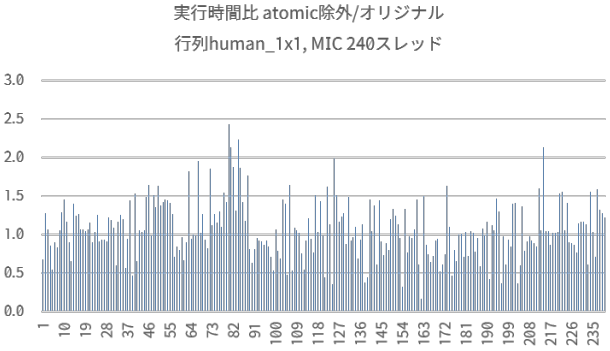

図 11　atomic 指示文の影響 (MIC, 行列 human_1x1)

つかのスケジューリング設定による性能を確認した. **図 12** はそれぞれの行列とスレッド数設定において, スケジューリング種別 {static, dynamic, guided} それぞれについてチャンクサイズ {1, 5, 10, 50, 100, 500, 1000, 5000, 10000} で実行した際の実行時間をオリジナル実装の実行時間との比で表した一覧である. 数値の小さいものはオリジナルよりも高速に実行できたことを意味する. 108kp22 MIC(240) のようにいずれのスケジューリング設定でも性能が向上しなかったものもあるが, 多くの行列・スレッド数で性能が向上した例があることがわかる. 特に dynamic や guided のチャンクサイズ 50 周辺での性能向上度合いが高い. 最も高い性能が得られた例としては, 行列 human_1x1 の 180 スレッド実行においてチャンクサイズ 50 で 0.59, すなわち 41%の実行時間が削減できた.

一方図 12 は同じ測定結果に対して, 分母をオリジナル実装のスレッド数 {60,120,180,240} の中で一番高速だったものに変更したものである. この比較方法でも行列 100ts, 216h, human_1x1 では性能が向上した例が残った一方, 行列 108kp12 や 108kp22 では性能向上したものがなくなってしまった. 最も高い性能が得られた例としては, やはり行列 human_1x1 の 180 スレッド実行においてチャンクサイズ 50 で 0.67, すなわち 33%の実行時間が削減できた.

以上の結果から, 行列によってはスケジューリング設定の最適化により性能向上が得られることが確認できた. ただし Hyper-Threading 環境下で動的なスケジューリングを用いている都合上, これらの性能は安定して得られるとは限らず, ある程度実行毎にばらついてしまう点には注意せねばならない.

3.3.5 ベクトル化の促進

MIC は 512bit 長のベクトル命令 (ストリーミング SIMD 拡張命令) に対応しており, これを十分に活用できるプログラムとそうではないプログラムでは性能に大きな性能が生じる. 最適化レポートのためのコンパイラオプション ($-qopt-report$) を付加して対象プログラムをコンパイルしたところ, 階層型行列ベクトル積内における低ランク行列や小密行列を用いた計算はいずれも既にベクトル化されていたことが確認できた. これは計算内容が単純であることから期待通りである.

一方で, ベクトル化可能であると判断されたプログラムであっても対象となるデータのアラインメントやループ回数の都合により実際にはベクトル計算が行えない場合や最適ではないベクトル命令が実行されてしまうことがある. 具体的には, 端数処理のために非ベクトル命令が実行されたり, バイト境界をまたぐため最大のベクトル長でデータをコピーできない, などの状況が発生し, これらの問題が多い場合と少ない場合では性能に有意な差が生じる. これらの問題の解決策としては, アラインメントを改善するコンパイラオプションや指示文の利用, 対象データを配置する配列を分けたりパディングしたりする, などの方法が考えられる.

今回の対象プログラムの場合は, すでにコンパイラオプションに $-alignarray64byte$ が指定されているため, 配列の先頭要素に対するアラインメントの問題は解消されている. ただし図 3 の最内ループ内の計算を見ると確認できるが, ループ処理時に配列インデックスが必ず 1 から始まる a1 や a2 および zbut といった配列はともかく, 同時に用いている zu や zaut のインデックスは nstrtt や nstrtl といった変数によって開始点が変わってしまう. そのため, これらの配列へのアクセスについては 512bit アドレス境界を常にまたがないようにすることができない. この問題を解決する方法としては, 例えばアラインメントの問題が生じない一時的な配列にコピーするという方法が考えられるが, 計算時間自体が短い処理であるため, コピーによる性能ペナルティが大きく適していない. 本稿では扱っていない範囲である階層型行列の生成処理を変更するという方法も考えられるが, これについては今後の課題とする.

また配列のパディングを行い常にループの回数を 8 などの倍数に揃えることでベクトル化やループアンローリングなどの際に端数部の処理を省略することができる. ただし上記のアラインメントの問題が解消していない状態でパディングを行うと計算回数が増加し実行時間も増加するという問題が生じることがある. 今回のプログラムにおいて 8 の倍数でループを実行できるようパディングを入れた結果, 実行時間は 10%から 20%低下してしまった.

上記以外にも指示文を用いてアラインメントが確実に揃っている配列を指定することでコンパイラにできる限り最適な SIMD 命令を生成させるという最適化手法が考えられる. 例えば図 3 の A 部の一つ目の it ループでは zbut および a1 のアクセスについては確実にアラインメントの問題が発生しない. しかし a1 は構造体メンバであるという都合により assume_aligned 指示文を用いることができず, zbut のみの指定では有意な性能向上が得られなかった. なお, ポインタ参照で置き換えることで a1 に対する指示文の適用が可能であるが, ポインタ参照を用いると性能が大きく低下するという現象が確認されている.

3.3.6 ブロック化

キャッシュアクセスの範囲を狭めることができるブロック化は長いループを処理する際に効果的な最適化手法である. 今回の対象プログラムの場合は, 低ランク行列を用いた行列ベクトル積と小密行列を用いた行列ベクトル積の両方に対してブロック化が可能である. ただし今回の対象プログラムの場合, 低ランク行列を用いた行列ベクトル積では一辺の長さが短い行列が多く用いられており, 小密行列を用いた行列ベクトル積では使用する行列が小さい. そのため, すでにブロック化を行った場合とあまり変わらない状況となってしまっており, 改めてブロック化を行っても性

対象行列とスレッド数

スケジューリング種別とチャンクサイズ		100ts				108kp12				108kp22				216h				human_1x1			
		MIC(60)	MIC(120)	MIC(180)	MIC(240)	MIC(60)	MIC(120)	MIC(180)	MIC(240)	MIC(60)	MIC(120)	MIC(180)	MIC(240)	MIC(60)	MIC(120)	MIC(180)	MIC(240)	MIC(60)	MIC(120)	MIC(180)	MIC(240)
static	1	1.13	0.91	0.98	0.96	1.28	1.27	1.33	1.52	1.16	1.24	1.47	1.26	1.17	1.13	1.00	1.11	0.99	0.91	0.86	1.01
	5	0.98	0.90	0.96	1.00	1.30	1.63	1.82	1.79	1.25	1.38	1.72	1.17	1.08	1.04	0.96	1.18	1.11	0.91	0.90	0.97
	10	0.92	0.93	1.00	1.01	1.50	1.52	1.85	1.78	1.17	1.28	1.50	1.66	1.04	1.03	1.06	1.14	0.88	0.98	0.73	1.05
	50	0.98	0.95	1.00	1.12	1.37	1.53	1.77	1.53	1.14	1.20	1.36	1.40	0.96	0.98	1.20	0.97	0.87	0.87	0.82	1.10
	100	0.99	1.06	1.02	1.20	1.43	1.60	1.82	1.74	1.17	1.40	1.53	1.32	1.06	1.18	1.11	0.99	0.94	0.98	0.96	1.05
	500	0.98	1.07	1.28	1.06	1.43	1.69	1.89	1.95	1.19	1.47	1.59	1.52	1.08	1.18	1.34	1.50	1.18	1.11	1.24	1.58
	1000	1.06	0.99	1.46	1.00	1.54	1.81	2.14	2.23	1.29	1.59	1.84	1.96	1.09	1.92	2.28	2.50	1.17	2.21	2.02	2.83
	5000	1.27	1.99	3.08	3.99	1.56	2.50	3.99	3.93	1.33	2.33	3.26	3.84	4.62	9.05	10.08	10.95	4.43	7.77	8.68	10.40
	10000	2.22	4.10	6.46	7.75	2.53	4.12	5.82	6.75	2.24	4.01	5.73	6.93	9.08	16.20	20.89	21.46	8.69	15.10	16.50	19.84
dynamic	1	1.28	1.18	1.15	1.16	1.43	1.38	1.48	1.49	1.34	1.43	1.39	1.72	1.38	1.55	1.37	1.34	1.23	1.24	1.05	0.92
	5	0.84	0.76	0.83	0.83	0.94	0.92	1.04	1.20	0.90	0.98	1.12	1.09	0.90	0.95	0.91	0.88	0.80	0.79	0.69	0.72
	10	0.82	0.76	0.80	0.79	0.93	0.97	1.25	1.28	0.87	1.00	0.99	1.15	0.86	0.84	0.85	0.85	0.77	0.73	0.68	0.67
	50	0.82	0.82	0.88	0.88	0.89	0.99	1.21	1.25	0.83	0.89	1.02	1.10	0.86	0.87	0.78	0.71	0.77	0.72	0.59	0.71
	100	0.87	0.88	0.91	0.99	0.97	1.10	1.24	1.34	0.84	1.09	1.13	1.21	0.89	0.92	0.92	0.89	0.82	0.85	0.80	0.88
	500	0.89	0.94	1.08	1.09	0.98	1.11	1.43	1.59	0.85	1.08	1.13	1.31	0.97	1.18	1.42	1.47	0.95	1.16	1.12	1.38
	1000	0.95	0.93	1.25	1.07	0.94	1.26	1.65	1.66	0.93	1.12	1.32	1.46	1.12	1.90	2.28	2.60	1.13	1.88	1.88	2.39
	5000	1.19	2.10	3.15	3.41	1.57	2.50	2.98	3.48	1.31	2.13	2.76	3.30	4.64	8.27	9.99	12.19	4.43	7.79	7.60	8.15
	10000	2.14	3.77	5.16	6.05	2.55	3.71	4.84	5.42	2.23	3.71	4.96	5.69	9.05	16.36	19.40	20.26	8.65	14.81	17.29	18.88
guided	1	0.83	0.83	0.90	0.94	1.01	1.37	1.44	1.53	0.82	1.00	1.22	1.37	0.82	0.93	0.92	0.92	0.76	0.80	0.75	0.88
	5	0.83	0.84	0.88	0.93	1.02	1.22	1.45	1.52	0.88	1.03	1.16	1.32	0.83	0.89	1.07	0.94	0.75	0.79	0.64	0.79
	10	0.83	0.82	0.91	0.91	0.99	1.26	1.50	1.52	0.91	1.04	1.20	1.24	0.86	0.87	0.85	0.77	0.74	0.80	0.66	0.72
	50	0.82	0.82	0.87	0.92	1.01	1.25	1.52	1.56	0.81	1.04	1.20	1.28	0.85	0.92	0.86	0.80	0.76	0.77	0.64	0.69
	100	0.86	0.83	0.91	1.01	1.01	1.25	1.45	1.40	0.92	1.03	1.23	1.20	0.85	0.91	0.81	0.97	0.79	0.79	0.79	0.85
	500	0.88	0.89	1.04	0.95	1.00	1.29	1.53	1.59	0.91	1.05	1.24	1.21	1.01	1.15	1.41	1.42	0.97	1.07	1.13	1.31
	1000	0.94	0.97	1.19	1.07	1.00	1.29	1.62	1.79	0.92	1.05	1.30	1.62	1.07	1.88	2.32	2.54	1.13	2.01	2.09	2.25
	5000	1.21	2.06	2.89	3.85	1.57	2.80	3.34	3.21	1.32	2.32	2.75	3.19	4.62	8.30	9.78	10.91	4.44	7.85	8.58	10.31
	10000	2.18	3.83	5.31	6.10	2.53	3.72	5.10	5.55	2.51	4.00	5.27	6.18	9.02	15.67	18.65	22.79	8.63	14.83	14.53	17.44

図 12　スケジューリング設定の影響 (オリジナル実装の同一スレッド数に対する実行時間比)

対象行列とスレッド数

スケジューリング種別とチャンクサイズ		100ts				108kp12				108kp22				216h				human_1x1			
		MIC(60)	MIC(120)	MIC(180)	MIC(240)	MIC(60)	MIC(120)	MIC(180)	MIC(240)	MIC(60)	MIC(120)	MIC(180)	MIC(240)	MIC(60)	MIC(120)	MIC(180)	MIC(240)	MIC(60)	MIC(120)	MIC(180)	MIC(240)
static	1	1.92	1.12	1.02	0.96	2.14	1.53	1.33	1.53	2.26	1.60	1.57	1.26	1.75	1.14	1.00	1.21	1.42	0.91	0.97	1.12
	5	1.67	1.10	1.00	1.00	2.17	1.97	1.82	1.80	2.44	1.78	1.83	1.17	1.62	1.06	0.96	1.29	1.59	0.91	1.01	1.07
	10	1.57	1.15	1.03	1.01	2.50	1.83	1.85	1.78	2.28	1.65	1.60	1.66	1.56	1.05	1.06	1.25	1.26	0.98	0.82	1.16
	50	1.67	1.17	1.03	1.12	2.29	1.85	1.77	1.54	2.22	1.55	1.45	1.40	1.44	0.99	1.20	1.06	1.25	0.87	0.92	1.22
	100	1.70	1.30	1.06	1.20	2.38	1.93	1.82	1.75	2.29	1.81	1.63	1.32	1.59	1.19	1.11	1.09	1.34	0.98	1.08	1.16
	500	1.67	1.32	1.33	1.06	2.38	2.04	1.89	1.96	2.32	1.90	1.69	1.52	1.61	1.19	1.34	1.64	1.69	1.11	1.39	1.75
	1000	1.80	1.22	1.52	1.00	2.57	2.18	2.14	2.24	2.52	2.04	1.96	1.96	1.64	1.94	2.28	2.74	1.67	2.21	2.28	3.13
	5000	2.16	2.46	3.19	3.99	2.60	3.01	3.99	3.94	2.59	3.01	3.47	3.84	6.93	9.18	10.08	11.98	6.36	7.77	9.78	11.49
	10000	3.79	5.06	6.70	7.75	4.21	4.96	5.82	6.77	4.38	5.17	6.12	6.93	13.60	16.42	20.89	23.47	12.46	15.10	18.60	21.93
dynamic	1	2.19	1.45	1.19	1.16	2.39	1.66	1.48	1.50	2.61	1.85	1.49	1.72	2.06	1.57	1.37	1.46	1.76	1.24	1.19	1.01
	5	1.44	0.94	0.86	0.83	1.57	1.11	1.04	1.21	1.75	1.27	1.20	1.09	1.35	0.96	0.91	0.97	1.14	0.79	0.77	0.79
	10	1.40	0.94	0.83	0.79	1.56	1.17	1.25	1.29	1.69	1.29	1.06	1.15	1.29	0.85	0.85	0.93	1.10	0.73	0.77	0.74
	50	1.40	1.01	0.91	0.88	1.49	1.19	1.21	1.25	1.63	1.15	1.09	1.10	1.28	0.88	0.78	0.78	1.11	0.72	0.67	0.78
	100	1.47	1.09	0.95	0.99	1.62	1.33	1.24	1.35	1.63	1.40	1.20	1.21	1.34	0.94	0.92	0.98	1.17	0.85	0.90	0.97
	500	1.52	1.16	1.11	1.09	1.63	1.34	1.43	1.59	1.65	1.39	1.21	1.31	1.45	1.19	1.42	1.61	1.36	1.16	1.26	1.52
	1000	1.62	1.14	1.30	1.07	1.56	1.51	1.65	1.66	1.80	1.45	1.41	1.46	1.68	1.93	2.28	2.84	1.62	1.88	2.12	2.64
	5000	2.02	2.59	3.27	3.41	2.61	3.01	2.98	3.49	2.56	2.75	2.95	3.30	6.96	8.38	9.99	13.33	6.35	7.79	8.57	9.00
	10000	3.65	4.65	5.35	6.05	4.24	4.47	4.84	5.43	4.35	4.78	5.29	5.69	13.57	16.58	19.40	22.16	12.40	14.81	19.49	20.87
guided	1	1.42	1.03	0.93	0.94	1.69	1.65	1.44	1.53	1.59	1.29	1.30	1.37	1.23	0.94	0.92	1.00	1.09	0.80	0.84	0.97
	5	1.41	1.03	0.91	0.93	1.69	1.47	1.45	1.52	1.73	1.33	1.24	1.32	1.25	0.90	1.07	1.03	1.07	0.79	0.72	0.87
	10	1.42	1.01	0.94	0.91	1.66	1.52	1.50	1.52	1.77	1.35	1.28	1.24	1.29	0.88	0.85	0.85	1.05	0.80	0.74	0.79
	50	1.40	1.01	0.91	0.92	1.68	1.51	1.52	1.56	1.59	1.34	1.28	1.28	1.28	0.93	0.86	0.87	1.09	0.77	0.72	0.77
	100	1.46	1.02	0.95	1.01	1.68	1.50	1.45	1.41	1.80	1.32	1.32	1.20	1.27	0.92	0.81	1.06	1.13	0.79	0.89	0.94
	500	1.50	1.10	1.08	0.95	1.67	1.55	1.53	1.59	1.77	1.35	1.32	1.21	1.51	1.17	1.41	1.55	1.38	1.07	1.27	1.45
	1000	1.60	1.20	1.24	1.07	1.67	1.55	1.62	1.80	1.79	1.36	1.39	1.62	1.61	1.91	2.32	2.78	1.62	2.01	2.36	2.49
	5000	2.06	2.54	3.00	3.85	2.61	3.38	3.34	3.22	2.57	2.99	2.93	3.19	6.93	8.41	9.78	11.93	6.37	7.85	9.67	11.40
	10000	3.71	4.72	5.51	6.10	4.22	4.48	5.10	5.56	4.90	5.16	5.62	6.18	13.52	15.88	18.65	24.93	12.38	14.83	16.39	19.27

図 13　スケジューリング設定の影響 (オリジナル実装の最速スレッド数に対する実行時間比)

能が改善しない可能性がある．逆にそもそもブロックに収まるような行列を計算する場合は, 大きさの判定などの処理が増えることで性能がわずかに低下してしまう．MIC はプロセッサ単体の処理性能が低く分岐処理も不得意であるため, 性能低下の影響はより大きくなると考えられる．本稿における階層型行列ベクトル積は計算対象となる行列の大きさが一定ではないため, 性能向上と性能低下の結果として全体としての性能が向上するかどうかは問題依存となる．

実際にブロック化を行った結果, やはり劇的な効果は得られなかったが, いくつか性能が向上した例も見られた．**図 14** は小密行列ベクトル積の内側ループに対するブロック化, **図 15** は小密行列ベクトル積の外側ループに対するブロック化の結果である．縦軸はオリジナル実装における実行時間に対する実行時間比をとっており, オリジナル・ブロック化版ともにスレッド数 60,120,180,240 のうち最も高速であったものを選択して比較した結果を用いている．最も高い効果が得られたのは 108kp22 行列に対して it ループ

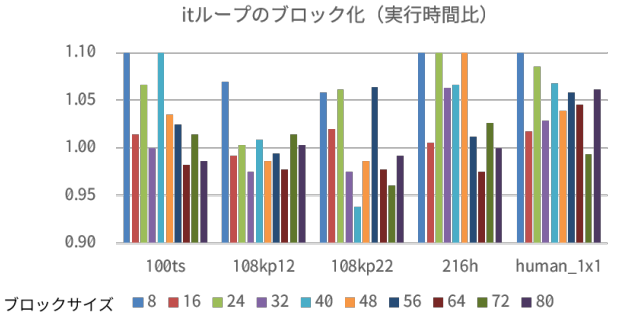

図 14　小密行列ベクトル積のブロック化: it ループ (内側ループ)

(内側ループ) をブロックサイズ 40 でブロック化したケースであり, 6.1%の実行時間が削減できた．一方, 低ランク行列に対する行列ベクトル積については既に一辺が短い行列を用いているためか性能向上を得ることはできなかった．

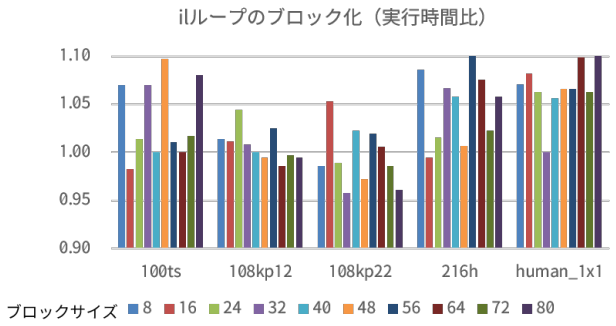

図 15 小密行列ベクトル積のブロック化: il ループ (外側ループ)

4. おわりに

本稿では静電場解析問題における階層型行列を用いた計算を題材として, メニーコアプロセッサ (Xeon Phi 5110P) 上にて階層型行列ベクトル積の性能評価を行った. マルチコア CPU(IvyBridge-EP) と性能を比較した結果, 全体的にマルチコア CPU の方がやや高速であった. またメニーコアプロセッサの性能は 120 スレッド利用時に最も高速という傾向が見られた. さらにメニーコアプロセッサ上での実行について調査したところ, 対象問題の並列度は十分高かった一方, スレッド間の実行時間のばらつきはマルチコア CPU 上での実行と比べて非常に大きかった. スレッド数が多いことから atomic 演算のコストが大きい可能性を考えたが, 詳細に実行時間を確認した結果その影響は軽微であり, むしろスレッド毎の実行時間のばらつきが大きいことが影響している可能性が確認された.

MIC における階層型行列ベクトル積の性能を向上させるために幾つかの項目について性能分析と最適化の試行を行った.

スレッド毎の実行時間のばらつきを軽減するために OpenMP の schdule 指示節を用いて動的なスケジューリング設定を行い性能を調査した. その結果, 同じスレッド数での比較においては human_1x1 行列の 180 スレッド実行にて最大 41%, スレッド数を問わない最大性能での比較においては同じ行列・スレッド数にて最大 33%の実行時間が削減できる例が確認できた. 小密行列ベクトル積についてブロック化を行ったところ, 行列 108kp22 行列において最大で 6.1%の性能が向上した. 小密行列についても低ランク行列についても十分に小さいサイズでの計算が多く行われる都合上, ブロック化の効果が限定的である点については納得できる結果であった. これらの性能向上は様々なパラメタを試した中で特に良かったものを抽出した例であり, 何らかの方法により最適なパラメタを導出できるかなどについては今後の検討課題である. さらに, ベクトル化を促進するためにコンパイラのメッセージやアラインメントなどについて調査を行った. その結果, 現在のプログラムは SIMD 命令自体は生成されているが, アラインメントの調整が困難な部分も存在することが確認された. プログラムの変更によってさらにうまくアライメントを調整できるかについては検討中である.

以上のように, メニーコアプロセッサ Xeon Phi 5110P を用いた最適化は限定的ながら効果が確認できた. 今後は最新のメニーコアプロセッサ（開発コードネーム Knights Landing）でも同様の評価や最適化を実施し, 現行プロセッサとの違いなどについて調査や評価を行う予定である. これらの異なる環境における評価においては, 計算対象となる行列のパラメタ等についても考慮したい. 例えば近似行列の精度などを変更すると, 計算対象となる要素数やリーフのサイズが変わるため, 各種の最適化手法にの効き具合にも影響する. これらの傾向等についても比較したいと考えている. また GPU 上での評価とハードウェア毎の性能の比較についても実施予定である.

謝辞 日頃より最適化プログラミングについて議論をさせていただいている東京大学情報基盤センタースーパーコンピューティング研究部門の皆様に感謝します. 本研究は科学技術振興機構戦略的創造研究推進事業 (JST/CREST), German Priority Programme 1648 Software for Exascale Computing (SPPEXA-II), 大規模学際情報基盤共同利用・共同研究拠点 (JHPCN) の支援を受けています.

参考文献

[1] 導入するスーパーコンピュータシステムの決定について - 最先端共同 HPC 基盤施設（JCAHPC）, http://jcahpc.jp/pr/pr-20160510.html.
[2] Oakforest-PACS リリースアナウンス, http://www.cc.u-tokyo.ac.jp/system/ofp/release-20160510.html.
[3] Börm, S., Grasedyck, L., Hackbusch, W.: Hierarchical matrices. Lecture note No. 21 of the Max Planck Institute for Mathematics in the Sciences (2003).
[4] ppOpen-HPC — Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications on Post-Peta-Scale Supercomputers with Automatic Tuning (AT), http://ppopenhpc.cc.u-tokyo.ac.jp/ppopenhpc/.
[5] Akihiro Ida and Takeshi Iwashita and Takeshi Mifune and Yasuhito Takahashi: Parallel Hierarchical Matrices with Adaptive Cross Approximation on Symmetric Multiprocessing Clusters, Journal of Information Processing, Vol.22, No.4, pp.642-650 (2014).
[6] Börm S., Grasedyck L. and Hackbusch W.: Hierarchical Matrices, Lecture Note, Max-Planck-Institut fur Mathematik (2006).
[7] Optimization and Performance Tuning for Intel Xeon Phi Coprocessors - Part 1: Optimization Essentials — Intel Software, https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-c
[8] Rezaur Rahman: Intel®Xeon Phi™Coprocessor Architecture and Tools, ISBN: 978-1-4302-5926-8 (Print) 978-1-4302-5927-5 (Online) (2013).