Vol.2018-HPC-165 No.16
2018/7/31

マルチSPMDプログラミング環境における線形ソルバの性能評価

辻 美和子[1,a)]　Gurhem Jérôme[2,3,b)]　Petiton Serge[2,3,4,c)]　佐藤 三久[1,d)]

概要：マルチ SPMD (mSPMD) プログラミングモデルは，複数の SPMD プログラムをワークフローのタスクとして編成する，大規模かつ階層的なシステムのためのプログラミングモデルである．本稿では，ブロック・ガウス消去法，ブロック LU 分解，ブロック・ガウス・ジョルダン法の 3 つの線形ソルバを mSPMD プログラミングモデル上で実装し，性能評価を行った．実験の結果，いずれのアルゴリズムにおいても，ワークフローによるタスク並列と SPMD による分散並列を適切に組み合わせることで，より効率的にソルバを実行できることがわかった．また，演算数は少ないがタスク並列度が低いブロック・ガウス消去法，ブロック LU 分解よりも，演算数は多いがタスク並列度が高いブロック・ガウス・ジョルダン法のほうがしばしば高速であった．

1. はじめに

次世代のスーパーコンピュータは，NUMA などの複雑なトポロジ構造を持つ高密度なメニーコア・ノードが多数連結された，大規模かつ階層的なアーキテクチャになると考えられる．そのような大規模システムにおいては，MPI+OpenMP などの既存のプログラミングモデルでは，MPI プロセス数や OpenMP スレッド数が多くなりすぎるなどの原因から，大規模アプリケーションの効率が低下するおそれがある．そこで，著者らは，階層的かつ大規模なアーキテクチャを効率的に利用するためのプログラミングモデルとして，ワークフローと分散並列/スレッド並列を組み合わせて利用するマルチ SPMD (mSPMD) プログラミングモデルを提案し，mSPMD プログラミングモデルに基づくアプリケーションの開発・実行環境を開発してきた [4], [6]．mSPMD プログラミングモデルでは，複数の並列プログラムをワークフローのタスクとして編成することで，

(1) 大規模 SPMD プログラムをワークフローの枠組みを用いて複数の中規模 SPMD プログラムに分割し，後に結果を集約することで，大規模 SPMD プログラムにおける通信のオーバーヘッドを削減する
(2) ワークフロープログラムにおいてボトルネックとなる逐次タスクを分散並列化することで全体の高速化をはかる

などの効果が期待される．

本稿では，mSPMD プログラミングモデル上で動作するアプリケーションの性質を調査するために，ブロック・ガウス消去法，ブロック LU 分解，ブロック・ガウス・ジョルダン法の 3 つの線形ソルバを実装し，性能評価を行った．実験の結果，いずれのアルゴリズムにおいても，ワークフローによるタスク並列と SPMD による分散並列を適切に組み合わせることで，より効率的にソルバを実行できることがわかった．また，演算数は少ないがタスク並列度が低いブロック・ガウス消去法，ブロック LU 分解よりも，演算数は多いがタスク並列度が高いブロック・ガウス・ジョルダン法のほうがしばしば高速であった．

2. 背景：マルチSPMDプログラミング開発実行環境

ノード内 NUMA などの複雑な内部トポロジを持つ多数のメニーコアーノードから構成された大規模システムにおいて効率的にアプリケーションを実行するためには，階層的アーキテクチャレベルに合わせた複数のプログラミングモデルを利用することが重要である．著者らは，このような大規模かつ階層的なアーキテクチャに対するプログラミングモデルとして，ワークフローと分散並列/スレッド

[1] 理化学研究所計算科学研究センター
RIKEN Cenetr for Computational Science
[2] Maison de la Simulation
[3] Centre National de la Recherche Scientifique
[4] Université de Lille
[a)] miwako.tsuji@riken.jp
[b)] jerome.gurhem@polytech-lille.net
[c)] Serge.Petiton@univ-lille1.fr
[d)] msato@riken.jp

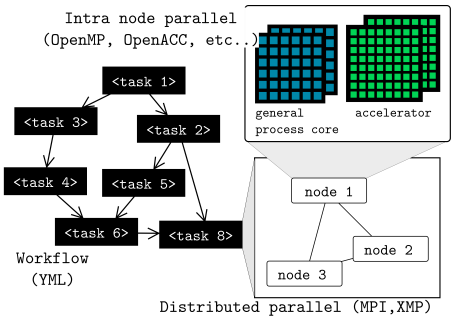

図 1　マルチ SPMD (mSPMD) プログラミングモデルの概要

```
double A[12];

#pragma xmp nodes p(4)                                  Data Mapping
#pragma xmp template t(0:11)
#pragma xmp distrubite t(block) onto p
#pragma xmp align A[i] with t(i)

  a one-dimensional block-distributed array A[]
  distributed over four processes
          0  1  2  3  4  5  6  7  8  9 10 11
Node1     ■  ■  ■
Node2              ■  ■  ■
Node3                       ■  ■  ■
Node4                                ■  ■  ■

#pragma xmp loop (i) on t(i)        Work Mapping
for(i=0;i<12;i++){
   A[i]=A[i]+1.0;
}
```

図 2　XMP により記述されたソースコードの例

並列を組み合わせて利用するマルチ SPMD (mSPMD) プログラミングモデルを提案し，mSPMD プログラミングモデルに基づくプログラムの開発・実行環境を開発してきた [4], [6].

図 1 に mSPMD プログラミングモデルの概要を示す．mSPMD プログラミングモデルにおいては，ワークフローのタスクは分散並列プログラムもしくは分散並列とスレッド並列のハイブリッドプログラムとして実行される．ワークフローを実行するために YML が用いられ，ワークフロー内のタスクを記述するために MPI に加えて XMP がサポートされる．

YML[1], [2] は，Delannoy らによって開発された科学技術計算のためのワークフロー開発実行環境であり，タスクどうしの依存関係を記述するワークフロー記述言語 YvetteML をサポートする．YML は YvetteML で記述されたワークフローを解釈してタスクどうしの依存関係を示す DAG を生成し，DAG に従ってタスクを実行する．オリジナルの YML は，逐次言語で記述されたタスクを P2P 環境や小規模クラスタで実行することを想定していたが，著者らが並列言語で記述されたタスクを大規模システムで実行できるように，これを拡張し，ミドルウエアを開発した．YvetteML での記述例については，3.3 で述べる．

XcalableMP (XMP) [3] は，指示文に基づく分散並列プログラミング言語である．XMP コンパイラは，XMP 指示文とベース言語 (C もしくは Fortran) で記述されたソースコードを，XMP ランタイムライブラリ呼出しを含むベース言語のソースコードに変換する．XMP ランタイムライブラリは，通信レイヤとして主に MPI を用いて通信を行う．**図 2** に，XMP により記述された分散並列のソースコードの例を示す．XMP では，**template** と呼ばれる仮想配列の **nodes** 上への分散を定義し，**align** 文を用いて配列と template と対応付けることで，配列のノード上への分散を定義する．また，処理の分散は，**loop** と template を用いて定義される．

mSPMD プログラミングモデルの利点として，

- 大規模分散並列プログラムをワークフローの枠組で中規模プログラムに分割し，後に結果を統合することで，通信のオーバーヘッドを防ぐことができる
- ワークフローにおいてボトルネックとなっていたタスクを分散並列を用いて並列化し，全体を高速化する
- 既存の並列プログラムや並列ライブラリをワークフローを用いて組み合わせることで，複雑なアプリケーションを平易に実装できる

などが挙げられる．

3. マルチ SPMD プログラミング開発実行環境における線形ソルバの実装

本章では，本研究で mSPMD プログラミング開発実行環境上に実装した線形ソルバ — ガウス・ジョルダン消去法，ガウス消去法，および LU 分解 — について述べる．

3.1 概要

本研究では，サイズ $n \times n$ の行列 A およびサイズ n のベクトル $\boldsymbol{b}$ が与えられたとき，サイズ n のベクトル $\boldsymbol{x}$ に関して，連立一次方程式 $A\boldsymbol{x} = \boldsymbol{b}$ を解くソルバを考える．以降では，行列およびベクトルの各成分は以下のように記述される：

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ & & \cdots & \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{pmatrix}, \boldsymbol{b} = \begin{pmatrix} b_0 \\ \cdots \\ b_{n-1} \end{pmatrix}$$

3.2 アルゴリズム

ガウス消去法

ガウス消去法は行列 A が三角行列になるまで対角成分より下を繰り返し消去し，得られた上三角行列に後退代入を繰り返して解を得るアルゴリズムである．**図 3** にガウス消去法のアルゴリズムを示す．本来のガウス消去法では，ゼロや微小な値での割り算を防ぐために，もっとも絶対値が

for k **from** 0 **to** $n-2$ **do**
  // $a_{k,k}$ を用いて $k+1$ 行目以降の x_k の係数を消去する
  for i **from** $k+1$ **to** $n-1$ **do**
    for j **from** $k+1$ **to** $n-1$ **do**
      Let $a_{i,j} := a_{i,j} - \left(\frac{a_{i,k}}{a_{k,k}}\right) a_{k,j}$
    end for
    Let $b_i := b_i - \left(\frac{a_{i,k}}{a_{k,k}}\right) b_k$
  end for
end for

for i **from** $n-1$ **to** 0 **do**
  Let $x_i := \left(b_i - \sum_{j=i+1}^{n-1} a_{i,j}x_j\right) / a_{i,i}$
end for

図 3　ガウス消去法のアルゴリズム

for k **from** 0 **to** $p-1$ **do**
  (1) $Inv^{(k)} = [A_{k,k}^{(k)}]^{-1}$
  (2) $\boldsymbol{b}_k^{(k+1)} = Inv^{(k)} \cdot \boldsymbol{b}_k^k$
  if $k \neq p-1$ **then**
    for i **from** $k+1$ **to** $p-1$ **do**
      (3) $A_{k,i}^{(k+1)} = Inv^{(k)} \cdot A_{k,i}^{(k)}$
      (4) $\boldsymbol{b}_i^{(k+1)} = b_i^k - A_{i,k}^{(k)} \cdot \boldsymbol{b}_k^{k+1}$
      for j **from** $k+1$ **to** $p-1$ **do**
        (5) $A_{i,j}^{(k+1)} = A_{i,j}^k - A_{i,k}^{(k+1)} \cdot A_{k,j}^{(k)}$
      end for
    end for
  end if
end for

for i **from** $p-1$ **to** 1 **do**
  for j **from** 0 **to** $i-1$ **do**
    (6) $\boldsymbol{b}_j^{(i+j+1)} = \boldsymbol{b}_j^{(i+j)} - A_{i,j}^{(i+1)} \cdot \boldsymbol{b}_j^{(p)}$
  end for
end for

図 4　ブロック・ガウス消去法のアルゴリズム．ただし $A_{i,j}$ は行列 A を $p \times p$ ブロックに分割したときの (i,j) 番目のブロック．$\boldsymbol{b}_i$ はベクトル $\boldsymbol{b}$ を p 分割したときの i 番目のブロック．

大きな係数が $a_{k,k}$ になるように行の入れ替えを行うが，本稿では簡単のために行の入れ替えを考慮しない．これは他のアルゴリズムについても同様である．

線形ソルバには，しばしば行列をいくつかのブロックに分割して処理する，ブロック・アルゴリズムが用いられる．ブロック・アルゴリズムは，単一の行，列，あるいはスカラではなく，複数の行，列，および行列に対して作用する．一般的に計算機は，行列やベクトルの処理に適しているため，ブロック化によってより効率的に解を得ることができる．

図 4 にブロック・ガウス消去用 (Block Gaussian Elimination, BGE) のアルゴリズムを示す．BGE では，$n \times n$ の行列を $p \times p$ ブロックに分割し，対角線上にあるブロック

for k **from** 0 **to** $p-1$ **do**
  (1) $Inv^{(k)} = [A_{k,k}^{(k)}]^{-1}$
  (2) $\boldsymbol{b}_k^{(k+1)} = Inv^{(k)} \cdot \boldsymbol{b}_k^k$
  if $k \neq p-1$ **then**
    for i **from** 0 **to** $p-1$ **do**
      if $i \neq k$ **then**
        (3) $A_{k,i}^{(k+1)} = Inv^{(k)} \cdot A_{k,i}^{(k)}$
        (4) $\boldsymbol{b}_i^{(k+1)} = b_i^{(k+1)} - A_{i,k}^{(k)} \cdot \boldsymbol{b}_k^{(k+1)}$
        for j **from** $k+1$ **to** $p-1$ **do**
          (5) $A_{i,j}^{(k+1)} = A_{i,j}^{(k)} - A_{i,k}^{(k+1)} \cdot A_{k,j}^{(k)}$
        end for
      end if
    end for
  end if
end for

図 5　ブロック・ガウス・ジョルダン消去法のアルゴリズム

クの逆ブロックを求める．得られた逆ブロックを用いて，他の行列やベクトルを更新する．これをすべての p 対角ブロックについて繰り返して上三角行列を得る．続いて，ブロック毎の行列ベクトル演算によって後退代入を行う．

BGE の計算量は，行列サイズ n に対して，$\frac{n^3+3n^2+8n}{3} \sim \frac{n^3}{3}$ である．

ガウス・ジョルダン消去法

ガウス・ジョルダン消去法は，行列 A の対角化を行うことで x を計算する．**図 5** にブロック・ガウス・ジョルダン消去法 (Block Gauss-Jordan Elimination, BGJ) のアルゴリズムを示す．BGJ では，BGE のように後退代入を行わない．

BGJ のアルゴリズムは BGE よりも単純であるが，計算量は行列サイズ n に対して $\frac{n^3+n^2+2n}{2} \sim \frac{n^3}{2}$ であり，BGE よりも増加する．

LU 分解

図 6 にブロック LU 分解 (Block LU factorization, BLU) のアルゴリズムを示す．LU 分解では，行列 A を $A = L \cdot U$ に分解する．ここで，L は下三角行列，U は上三角行列である．次ぎに，$U\boldsymbol{y} = \boldsymbol{b}$ と $L\boldsymbol{x} = \boldsymbol{y}$ を解くことで x を求める．BLU の計算量は，BGE とほぼ同等で $\frac{n^3+3n^2+2n-3}{3} \sim \frac{n^3}{3}$ である．

3.3　YML ワークフローによる線形ソルバの実装

YML ではタスクどうしの依存関係を記述する言語である YvetteML 言語がサポートされる．本稿では，図 4–6 のブロックアルゴリズムを，各ブロックに対する処理を XMP で記述してタスクとして実装し，それらの依存関係を YvetteML で記述した．

図 7 に，BGJ アルゴリズムを YvetteML で記述した例を

```
for k from 0 to p − 2 do
  (1) Inv^(k) = [A_{k,k}^(k)]^{-1}
  for i from k + 1 to p − 1 do
    (2) A_{k,i}^(k+1) = A_{k,i}^(k) · Inv^(k)
    for j from k + 1 to p − 1 do
      (3) A_{i,j}^(k+1) = A_{i,j}^(k) − A_{i,k}^(k+1) · A_{k,j}^(k)
    end for
  end for
end for

for i from 0 to p − 2 do
  for j from i + 1 to p − 1 do
    (4) b_j = b_j − A_{j,i} · b_i
  end for
end for

for k from p − 1 to 0 step −1 do
  (5) solve b_k = A_{k,k} · b_k
  if k ≠ 0 then
    for i from k − 1 to 0 step −1 do
      (6) b_i = b_i − A_{i,k} · b_k
    end for
  end if
end for
```

図 6　LU 分解のアルゴリズム

```
par (k:=0; p-1); do
 par
  wait(Aev[k][k][k]);
  compute XMP_inversion(A[k][k],AInv[k]);
  notify(Aev[k][k][k+1]);
 //
 if(k neq p-1) then
  par(j:= k+1; p-1); do
   par
    wait(Aev[k][k][k+1]); wait(Aev[k][j][k]);
    compute XMP_prodMat(AInv[k],A[k][j]);
    notify(Aev[k][j][k+1]);
   //
   par (i:=0; p-1); do
    if(k neq i) then
     #A[i,j] = A[i,j] - A[i,k] * A[k,j]
     wait(Aev[i][k][k]); wait(Aev[i][j][k]); wait(Aev[k][j][k+1]);
     compute XMP_prodDiff(A[i][k],A[k][j],A[i][j]);
     notify(Aev[i][j][k+1]);
    endif
   enddo
  endpar
 enddo
 endif
 //
 wait(Aev[k][k][k+1]);  wait(Bev[k][k]);
 compute XMP_prodMV(AInv[k],B[k]);
 notify(Bev[k][k+1]);
 //
 par (i:=0; p-1); do
  if(k neq i) then
   #B[i] = B[i] - A[i,k] * B[k]
   wait(Bev[i][k]); wait(Bev[k][k+1]); wait(Aev[i][k][k]);
   compute XMP_prodDiffMV(A[i][k],B[k],B[i]);
   notify(Bev[i][k+1]);
  endif
 enddo
endpar
enddo
```

図 7　YML ワークフローの例 BGJ．BGJ アルゴリズム

示す．図で **compute** 節はタスクの呼出しをしめす．**par (i:=0; p-1); do** ∼ **enddo** は，繰り返しの各 **i** が互いに独立に実行できることを示す．**par** ∼ **endpar** は，この間の//で区切られた各ブロックが互いに独立に実行できることを示す．**notify** は，ユーザによって定義されたイベント (たとえば Aev[i][j][k] など) を通知し，**wait** は，対応するイベントの通知まで待つ必要があることを示す．**wait**–**notify** の組は，タスク間のデータの依存性を記述するために用いられる．タスク A の出力をタスク B が必要とする場合，タスク A の直後でタスク A の終了を **notify** し，タスク B の直前でこれを **wait** する．

3.4　XMP タスクの実装

本節では，ワークフローのタスク内の処理について述べる．mSPMD プログラミングモデルでは，タスクは分散並列モデルに基づいて並列に処理される．

タスクの記述には MPI に加えて，並列プログラミング言語 XMP を用いることができる．XMP による YML タスク記述の例を図 **8** に示す．YML では，タスクは xml で定義される．

mSPMD プログラミングモデルにおいては XMP でタスクを記述する場合，タスクへの入出力となるデータについては，図 2 で示した指示文ではなく，図 8 で示すように <templates> および <distribute> のような xml の要素・属性として定義される．これはワークフローの枠組において，入出力データを適切に各プロセスに分配するためである．現在の実装ではすべての入出力データは，共有ファイルシステムから読み出し (へ書き出し) される．YML のタスクジェネレータは，xml での記述を解釈し，各プロセスが適切な分散データを入出力できるよう，MPI-IO のコードを自動生成する．ただし，IO を介したデータ分配は速度面で問題があることがわかっており [5]，将来的にはデータサーバなどを用いた方法に変更される予定である．その場合でも，入出力データの分散をワークフローのクライアント側とサーバ側とで共有できる枠組を持つことは重要である．

4.　実験

本稿では，3 つの線形ソルバを mSPMD プログラミングモデルで実装し，京コンピュータ上で実行して性能評価を行った．

```
<component type="impl" name="mul" abstract="mul"
          description="A = B x A">
 <impl lang="XMP" nodes="CPU:(16,16)">
  <templates>
   <template name="t" format="block,block"
         size="1024,1024"/>
  </templates>
  <distribute>
   <param template="t" name="B0(1024,1024)"
          align="[i][j]:(j,i)"/>
   <param template="t" name="A0(1024,1024)"
          align="[i][j]:(j,i)"/>
  </distribute>
  <header><![CDATA[
#include<xmp.h>
double A[1024][1024];
double B[1024][1024];

#pragma xmp align A[i][j] with t(j,i)
#pragma xmp align B[i][j] with t(j,i)

#pragma xmp shadow A[*][0]
#pragma xmp shadow B[0][*]
 ]]></header>
<source><![CDATA[
 int      i, j, k, n=1024;
 double dtmp;

 ...
#pragma xmp loop (j,i) on t(j,i)
 for(i=0;i<n;i++){
  for(j=0;j<n;j++){
   B[i][j]=B0[i][j];  } }

 ...
#pragma xmp loop (j,i) on t(j,i)
 for(i=0;i<n;i++){
  for(j=0;j<n;j++){
   A[i][j]=A0[i][j];
   A0[i][j]=0.0;  } }

#pragma xmp reflect (A,B)

#pragma xmp loop (i) on t(j,i)
 for(i=0;i<n;i++){
  for(k=0;k<n;k++){
   if(B[i][k]!=0){
    dtmp=B[i][k];
#pragma xmp loop (j) on t(j,i)
     for(j=0;j<n;j++){
      A0[i][j] += (dtmp*A[k][j]) ;
     }}}}
]]></source><footer /></impl></component>
```

図 8　XMP による YML タスク記述の例. $A := B \times A$ を二次元トポロジのノード上に分散して計算する例

表 1　京コンピュータ諸元

CPU	Fujitsu SPARC64VIIIfx, 8 core, 2.00 GHz
Memory	16GB , 64GB/s
Cache	L1:32+32KB/core, L2:6MB/core
Network	Tofu (6D mesh/torus) Interconnect 5GB/s x 2

4.1　実験環境と問題サイズ

本実験で用いた京コンピュータの諸元を**表 1** に示す．

表 2　ブロック数, ブロックサイズ

16K:16384 × 16384

# of blocks p	2×2	4×4	8×8	16×16
block size	8192^2	4096^2	2048^2	1024^2

32K:32768 × 32768

# of blocks p	2×2	4×4	8×8	16×16
block size	16384^2	8192^2	4096^2	2048^2

表 3　プロセス数

16K:16384 × 16384

total	par task						
1024	16	32	64	128	256	512	1024

32K:32768 × 32768

total	par task						
8192	64	128	256	512	1024	2048	4096

表 4　タスク数 (ブロックやベクトルの初期化を含む)

# of blocks	2×2	4×4	8×8	16×16
BGE	14	60	312	1904
BLU	13	59	311	1903
BGJ	14	64	368	2464

本実験では，3.2 の 3 種類の線形ソルバについて，2 種類のサイズの行列に関して，ブロックサイズおよびタスクごとのプロセス数を変化させ，実行時間を調査した．

用いた行列サイズは, 16384×16384 および, 32768×32768 である．それぞれ，16K および 32K 行列と呼ぶ．この行列を**表 2** で示すように，$2 \times 2 \sim 16 \times 16$ ブロックに分割し，3 種類のアルゴリズムを実行した．また，タスクごとのプロセス数およびワークフロー全体に確保されたプロセス数を**表 3** に示す．全体のプロセス数が 1024 で，タスクごとのプロセス数が 256 だった場合，最大で 4 つのタスクを同時に処理可能である．本実験では，各タスクは，8 コアを有する 1 ノードに 8MPI プロセスを立ち上げるフラット MPI モデルで実行された．

表 4 に各ワークフローにおけるブロック数に対するタスク数を示す．数字は，行列やベクトルの初期化のためのタスクを含む．BGJ は，BGE や BLU と比較して，タスクの数が多く，ブロック数が大きくなるとその差はさらに顕著になる．

4.2　結果

実験の結果を**図 9**–11, 15–17 に示す．これらの図では，それぞれのアルゴリズムの実行時間を Y 軸，タスクごとのプロセス数を X 軸として，異なるブロック数を異なる線で示している．また，前述のように現状の実装はタスクへのデータ IO のオーバーヘッドが大きいことから，IO を省略して適当なダミーデータを使用した場合の実験も行った．この結果を**図 12**–14, 18–20 に示す．キャプションには各

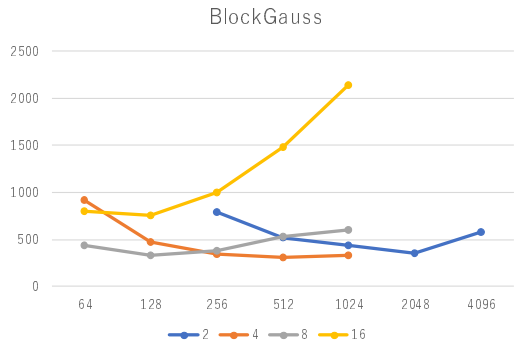

図 9 BGE, 32K, min:4x4,512,306.12(sec)

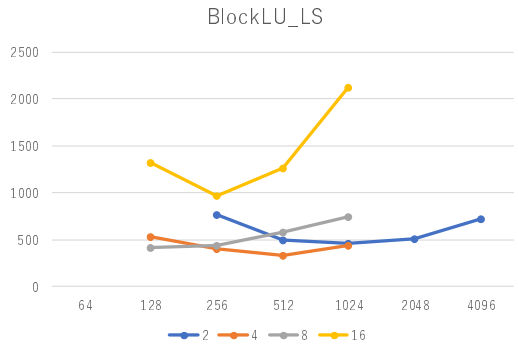

図 10 BLU, 32K, min:4x4,512,330.70(sec)

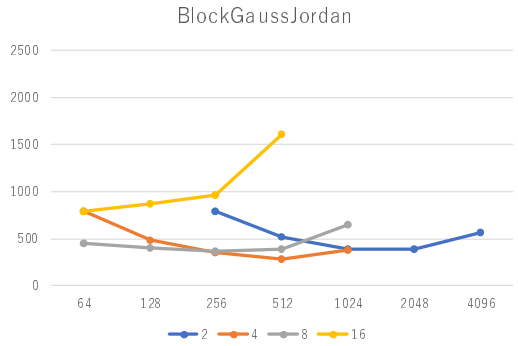

図 11 BGJ, 32K, min:4x4,512,288.88(sec)

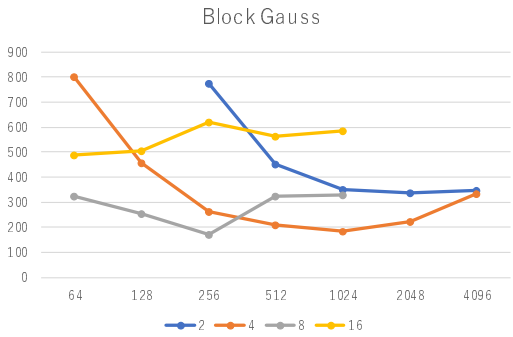

図 12 BGE, 32K, min:8x8,256,170.51(sec)

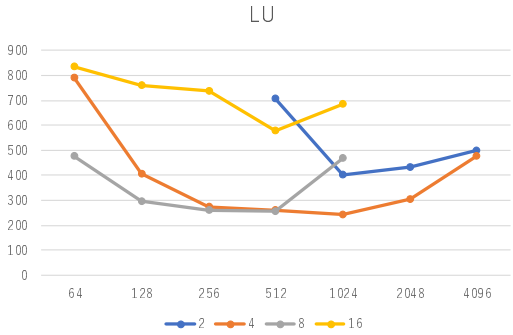

図 13 BLU, 32K, min:4x4,1024,245.18(sec)

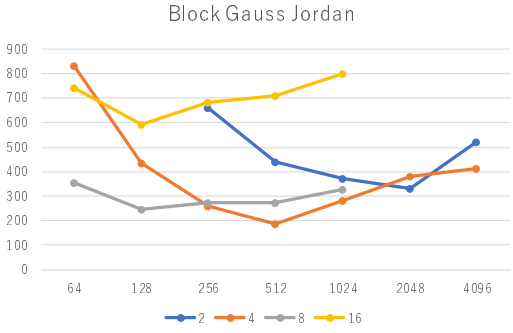

図 14 BGJ, 32K, min:4x4,512,189.67(sec)

アルゴリズムについて，最小の実行時間を与えたブロック数およびタスクごとのプロセス数を併記した．いずれのアルゴリズムにおいても，極端にタスク並列数が大きい場合や，タスク毎のプロセス数が大きい場合よりも，適度なタスク並列数に適度な分散並列数を割り当てた場合のほうが，性能がよいという結果になった．

32K の行列について，IO を行うケースで，すべてのアルゴリズムでもっとも高速だったのは，$4 \times 4 = 16$ 分割されたブロックを処理するタスクに 512 プロセスを割り当てた場合であった．ワークフロー全体に割り当てられたプロセス数は 8192 なので，最大で 16 個のタスクが同時に処理できる．一方，本稿で用いたブロック分割のアルゴリズムは，ある時点で同時に処理が可能になる (Ready になる) タスク数の最大値は，ブロック数と同数である．すなわち，割り当てられたプロセスを同時にすべて使い切ることができる最小のブロック数のときが，もっとも効率的であった．このとき，BLU は 330.7 秒，BGE は 306.12 秒，BGJ は 288.88 秒を要した．LU 分解で連立一次方程式を解く場合，まず LU 分解を行い，続いて 2 つの三角行列を解く必要がある．これらの 3 段階は並列に行うことができない．ゆえに，BLU はもっとも少ない計算量にもかかわらず，実行時間は最小にはならなかった．BGE のタスク数は BLU とほぼ同じである．ただし，LU 分解と異なり，後退代入で解くべき三角行列は 1 つだけである．よって，BGE は，並列化できないクリティカルパスは BLU よりも短く，実行時間も BLU よりも短かった．BGJ は，三角行列を解く後退代入のプロセスがなくクリティカルパスが短い，一方で，対角成分の下のブロックのみを計算する BGE と異なり，対角成分の上も計算する必要があるため計算量やタス

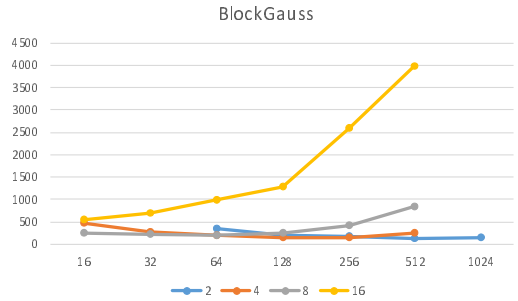

図 **15** BGE, 16K, min:2x2,512,124.07(sec)

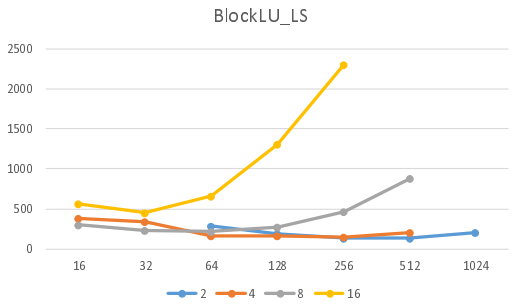

図 **16** BLU, 16K, min:2x2,512,136.12(sec)

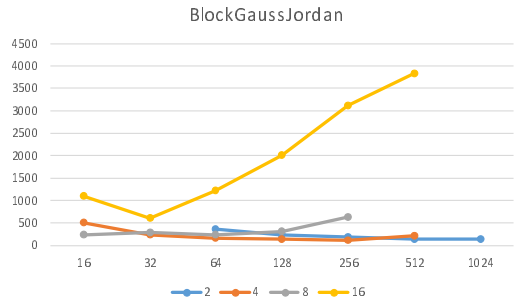

図 **17** BGJ, 16K, min:4x4,256,114.03(sec)

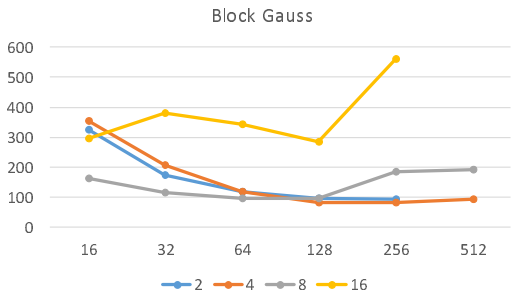

図 **18** BGE, 16K, min:4x4,256,78.78 (sec)

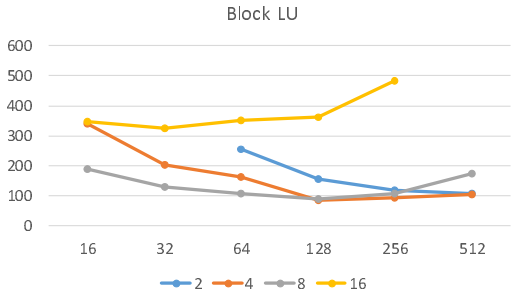

図 **19** BLU, 16K, min:4x4,256,83.12(sec)

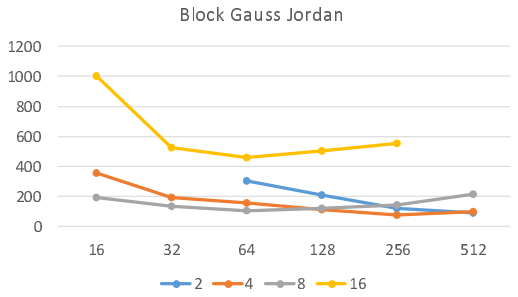

図 **20** BGJ, 16K, min:4x4,128,74.64(sec)

ク数は多い．しかし，これらのブロックの処理は並列に行えるため，もっとも高速に解を得ることができた．

16K の行列について，BGE および BLU では $2 \times 2 = 4$ 分割されたブロックを処理するタスクに 512 プロセスを割り当てた場合が最も高速であった．BGJ では，$4 \times 4 = 4$ 分割されたブロックを処理するタスクに 256 プロセスを割り当てた場合が最も高速であった．これは，タスク並列性の高い BGJ では，BGE や BLU よりも粒度の高いタスク分割がしばしば有効なことがあるからであると考えられる．16K 行列でも最速の場合を比較すると BGJ がもっとも高速であった．

アルゴリズムどうしを比較するために，図 **21**, 22 に，32K および 16K の行列を，2×2, 4×4, 8×8 に分割したときの，タスク毎のプロセス数に対する各アルゴリズムの実行時間の比較を示す．図 21 に示すように，32K 行列では，BLU は多くのケースでもっとも長い実行時間を要した．BGE と BGJ は似た結果を示すが，ブロック数が増加すると BGJ の BGE に対する相対的なタスク数が増すため，BGE のほうが高速な傾向を示した．また，32K 行列では，4×4 分割あるいは 8×8 分割などのときに，タスクごとのプロセス数が増加したとき，BGJ よりも BGE が高速な傾向がみられた．これは，タスクごとのプロセス数が増加すると，1 度に実行できるタスクの数は減少するため，BGJ のタスク並列度の高さがアドバンテージにならないからであると考えられる．

以上から，mSPMD プログラミング・モデル上では，演算数は少ないがタスクどうしの独立性が低いアルゴリズムは，プロセスごとに利用可能なメモリ容量の制限などで 1 タスクに大きなプロセス数を割り当てる必要があり，多数のタスクを同時に実行する余地がない場合には，利用を検討する価値があると考えられる．しかし，一般的には，mSPMD プログラミング・モデルでアルゴリズムを実装する際には，演算数やタスク数を増やしても，タスクどうしの依存性が低くクリティカル・パスが短いアルゴリズムに

書き換えるほうが効率的であると考えられる．

5. おわりに

大規模かつ階層的な次世代アーキテクチャを効率的に利用するためのプログラミングモデルとして提案された，ワークフローと分散並列/スレッド並列を組み合わせて利用するマルチ SPMD (mSPMD) プログラミングモデルの上で，3 種類の線形ソルバを実装し，性能を比較・評価した．実験の結果，いずれのアルゴリズムにおいても，ワークフローによるタスク並列と SPMD による分散並列を適切に組み合わせることで，より効率的にソルバを実行できることがわかった．また，演算数は少ないがタスク並列度が低いブロック・ガウス消去法，ブロック LU 分解よりも，演算数は多いがタスク並列度が高いブロック・ガウス・ジョルダン法のほうがしばしば高速だった．これらの結果から，mSPMD プログラミング・モデル上でアルゴリズムを実装する際には，ワークフローレベルでタスクどうしの依存性が低くクリティカル・パスが短いアルゴリズムを記述することが重要であると考えられる．

謝辞

本研究成果（の一部）は、理化学研究所のスーパーコンピュータ「京」を利用して得られたものです．

参考文献

[1] O. Delannoy, N. Emad, and S. Petiton. Workflow global computing with yml. In *The 7th IEEE/ACM International Conference on Grid Computing*, pp. 25–32, 2006.
[2] O. Delannoy and S. Petiton. A peer to peer computing framework: Design and performance evaluation of yml. In *Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pp. 362–369, 2004.
[3] J. Lee and M. Sato. Implementation and performance evaluation of XcalableMP: A parallel programming language for distributed memory systems. In *39th Annual International Conference on Parallel Processing*, pp. 413–420, 2010.
[4] M. Tsuji, M. Sato, M. Hugues, and S. Petiton. Multiple-SPMD programming environment based on PGAS and workflow toward post-petascale computing. In *Proceedings of the 2013 International Conference on Parallel Processing (ICPP-2013)*, pp. 480–485. IEEE, 2013.
[5] 辻美和子, 佐藤三久. 大規模システムにおける耐故障マルチ SPMD プログラミング開発実行環境の応用と評価. 情報処理学会研究報告, No. 2017-HPC-158, p. On Line. 情報処理学会, 2017.
[6] 辻美和子, 佐藤三久, M. Hugues, S. Petiton. 並列コンポーネントを統合する階層型並列プログラミングモデル. 情報処理学会研究報告, No. 2012-HPC-135, pp. 1–10. 情報処理学会, 2012.

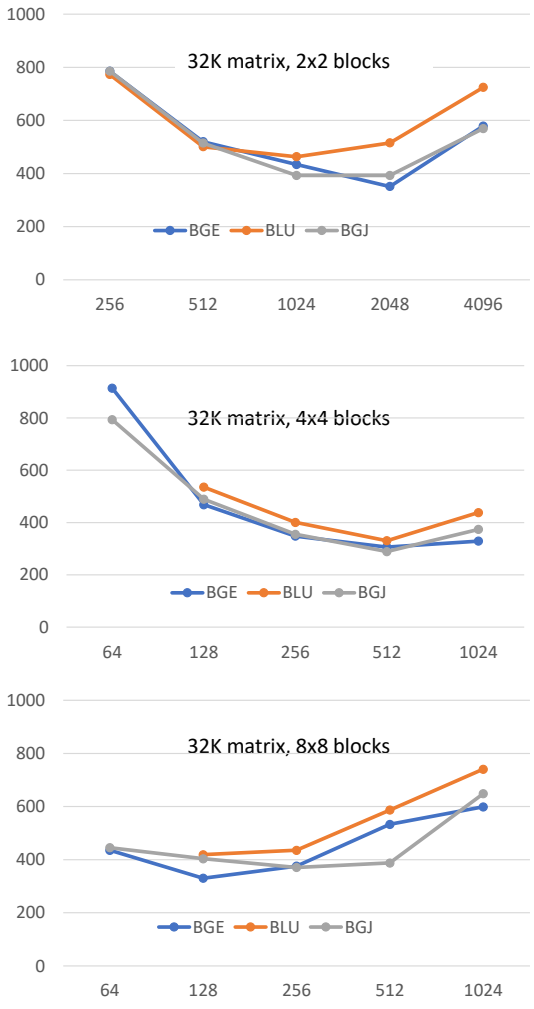

図 21 32k 行列・2 × 2, 4 × 4, 8 × 8 分割のときの，タスク毎のプロセス数に対する各アルゴリズムの実行時間の比較

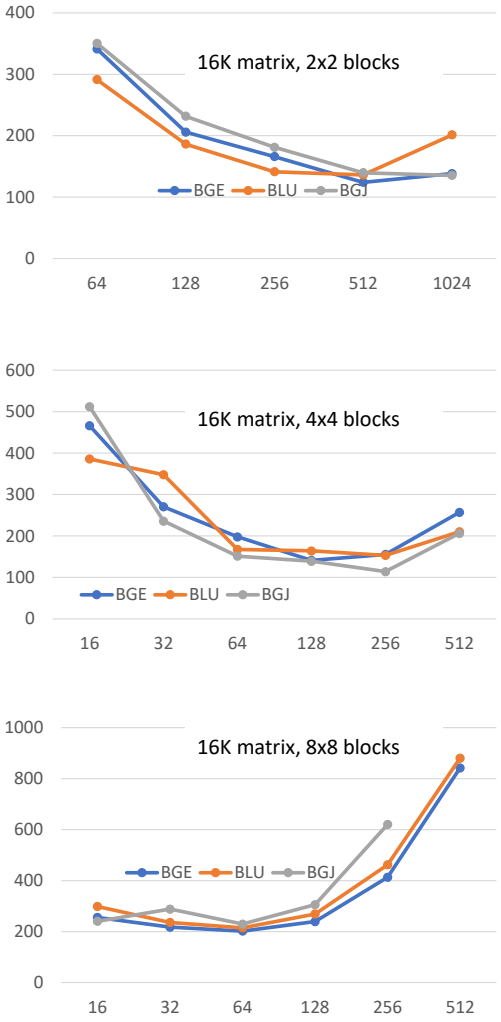

図 22 16k 行列・2 × 2, 4 × 4, 8 × 8 分割のときの，タスク毎のプロセス数に対する各アルゴリズムの実行時間の比較