Vol.2010-HPC-126 No.7
2010/8/3

XcalableMPによるNAS Parallel Benchmarksの実装と評価

中尾昌広[†1] 李珍泌[†2]
朴泰祐[†1,†2] 佐藤三久[†1,†2]

XcalableMP は C や Fortran 言語といった既存言語の並列拡張であり，分散メモリ環境で動作する並列アプリケーションを簡易に作成することができる新しいプログラミングモデルとして提案されている．XcalableMP で記述された並列アプリケーションの評価を行うため，NAS Parallel Benchmarks の Embarrassingly Parallel（EP），Integer Sort（IS）および Conjugate Gradient （CG）の実装を XcalableMP で行った．XcalableMP で実装した EP，IS および CG と既存の MPI を用いたそれらとの性能比較を行った結果，XcalableMP は MPI とほぼ同等の性能を示した．特に IS はヒストグラムを用いた手法，CG は二次元分割を行った手法の性能が高いことがわかった．このことから，XcalableMP は少ないプログラミングコストで分散メモリ環境上で効率よく動作するプログラムを作成できることがわかった．

Implementation and Evaluation of NAS Parallel Benchmarks in XcalableMP

Masahiro Nakao,[†1] Jinpil Lee,[†2] Taisuke Boku [†1,†2]
and Mitsuhisa Sato[†1,†2]

XcalableMP is an parallel extension of existing languages such as C and Fortran, which is proposed as a new programming model that can program parallel applications easily for distributed memory systems. In order to investigate the performance of parallel program written in XcalableMP, we have implemented some of NAS Parallel Benchmarks, an Embarrassingly Parallel (EP), Integer Sort (IS) and Conjugate Gradient (CG) by using XcalableMP. The performance results show that the performances of the XcalableMP programs are comparable to that of original MPI programs. In particular, the performance results of IS with histogram and CG with two-dimension-devide are high. It also shows that XcalableMP allows the user to write efficient parallel applications with lower programming cost.

1. はじめに

PC クラスタなどに代表される分散メモリ環境の並列プログラミングモデルとして Message Passing Interface（MPI）が広く用いられている．しかし，逐次プログラムをもとにして MPI を用いたプログラムを作成する場合，各プロセスにデータを分散して配置する等といった分散メモリ環境特有の処理を考慮する必要があるため，そのプログラミングコストの大きさが問題となっている．

そのような背景から，分散メモリ環境を対象とした新しい並列プログラミングモデルとして XcalableMP[1),2)] が提案されている．XcalableMP は，OpenMP のようにコメントによる指示文を逐次プログラムに加えることにより，データの分散配置やデータの受け渡しを指定する．XcalableMP を用いることで，既存の逐次プログラムと互換性を保ちつつ，並列プログラムを簡易に作成することが可能である．XcalableMP は C と Fortran 言語を対象に開発されており，2009 年 11 月にドラフト 0.7 がリリースされ，2010 年度中に 1.0 がリリースされる予定である．

本稿では，C 言語版の XcalableMP を用いて，代表的な並列ベンチマークである NAS Parallel Benchmarks（NPB）[3)] の Embarrassingly Parallel（EP），Integer Sort（IS）および Conjugate Gradient （CG）の実装および性能測定を行うことにより，XcalableMP の有用性と潜在能力を示すことを目的とする．

2. XcalableMP のプログラミングモデル

2.1 概　　要

XcalableMP の実行モデルは，MPI と同様に SPMD（Single Program Multiple Data）である．XcalableMP では，処理の実行単位（MPI におけるプロセス）をノードと定義する．XcalableMP ではノード毎に処理を行うため，すべてのデータはローカルデータとして処理を行う．あるデータを任意のノードに通信，集約や同期などをさせたい場合，ユーザが指示文を用いて明示的に指定する．

†1 筑波大学 計算科学研究センター
Center for Computational Sciences, University of Tsukuba
†2 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

XcalableMP はグローバルビューとローカルビューという 2 つのプログラミングモデルを提供しており，それぞれを順に適用させることで少ない負担で効率的な並列化を行うことができる．以下，それぞれについて述べる．

2.2 グローバルビューモデル

XcalableMP では，ノード間に跨る配列データを宣言し，その配列データに対する処理を各ノードで分担させて行うことを基本としている．これをグローバルビューによるプログラミングと呼ぶ．グローバルビューでは典型的な並列化のための通信をサポートしており，その手順を簡潔に記述することができる．ノードに跨がる配列を操作する場合，テンプレートと呼ばれる仮想的な配列を用いることで，逐次プログラムに特別な変更を行うことなく，配列を操作することができる．テンプレートの概念図を**図 1** に示す．テンプレートを用いることで，ユーザは配列データが複数のノードに分散されていることを意識せずに，プログラミングを行うことができる．

C 言語によるグローバルビューのプログラム例を**図 2** に示す．図 2 の 3 行目において計算に用いるノード集合（今回は 4 ノード）を定義している．4 行目では配列に用いるインデックスの下限値（0）と上限値（N-1）を設定したテンプレートを宣言している．5 行目ではノード集合に対するテンプレート割り当て方法を設定し，6 行目では配列とテンプレートとの関連付けを行っている．テンプレートの割り当て方法には，ブロック分割の block，サイクリック分割の cyclic，任意の分割を指定する gblock がある．12〜15 行目のループ文は，11 行目の指示文により各プロセスが独立に処理する．具体的には，ノード 1 は i=0 から N/4 -1 を，ノード 2 では i=N/4 から N/2 - 1 を処理する．17 行目では reduction 指示文により，各ノードで行った処理結果を集約している．リダクションには，定型的な演算子（+や MAX など）などが提供されている．

XcalableMP では，そのほかにも様々な定型的な通信手法が提供されている．その一部を下記に示す．

- #pragma xmp bcast *var* [, *var*...] [from *nodes* [on *nodes* | *template*]
  指定されたノードからのデータのブロードキャストを行う
- #pragma xmp barrier [on *nodes* | *template*]
  各プロセスでバリア同期を行う
- #pragma xmp gmove
  直後に記述される代入文がローカル領域ではなく，データが割り当てられたノードを参照するように通信を生成し，代入を行う

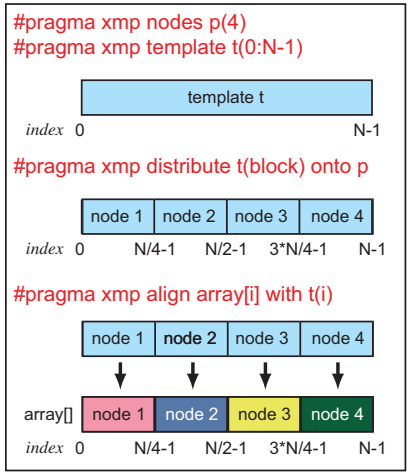

図 1 テンプレートの概念図
Fig. 1 Conceptual Diagram of "Template"

```
1  int array[N];
2
3  #pragma xmp nodes p(4)
4  #pragma xmp template t(0:N-1)
5  #pragma xmp distribute t(block) onto p
6  #pragma xmp align array[i] with t(i)
7
8  main(void){
9    int i, res = 0;
10
11 #pragma xmp loop on t(i)
12   for(i = 0; i < N; i++){
13     array[i] = func(i);
14     res += array[i];
15     }
16
17 #pragma xmp reduction (+:res)
18 }
```

図 2 グローバルビューのプログラム例
Fig. 2 Example of "Global View" Program

さらに，XcalableMP では，配列の各要素に対する計算が隣の要素に依存するパターンに対応するため，High Performance Fortran[4] で実装されている shadow を利用することができる．shadow で宣言されたデータ領域は reflect 指示文により，そのデータ領域をすべてのノードと同期することができる．

2.3 ローカルビューモデル

ローカルビューでは，各ノードのローカルメモリとノード間通信を強く意識したプログラミングを行うことで性能向上を狙う．ローカルビューではノード番号を指定することによって， そのノードに配置されたローカルデータを参照することができる．このようなプログラミングをローカルビューによるプログラミングと呼ぶ．

XcalableMP ではノード間のメッセージ通信を行うため，CAF[5] の仕様である Co-array 機能を提供している．C 言語による Co-array の例を**図 3** 示す．XcalableMP では Co-array 配列へのアクセスのために，配列の後の":"記号の後にノード番号を記述する．また，利便性向上のために，C 言語であっても，Fortran 言語で用いられている配列のように，配列内に":"記号を用いることで，参照範囲を指定することができる．図 3 の例では，配列 tmp の c から d までの要素を，ノード番号 1 の配列 array の a から b までの要素に代入することを示している．

```
int array[N];
# pragma xmp coarray array[*]
    ・・・
array[a:b]:[1] = tmp[c:d];
```

図 3 ローカルビューのプログラム例
Fig. 3 Example of "Local View" Program

```
#pragma xmp nodes p(NUM_PROCS)
#pragma xmp template t(1:NN)
#pragma xmp distribute t(block) onto p
    ・・・
#pragma xmp loop on t(k)
  for (k = 1; k <= NN; k++) {
    ・・・// 乱数発生
}
```

図 4 XcalableMP 版 EP
Fig. 4 XcalableMP Version EP

3. 実　　装

EP，IS および CG の XcalableMP を用いた実装方法について述べる．なお，XcalableMP は実装中であるため，後述する Co-array などのいくつかの機能は実現できていない．今回は，手動で通信関数の挿入を行い，実装を行った．

3.1 Embarrassingly Parallel ベンチマークの実装

EP は擬似乱数を生成するアルゴリズムである．並列化の方法としては，乱数発生を行う箇所を各プロセスに分散して実行するのみで良いため，プログラムの実行には通信をほとんど必要としない．そのため，並列計算に非常に適しているアプリケーションといえる．

EP の XcalableMP 実装を**図 4** に示す．乱数を発生させる回数を設定したテンプレートを用意し，乱数発生命令があるループ文を分散処理するようにすれば良いのみである．

3.2 Integer Sort ベンチマークの実装

まず，IS で用いられている整数ソート手法である並列化バケツソートの概要について述べる．IS ではヒストグラムを用いてバケツソートを行う手法と，ヒストグラムを用いない手法との 2 通りが提供されている．MPI 版の IS（NPB 3.3）では，ヒストグラムを用いた手法のみが提供されている．そのため，ヒストグラムを用いない手法については，Omni Compiler Project が提供している OpenMP 版の IS[6] を参考にして説明する．

3.2.1 既存のヒストグラムを用いる手法

各プロセスに均等に振り分けられたキーを，ランクの値が小さいプロセスには小さい値を持つキーを，ランクの値が大きいプロセスには大きい値を持つキーを振り分けるように，各プロセスが交換し合うことで，全体のソートを行うアルゴリズムである．

(1) プロセス毎に，初期化された TOTAL_KEYS/NUM_PROCS 個のキーを走査し，0〜MAX_KEY-1 の各値を持つキーの個数をカウントし，ヒストグラムを作成する．ここで，TOTAL_KEYS はキーの総数，NUM_PROCS はプロセス数，MAX_KEYS-1 はキーの値の最大値である．

(2) プロセス毎に，(1) で作成されたヒストグラムの各キーの個数を累計する．

(3) プロセス毎に，キーのソートを行う．

(4) 各プロセスのヒストグラムの各キーの個数の合計を求めることで，全体のヒストグラムを作成する（MPI_Allreduce を利用）．

(5) 全体のヒストグラムを参考にして，自プロセスが持つどのキーをどのプロセスに割り当てるか計算する．

(6) キーの交換を行う（MPI_Alltoall と MPI_Alltoallv を利用）．

(7) プロセス毎に，再度キーのソートを行う．

3.2.2 既存のヒストグラムを用いない手法

各プロセスに均等に振り分けられたキーの値を各プロセスが独立にカウントし，カウントした値の総計を求めることで，全体のソートを行うアルゴリズムである．

(1) スレッド毎に，初期化された TOTAL_KEYS/NUM_PROCS 個のキーを走査し，0〜MAX_KEY-1 の各値を持つキーの個数をカウントする．

(2) (1) でカウントしたキーの個数の累計を計算する（critical 構文を利用）．

3.2.3 ヒストグラムを用いる手法の XcalableMP 化

図 5 と下記に XcalableMP によるヒストグラムを用いた IS の要点を示す．

- テンプレートの宣言
  各ノードでキーの交換を行う際（3.2.1 節の (6)），各ノードに割り当てられるキーの個数が TOTAL_KEYS/NUM_PROCS よりも大きくなる場合がある．そのため，テンプレートの設定ではテンプレートの最大値に TOTAL_KEYS よりも大きな定数である SIZE_OF_BUFFERS を用いる．これは，MPI 版の IS でも同様のことが行われている．また，各プロセスでは TOTAL_KEYS/NUM_PROCS 個のインデックスを処理するように設定するため，データ分割方式には gblock を用いる．
- ループ文の処理
  キーに対して処理を行うループ文については，それぞれに loop 指示文を追記する．
- 全体のヒストグラムの作成
  XcalableMP が提供しているグローバルビューモデルの reduction 指示文を用いることで MPI_Allreduce と同様の処理が行える．

```
#pragma xmp nodes p(NUM_PROCS)
#pragma xmp template t(0:SIZE_OF_BUFFERS-1)
#pragma xmp distribute t(gblock(m)) onto p
#pragma xmp align key_array[i] with t(i)
・・・
#pragma xmp loop on t(i)                    // ヒストグラムの作成
   for( i=0; i<NUM_KEYS; i++ )
      bucket_size[key_array[i] >> shift]++;
・・・
#pragma xmp loop on t(i)                    // ソート
   for( i=0; i<NUM_KEYS; i++ )
   {
      key = key_array[i];
      key_buff1[bucket_ptrs[key >> shift]++] = key;
   }
・・・
#pragma xmp reduction(+:bucket_size)
・・・
int send_count[NUM_PROCS][NUM_PROCS];
int recv_count[NUM_PROCS][NUM_PROCS];
#pragma xmp template t2(0:NUM_PROCS-1)
#pragma xmp distribute t2(block) onto p
#pragma xmp align send_count[i][*] with t2(i)
#pragma xmp align recv_count[*][i] with t2(i)
・・・
#pragma xmp gmove
   recv_count[:][:] = send_count[:][:];

#pragma xmp loop on t2(i)
for(i=0;i<NUM_PROCS;i++)
   for(j=0;j<NUM_PROCS;j++)
      recv_count2[j] = recv_count[j][i];
・・・
#pragma xmp gmove
tmp[:][:] = recv_displ[:][:];

#pragma xmp loop on t2(i)
for(i=0;i<NUM_PROCS;i++)
   for(j=0;j<NUM_PROCS;j++)
      recv_displ2[j] = tmp[j][i];
・・・
for(i=0;i<NUM_PROCS;i++)                    // Co-arrayを用いた通信
   key_buff2[recv_displ2[i]:recv_displ2[i]+send_count[i]:[i] =
         key_buff1[send_displ[i]:send_displ[i]+send_count[i]];
```

図 5 ヒストグラムを用いる手法の XcalableMP 版 IS
Fig. 5 XcalableMP Version IS with Histogram

- キーの交換 1：(3.2.1 節の (6) の MPI_Alltoall に相当)
  MPI_Alltoall 通信を実現するために，2 次元配列を 2 つ用意し，それぞれ異なる次元をテンプレートを用いて分割する．その 2 次元配列を gmove 指示文を用いてコピーを行うことで，MPI_Alltoall 通信と同様の操作を実現することができる．補足説明を**図 6** に示す．図 6 では，たとえばノード k は配列 A の k 行目のデータのみを持っており，gmove を行うことでノード k の A[k][m] のデータはノード m の B[k][m] にコピーされる．
- キーの交換 2：(3.2.1 節の (6) の MPI_Alltoallv に相当)
  各ノード固有のデータを柔軟に操作する必要があるため，Co-array を用いたローカルビューによるプログラミングを行った．図 5 では，下記に示す MPI_Alltoallv を XcalableMP で実装を行っている．

```
MPI_Alltoallv(send_buf, send_count, send_displ, send_type, recv_buf,
                recv_count, recv_displ, recv_type, comm);
```

MPI_Alltoallv の引数の 1 つである recv_displ は「自プロセスがデータを受け取る位置」である．Co-array を用いて相手に通信を行う場合,「相手プロセスがデータを受け取る位置」を計算する必要があるため，まず，gmove を用いた Alltoall 通信により，各ノードの recv_displ から「相手プロセスがデータを受け取る位置」を計算する．そして，得た結果の配列をローカライズし，Co-array の代入文に用いている．

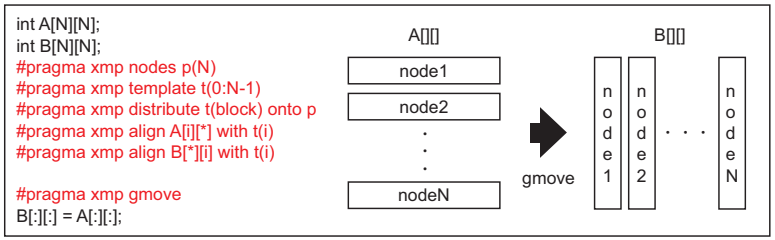

図 6 gmove を用いた Alltoall 通信
Fig. 6 Alltoall Communication by Using "gmove"

3.2.4 ヒストグラムを用いない手法の XcalableMP 化

図 7 と下記に XcalableMP によるヒストグラムを用いない IS の要点を示す．ヒストグラムを用いない手法の XcalableMP 化はグローバルビューのみで行うことが可能である．

- テンプレートの宣言
  本手法ではキーの交換は行わないため，テンプレートの最大値を TOTAL_KEYS に設定して宣言する．
- ループ文の処理
  ヒストグラムを用いる手法と同様に，キーに対して処理を行うループ文に loop 指示文を追記する．
- キーの個数の累計の計算
  各ノードがカウントした 0〜MAX_KEY-1 の各値を持つキーの個数を，reduction 指示文を用いて集約し，各キーの累計を計算する．

3.3 Conjugate Gradient ベンチマークの実装

CG は正値対称な大規模疎行列の最小固有値を共役勾配法を用いて解くプログラムである．CG の並列化手法として，プロセスを 1 次元に分割する方法と 2 次元に分割する方法があり，後者の方が性能が高いとされている[7]．今回は，両方の実装を行った．それぞれのコードの主要箇所を**図 8** と**図 9** に示す．

まず，図 8 と図 9 共に nodes 指示文で実行するノードの集合を宣言する．2 次元分割の場合，NPCOL × NPROW の値は MPI 版 CG と同様にノード数（NUM_PROCS）と同

```
#pragma xmp nodes p(NUM_PROCS)
#pragma xmp template t(0:TOTAL_KEYS-1)
#pragma xmp distribute t(block) onto p
#pragma xmp align key_array[i] with t(i)
・・・
#pragma xmp loop on t(i)
  for( i=0; i<NUM_KEYS; i++ ){
    key_buff2[i] = key_array[i];
    prv_buff1[key_buff2[i]]++;
  }
```

```
for( i=0; i<MAX_KEY-1; i++ )
      prv_buff1[i+1] += prv_buff1[i];

#pragma xmp reduction(+:prv_buff1)

for( i=0; i<MAX_KEY; i++ )
      key_buff1[i] += prv_buff1[i];
```

図 7 ヒストグラムを用いない手法の XcalableMP 版 IS
Fig. 7 XcalableMP Version IS without Histogram

```
#pragma xmp nodes p(NUM_PROCS)
#pragma xmp template t(0:na+1)
#pragma xmp distribute t(block) onto p
#pragma xmp align x[i] with t(i)
#pragma xmp align z[i] with t(i)
#pragma xmp align p[i] with t(i)
#pragma xmp align q[i] with t(i)
#pragma xmp align r[i] with t(i)
#pragma xmp align w[i] with t(i)
#pragma xmp shadow p[*]
#pragma xmp shadow z[*]
      ・・・
#pragma xmp loop on t(j)
for (j = 1; j <= lastcol-firstcol+1; j++) {
      ・・・      // ベクトルの計算
}
```

```
#pragma xmp reflect p

#pragma xmp loop on t(j)
for (j = 1; j <= lastrow-firstrow+1; j++) {
  sum = 0.0;
  for (k = rowstr[j]; k < rowstr[j+1]; k++) {
    sum = sum + a[k]*p[colidx[k]];
  }
  w[j] = sum;
}

#pragma xmp reduction(+:w)

#pragma xmp loop on t(j)
for (j = 1; j <= lastrow-firstrow+1; j++) {
  q[j] = w[j];
}
```

図 8 1 次元分割の XcalableMP 版 CG
Fig. 8 One-Dimension-Divide XcalableMP Version CG

じになるようにし，NPCOL の値は NPROW の値と同じか 2 倍になるようにする．次に，疎行列の計算のための配列を template 指示文を用いて宣言する．ここで，2 次元分割の場合，リダクション操作を短時間で行うために，配列 w の分割は他の配列とは異なる分割方法を宣言する．

2 重ループ文で用いられている配列 a は CRS 形式の一次元疎行列配列である．配列 a の参照を行うためには，配列 a の生成時につくられるインデックス配列 rowstr と colidx が必要である．2 重ループ文において，1 次元分割の場合，配列 p の値がすべてのノードにおいて必要であるため，shadow および reflect 指示文を用いて，すべての要素を参照前に同期する．また，図では省略しているが，配列 z についても同様の操作が必要である．2 次元分

```
#pragma xmp nodes p(NPCOL, NPROW)
#pragma xmp template t(0:na+1, 0:na+1)
#pragma xmp distribute t(block, block) onto p
#pragma xmp align x[i] with t(i, *)
#pragma xmp align z[i] with t(i, *)
#pragma xmp align p[i] with t(i, *)
#pragma xmp align q[i] with t(i, *)
#pragma xmp align r[i] with t(i, *)
#pragma xmp align w[i] with t(*, i)
      ・・・

#pragma xmp loop on t(j, *)
for (j = 1; j <= lastcol-firstcol+1; j++) {
      ・・・      // ベクトルの計算
}
```

```
#pragma xmp loop on t(*, j)
for (j = 1; j <= lastrow-firstrow+1; j++) {
  sum = 0.0;
  for (k = rowstr[j]; k < rowstr[j+1]; k++) {
    sum = sum + a[k]*p[colidx[k]];
  }
  w[j] = sum;
}

#pragma xmp reduction(+:w) on p(*, :)

#pragma xmp gmove
  q[:] = w[:];
```

図 9 2 次元分割の XcalableMP 版 CG
Fig. 9 Two-Dimension-Divide XcalableMP Version CG

割の場合，事前に配列 a のデータが割り当てられた template 空間に存在する場合，その情報をインデックス配列を保存しておく．この操作を行うことで shadow を用いることなく，2 重ループ文の計算を行うことが可能である．この操作は MPI 版と同様である．

2 重ループ文の計算結果である配列 w をリダクション操作を行った後に配列 q に保存する．2 次元分割の場合，配列 w と q の分割方法が異なるため，gmove 指示文を用いて保存を行う必要がある．

4. 性能評価

4.1 実験環境

本章では，3 章で実装した EP，IS および CG の性能を評価する．実験には**表 1** に示す一般的な PC クラスタおよび T2K-Tsukuba システムを最大 16 ノード用いて計測した．それぞれの各ノードは Quad Core CPU を持つが，ノード内並列化は行わずに各ノード 1 コアのみを用いる．また，EP，IS および CG の問題サイズは CLASS B とし，コンパイルオプションはすべて”-O3”とした．

4.2 結　　果

EP の結果を**図 10** に，IS の結果を**図 11** に，CG の結果を**図 12** に示す．それぞれ参考のため，既存の MPI 版（ただし EP と CG は Fortran 言語で実装されている）の結果も示す．

EP の場合，図 10 の結果より，MPI 版の方が性能が高いことがわかる．しかし，1 ノード実行時における性能比と 16 ノード実行時における性能比は同じであり，かつ EP は通信

表 1 実験環境のノード構成
Table 1 Specifications of Each Node on Experimental Environment

	PC Cluster	T2K Tsukuba System
CPU	Intel Core2 Quad CPU Q9650 3.00GHz	AMD Opteron Quad-Core 8000 series 2.3GHz
Memory	8GB	32GB
Network	Gigabit Ethernet	Infiniband DDR (4 rails)
OS	Linux 2.6.28	Linux 2.6.18
MPI	openmpi 1.3.3	MVAPICH2 1.4.1
C Compiler	gcc 4.1.2	gcc 4.1.2
Fortran Compiler	gcc 3.4.6	gcc 3.4.6

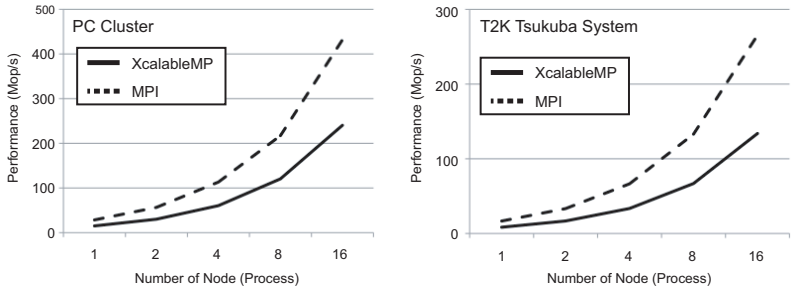

図 10 EP の結果
Fig. 10 Result of EP

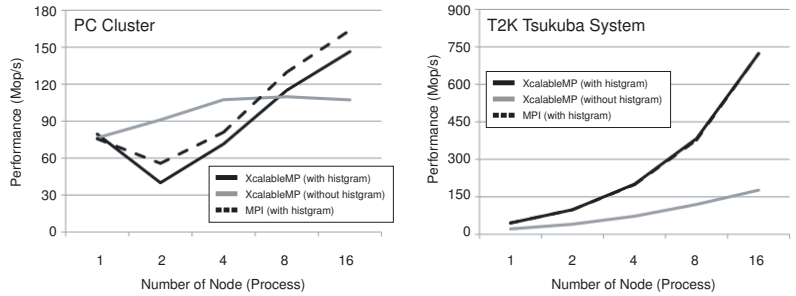

図 11 IS の結果
Fig. 11 Result of IS

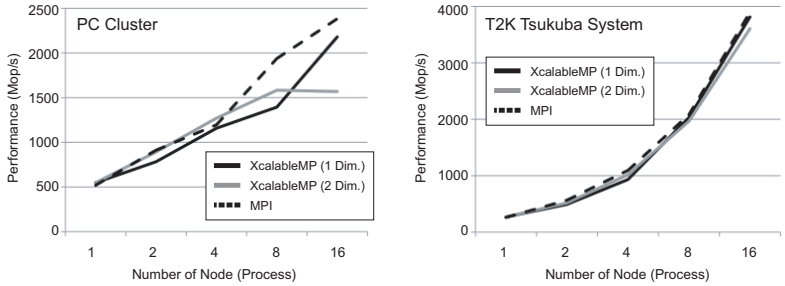

図 12 CG の結果
Fig. 12 Result of CG

はほとんど発生しない実装であるため，C 言語と Fortran 言語による違いが性能差の原因であると考えられる．

IS の場合，図 11 の結果より，ヒストグラムを用いた XcalableMP 版の IS と MPI 版の IS は似た傾向を示すことがわかる．特に，T2K Tsukuba システムの結果では，ほぼ同じ値を出力している．ヒストグラムを用いない IS では，PC クラスタにおいて 4 ノード以上を計算に用いた場合，性能が向上しないことがわかる．

CG の場合，図 12 の結果より，2 次元分割の場合は XcalableMP 版の CG と MPI 版の CG は似た傾向を示すことがわかる．しかし，PC クラスタを用いた場合，ノード数が 2 と 8 では MPI 版の方が性能が高い．また，1 次元分割の場合，8 ノードと 16 ノードの性能がほぼ等しくなっているため，これ以上にノード数を用いても性能は高くならないと予測できる．T2K Tsukuba システムを用いた場合は，すべての結果がほぼ同じ傾向を示す．

4.3 考　　察

IS において通信時間の中でも最も時間を要する箇所は，ヒストグラムを用いる手法の場合はキーの交換操作であり，ヒストグラムを用いない手法の場合はキーの個数を格納した配列のリダクション操作である．そこで各手法において通信に時間を要する時間を測定した．その結果を**表 2** に示す．表 2 より，PC クラスタにおいてヒストグラムを用いる手法の場合，MPI 版の方が XcalableMP 版よりも常に通信時間が少ないことがわかる．キーの交換操作には MPI 版は MPI_Alltoallv を用いた集合通信を用いているのに対し，XcalableMP 版は Co-array を用いた片側通信を用いている．そのため，XcalableMP 版の方が時間を要したのだと考えられる．また，ヒストグラムを用いない IS の場合，計算に用いるノード数

表 2 IS の通信に要する時間（秒）
Table 2 Communication Time (sec.) of IS

Process	PC Cluster			T2K Tsukuba System		
	With Histogram		Without Histogram	With Histogram		Without Histogram
	XcalableMP	MPI	XcalableMP	XcalableMP	MPI	XcalableMP
2	6.62	4.26	1.44	0.47	0.49	0.21
4	3.78	3.24	1.90	0.22	0.24	0.25
8	2.44	2.13	2.34	0.15	0.17	0.27
16	1.98	1.77	2.67	0.11	0.11	0.29

表 3 CG の通信に要する時間（秒）
Table 3 Communication Time (sec.) of CG

Process	PC Cluster			T2K Tsukuba System		
	two-dimension		one-dimension	two-dimension		one-dimension
	XcalableMP	MPI	XcalableMP	XcalableMP	MPI	XcalableMP
2	19.53	8.77	9.20	2.77	0.83	0.82
4	21.83	19.68	16.68	2.02	1.14	1.01
8	26.84	15.76	21.33	2.05	1.16	1.16
16	18.72	16.47	28.09	1.53	1.06	1.60

だけキーの個数を格納した配列のリダクション操作を行う必要があるため，ノード数に比例して通信量が増えることがわかる．

CG においても，IS と同様のことを検証するために，各手法において通信に要する時間を測定した．その結果を**表 3** に示す．ノード数が 2 と 8 の場合に XcalableMP 版の性能が MPI に及ばない理由は，XcalableMP では割り当てられたすべての要素に対してデータ転置（gmove）が行われるのに対し，MPI では該当要素のみが転置されるからである．ノード数が 4 と 16 の場合は，XcalableMP と MPI の通信コストはほぼ等しくなるため，その性能は接近していることが図 12 と表 3 よりわかる．PC クラスタを用いた場合，一次元分割において 8 ノード以上では性能が高くならない理由は，一次元分割では全ノードでリダクション操作が必要であるため，ノード数に比例して通信時間が必要になるからである．このことも表 3 から確認ができる．

PC クラスタにおいて XcalableMP 版の IS および CG を実行した場合，その性能は MPI 版に及ばないものの，プログラミングコストを考えると妥当なものである．また，高速ネットワークを搭載した T2K Tsukuba システムで実行した場合，実装は容易であるが PC クラスタにおいては性能の低いヒストグラムを用いない IS および CG の一次元分割であっても，今回用いたノード数程度であれば MPI 版とほぼ同等の性能であることがわかった．

5. まとめと今後の課題

新しい分散メモリ環境用プログラミングモデルである XcalableMP で記述された並列アプリケーションの性能評価を行うため，NPB の EP，IS および CG の実装を XcalableMP を用いて行った．それらと既存の MPI を用いた実装との比較を行った結果，XcalableMP は MPI と同等の性能を示すことがわかった．特に性能評価では，IS はヒストグラムを用いた手法，CG は二次元分割を行った手法の性能が高いことがわかった．

今後は，他の NPB のベンチマークの実装，またユーザに対して通信の最適化を行うための判断材料を提供するために，XcalableMP 用のプロファイリングツールの検討および実装を行う予定である．

謝辞 本研究の一部は，文部科学省「e-サイエンス実現のためのシステム統合・連携ソフトウェアの研究開発，高生産・高性能計算機環境実現のための研究開発，シームレス高生産・高性能プログラミング環境」による．また XcalableMP の仕様は，次世代並列プログラミング言語検討委員会によるものである．

参 考 文 献

1) XcalableMP specification Working Group: XcalableMP, http://www.xcalablemp.org/.
2) 李 珍泌，朴 泰祐，佐藤三久：分散メモリ向け並列言語 XcalableMP コンパイラの実装と性能評価，情報処理学会研究報告コンピューティングシステム第 31 号（採録決定）(2010).
3) Bailey, D.H. and et al.: THE NAS PARALLEL BENCHMARKS, Technical Report NAS-94-007, Nasa Ames Research Center (1994).
4) JAHPF (Japan Association for HPF) : High Performance Fortran 言語仕様書 Version 2.0，http://www.hpfpc.org/jahpf/spec/hpf-v20-j10.pdf.
5) Robert W. Numrich and John Reid: Co-array Fortran for parallel programming, ACM SIGPLAN Fortran Forum Volume 17 Issue 2 (1998).
6) Omni Compiler Project: NPB2.3-omni-C, http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/.
7) 李 珍泌，朴 泰祐，佐藤三久：分散メモリ向け並列言語 XcalableMP コンパイラの試作と評価，情報処理学会研究報告，2009-HPC-121, No.6 (2009).