Vol.2015-HPC-148 No.21
2015/3/3

PGAS言語XcalableMPを用いたHPC Challengeベンチマークの実装と評価

中尾 昌広[1,2,a)] 村井 均[1] 岩下 英俊[1] 下坂 健則[1] 朴 泰祐[2,3] 佐藤 三久[1,2,3]

概要：本稿では，XcalableMP の生産性と性能を明らかにするため，HPC Challenge ベンチマークの中の 4 つのベンチマーク STREAM，RandomAccess，Fast Fourier Transform と High Performance Linpack を XMP で実装した．その結果，XMP で実装した各ベンチマークは，MPI と比較して簡易に実装でき，さらに MPI による実装とほぼ同じ性能を達成することができた．また，データ並列を行うための高生産性 HPC 言語の要件についてまとめ，それらの要件に対する XcalableMP のデザインについて紹介する．

1. はじめに

分散メモリ環境における並列アプリケーションの作成には，規模の大小を問わず Message Passing Interface (MPI) が広く用いられている．しかしながら，近年，MPI よりも効率よく並列アプリケーションを作成するためのプログラミングモデルとして Partitioned Global Address Space（PGAS）プログラミングモデルが提案されている．PGAS プログラミングモデルを採用した言語としては，Coarray Fortran (CAF) [1], PCJ [2], Unified Parallel C (UPC) [3], UPC++ [4]，HabaneroUPC++ [5]，X10 [6]，Chapel [7] などがある．アプリケーションの一般的な並列化の種類としてはデータ並列とタスク並列があり，CAF，PCJ，UPC，UPC++はデータ並列の機能を提供しており，HabaneroUPC++，X10，Chapel はデータ並列とタスク並列の両方の機能を提供している．なお，HabaneroUPC++は UPC++に対してタスク並列の機能を追加したものである．

PGAS プログラミングモデルでは，並列に実行されるプロセスなどの実行主体同士が，透過的にアクセス可能な領域（大域アドレス空間）を介してデータの read/write を行う．さらに，その大域アドレス領域は論理的に区切られており，区切られた領域は各実行主体と対応しているため，プログラマは通信の発生する箇所を意識したプログラミングを行うことが可能である．すなわち，プログラマはデータの局所性を意識しつつ，共有メモリを用いるように実行主体間でデータ通信を行うことが可能になるため，PGAS プログラミングモデルは性能と生産性のバランスを考慮したモデルであると言える．

我々は，データ並列のための PGAS 言語 XcalableMP（XMP）[8–11] およびその処理系である Omni XMP Compiler [12] を開発している．XMP は，HPC 分野でよく用いられている C 言語（C99）および Fortran（Fortran95）に対して，データ並列のための指示文および拡張構文を提供している．また，Omni XMP Compiler では通信ライブラリに MPI と GASNet [13] を用いており，Linux クラスタおよび様々な計算システム（e.g. スーパーコンピュータ「京」[14]，富士通 FX10，NEC SX-9, 日立 SR16000, Cray XE6，IBM BlueGene/Q など）で動作が確認されている．特に「京」においては，GASNet の代わりに「京」が提供している拡張 RDMA インタフェースを利用しているため，より高速な通信を行うことができる．

計算システムの性能を多角的に評価するためのベンチマークセットとして，HPC Challenge（HPCC）ベンチマーク [15, 16] がある．並列言語のコンテストである HPCC Awards Competition Class 2 [17] では，HPCC ベンチマークの主要な 4 つのベンチマーク STREAM，RandomAccess，Fast Fourier Transform（FFT），High Performance Linpack（HPL）が課題として与えられており，それらを並列言語を用いて実装することで，その並列言語の生産性と性能が評価される．なお，HPCC Awards Competition には Class 1 も存在する．Class 1 は，言語を問わずに計算システムの性能のみを競う部門である．

本稿では，XMP の生産性と性能を評価するため，上記で

1 理化学研究所 計算科学研究機構
RIKEN Advanced Institute for Computational Science
2 筑波大学 計算科学研究センター
Center for Computational Sciences, University of Tsukuba
3 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
a) masahiro.nakao@riken.jp

ⓒ 2015 Information Processing Society of Japan

述べた4つのベンチマークの実装を「京」の上で行う．過去の研究 [9,18] において，XMP を用いた HPCC ベンチマークの実装は行われているが，その性能は HPCC Awards Competition Class 1 に提出された HPCC ベンチマークの性能 [19] と比較して低いものだった．そこで本稿では，より高い性能を達成するために行った各ベンチマークに対するチューニングについても述べる．本稿の内容は，2014 年 HPCC Awards Competition Class 2 に投稿した内容 [20] である．また，本稿では，データ並列のための高生産性 HPC 言語の要件について述べ，それらに対する XMP のデザインについても述べる．

本稿の構成は下記の通りである．2 章ではデータ並列のための高生産性 HPC 言語の要件について述べる．3 章では XMP の概要について説明し，2 章で述べた要件に対する XMP のデザインについて述べる．4 章では HPCC ベンチマークの実装および性能評価を行う．5 章では考察を行い，6 章ではまとめと今後の課題について述べる．

2. データ並列のための高生産性 HPC 言語の要件

高生産性 HPC 言語では，アプリケーションの開発・実行・性能チューニング・ポーティング・完成したコードの拡張や再利用などに要する総時間を小さくすることが重要である．そのため，高生産性 HPC 言語の生産性と性能は，高いレベルで両立していることが求められる．具体的には，現在広く用いられている MPI+C/Fortran 言語と比較して，性能を維持しつつ生産性を高くすることが望ましい．

本章では，データ並列を行う上で，高い生産性と性能を実現するために HPC 言語に求められる要件について述べる．

2.1 通信の簡易化

MPI によるプログラミングが困難な理由の 1 つに，デッドロックが発生しないように実行のフローを常に意識しつつ，データの送受信を行う必要がある点が挙げられる．そのため，デッドロックが発生しない，より簡易な通信手段を提供する必要がある．また，様々なアプリケーション開発に対して高い生産性と性能を達成するため，片側通信，集合通信や P2P バリアといった機能も必要である．

2.2 計算システムの抽象化

計算システムにおいて高い性能を発揮するには，計算システムを構成する計算ノード，CPU ソケット，CPU コア，アクセラレータなどの各コンポーネントとその階層構造を意識した局所性の高いプログラミングを行う必要がある．そのようなプログラミングを行うためには，各コンポーネントを抽象化してユーザに提供する必要がある．また，この抽象化により，高いポータビリティを達成することを期待できる．

2.3 既存コードおよび外部ライブラリとの相互運用性

既存の HPC アプリケーションを高生産性 HPC 言語を用いて書き直す場合，その高生産性 HPC 言語の生産性と性能を確認しながら段階的に書き直していくのが現実的であると考えられる．さらに，HPC アプリケーションの中で特に性能を重視する箇所は，既存コードをそのまま残すことも考えられる．これらのことから，既存コードとの相互運用性が重要になる．この相互運用性は，複数の数理モデルを用いたシミュレーション（e.g. Parallel Ocean Program [21] など）において，数理モデルによって言語を使い分ける場合も有用である．

さらに，各計算システムでは，そのハードウェアに最適化された BLAS などの数値計算ライブラリや MPI などの通信ライブラリが提供されている場合がある [22]．これらのライブラリとの相互運用性を持つことにより，高いポータビリティと性能を達成できる．

2.4 分散データの抽象度の調節

MPI や CAF を用いてアプリケーションの並列化を行う場合，実行主体に対するデータの割り当て，割り当てられたデータに対する処理，データ通信などを，実行主体毎に記述する．MPI や CAF のように，各実行主体が固有の名前空間を持つ並列化ビューをローカルビューと呼ぶ．ローカルビューの問題点として，逐次コードから並列コードを作成する際に，逐次コードを大きく書き換えることになるため，プログラミングコストは大きくなり，さらに可読性も低下する点が挙げられる．

ローカルビューとは逆に，各実行主体が名前空間を共有している並列化ビューをグローバルビューと呼ぶ．グローバルビューにおいては，データ分散は抽象化されており，プログラマはグローバルインデックスなどを用いてデータに対する処理を記述する．すなわち，ローカルビューでは個々の実行主体の処理を記述するのに対し，グローバルビューでは全体の処理を記述することが可能になるため，逐次コードのように並列コードを記述することが可能になる．

しかしながら，グローバルビューの抽象度は一般に高いため，ユーザの予期しない箇所で性能低下が起きる可能性がある．逆に，ローカルビューの抽象度は低く，実行主体毎の処理を記述するので，計算機の動作が予測しやすいプログラミングを行えるという利点がある（Performance-aware programming）．そのため，ローカルビューはグローバルビューと比較して，チューニングが行い易い場合がある．

以上のことから，ローカルビューとグローバルビューには，それぞれ利点と問題点が存在することがわかる．それぞれの機能は互いに直交しているため、アプリケーションに応じて並列化ビューを切り替えることで，分散データに対する抽象度を調節できることが望ましいと考える．

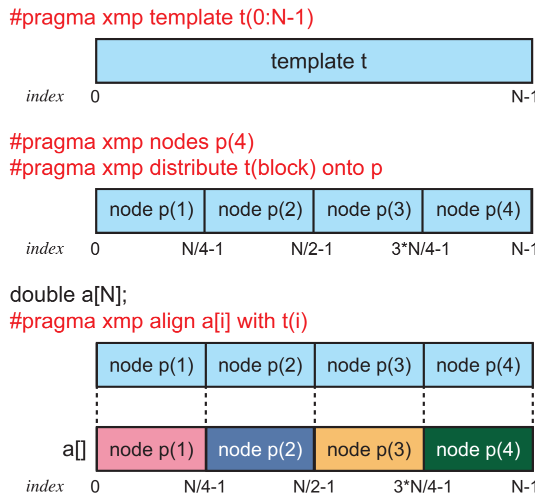


図 1　分散配列の定義とループ文の並列処理 [11]

```
#pragma xmp loop on t(i)
#pragma omp parallel for
for(int i=0;i<N;i++){
  a[i] = ...
```

図 2　ループ文の並列処理 [11]

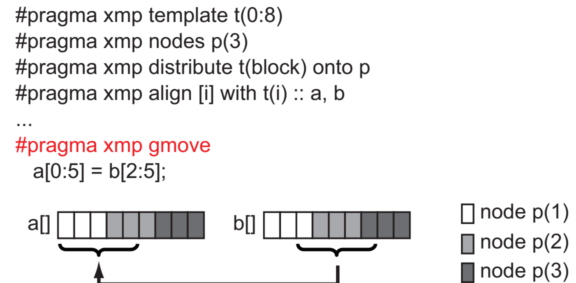


図 3　**gmove** 指示文の例 [11]

3. XcalableMP

本章では，まず XMP の概要について述べ，次に 2 章で述べた要件に対する XMP のデザインについて述べる．なお，XMP は C 言語と Fortran に対応しているが (本稿では XMP/C と XMP/Fortran と呼称する)，本章では XMP/C を用いて説明を行う．最後に XMP の処理系である Omni XMP Compiler について説明する．

3.1　概要

XMP は分散メモリシステムのための指示文ベースのプログラミングモデルである．XMP の仕様は PC クラスタコンソーシアム [23] によって策定されており，XMP の仕様の一部は，CAF と High Performance Fortran [24,25] がベースとなっている．なお，XMP/Fortran は CAF の上位互換である．

XMP の実行モデルは Single Program Multiple Data (SPMD) である．XMP の実行主体を “ノード” と呼び，全ノードで同じプログラムが実行される．ノードは OS レベルのプロセスと同義である．XMP 指示文は分散配列の定義，ループ文の分散，通信の実行などを行う．指示文で定義されていない変数は，全ノードで重複して持つ．

3.2　データ並列のための高生産性 HPC 言語のデザイン

3.2.1　分散配列の定義とループ文の並列処理

XMP では分散配列を定義するために，仮想インデックス集合である “テンプレート” を用いる．**図 1** と**図 2** に分散配列の定義とループ文の並列処理の例を示す（2.2 節，2.3 節，2.4 節に関係）．

(1) **template** 指示文はテンプレート t を定義する．図 1 の t のインデックスは 0〜N-1 である．
(2) **node** 指示文はノード集合 p を定義する．図 1 では，p は 4 ノードから構成されている．
(3) **distribute** 指示文はテンプレート t をノード集合 p に指定した分散方式で割り当てる．図 1 の場合はブロック分散であり，各ノードに同じ要素数を割り当てることを意味している．他の分散方式として，サイクリック分散，ブロックサイクリック分散，ユーザ定義分散がある．また，配列の次元毎に分散を設定できる．
(4) **align** 指示文は分散配列 *a[]* を，各ノードに割り当てたテンプレート t に整列させる．図 1 で N=16 であるならば，各ノードは分散配列 *a[]* の 4 要素ずつを持つ．
(5) **loop** 指示文はテンプレート t に従って，ループ文を分割する．図 1 と図 2 で N=16 であるならば，ノード *p(1)* はインデックスの 0〜3 を実行する．また，OpenMP 指示文と組合せることで，スレッド並列を同時に行うことも可能である．図 2 の場合，XMP 指示文と OpenMP 指示文の順序は問わない．

このように，XMP ではグローバルビューに対応しており，逐次コードに指示文を追加するのみで，ループ文の並列処理を行うことができる．また，XMP は既存言語の拡張であるので，図 2 のように OpenMP などの他のライブラリとの相互運用が可能である．さらに，XMP の実行主体である “ノード” を CPU ソケットと対応させることで，CPU ソケットと CPU コアを意識したプログラミングが可能になる．

3.2.2　集合通信

本節では，グローバルビューにおける集合通信のための指示文について説明する（2.1 節と 2.4 節に関係）．

- **gmove** 指示文は，分散配列に対して通信を発生させることができる．**図 3** のように **gmove** 指示文の直後に代入式を記述する．XMP/C において複数の要素の転送を表現するために，コロンの前は転送開始インデックス，コロンの後は転送要素数を記述する．この例では，分散配列 *b[2]*〜*b[6]* までの 5 要素を，分散配列 *a[0]*〜*a[4]* に転送している．

ⓒ 2015 Information Processing Society of Japan

- **bcast** 指示文は，指定されたノードが持つローカル変数に対してブロードキャスト通信を発生させる．下記の例では，ノード *p(2)* が持つローカル変数 *e* をブロードキャストしている．

```
1  #pragma xmp bcast (e) from p(2)
```

- **reduction** 指示文は，集約演算を行う．下記の例では，全ノードが持つローカル変数 *f* の総計を求めている．

```
1  #pragma xmp reduction (+:f)
```

3.2.3 分散配列に対する問合せ関数

既存コードおよび外部ライブラリを分散配列に適用するために，分散配列とローカルメモリとの対応情報を返す問合せ関数が提供されている（2.3 節に関係）．

下記に，分散配列の leading dimension を返す関数 xmp_array_lead_dim() の利用例を示す．

```
1  xmp_desc_t A_desc =
2               xmp_desc_of(A); // A[N][N] is distributed
3  int ld[2];
4  xmp_array_lead_dim(A_desc, ld);
```

1 行目と 2 行目は 2 次元分散配列 *A[][]* のディスクリプタを変数 *A_desc* に代入している．4 行目は *A[][]* の各次元の leading dimension を int 型配列 *ld[]* に代入している．他にも，グローバルインデックスからローカルインデックスを返す関数 xmp_array_gtol() や分散配列のローカル上の先頭アドレスを返す関数 xmp_array_laddr() などが用意されている．

XMP では，ある要素に対する分散配列のポインタは，その要素を持っているノードにおいては，その要素に対するローカルポインタを示す．また，Omni XMP Compiler については，分散配列とローカルに確保されるメモリ領域のレイアウトは定義されている [8] ので，分散配列とローカル領域との対応をユーザが計算することも可能である．

また，XMP では MPI との相互運用を行うために，MPI の初期化，終了処理を行う関数 xmp_init_mpi() と関数 xmp_finalize_mpi()，MPI のコミュニケータを取得する関数 xmp_get_mpi_comm() が提供されている．下記にそれらの関数の利用例を示す．このように，XMP のコード上から MPI をシームレスに利用できる．

```
1  #include <stdio.h>
2  #include 'mpi.h'
3  #include 'xmp.h'
4
5  #pragma xmp nodes p(4)
6
7  int main(int argc, char *argv[]) {
8    xmp_init_mpi(&argc, &argv);
9    MPI_Comm comm = xmp_get_mpi_comm();
10
11   int rank;
12   MPI_Comm_rank(comm, &rank);
13       :
14   xmp_finalize_mpi();
15   return 0;
16 }
```

3.2.4 XcalableMP/C における Coarray

本節では，XMP/C における CAF の記法をベースにした片側通信機能について説明する（2.1 節と 2.4 節に関係）．

下記に，CAF 記法の例を示す．

```
1  int a[N]:[*];
2  int b[N], status;
3   :
4  a[0:5]:[1] = b[2:5]; // Put
5  xmp_sync_memory(&status);
6  b[2:5] = a[0:5]:[1]; // Get
```

1 行目は配列 *a[]* を Coarray として宣言している．4 行目は Put 通信を，6 行目は Get 通信を表現している．XMP/C において Coarray を参照する際は，通常の配列参照の後ろにコロンと角括弧を用いてノードの番号を指定する．また，**gmove** 指示文と同様に，コロンを用いて参照領域を指定する．例えば，4 行目はノード番号 1 が持つ *a[0]*〜*a[4]* の要素に，自ノードの *b[2]*〜*a[6]* を転送している．5 行目の関数 xmp_sync_memory() は，それより前に発行された Coarray 通信の完了を保証する．

このように，XMP では Coarray によるローカルビューにも対応している．この機能により，ユーザはグローバルビューよりも柔軟なアルゴリズムを実装することができる．

3.2.5 post 指示文と wait 指示文

ノード間の処理の依存関係を記述するために，**post** 指示文と **wait** 指示文が提供されている（2.1 節に関係）．

下記に，**post** 指示文と **wait** 指示文の例を示す．

```
1  int tag = ...;
2  if(xmp_node_num() == 1){
3     :
4  #pragma xmp post(p(2), tag)
5  }
6  else if(xmp_node_num() == 2){
7  #pragma xmp wait(p(1), tag)
8   :
9  }
```

関数 xmp_node_num() はノード番号を返す関数である．2〜5 行目はノード番号 1 が処理を行い，6〜9 行目はノード番号 2 が処理を行う．上の例では，ノード番号 1 が **post** 指示文を実行するまで，ノード番号 2 は **wait** 指示文の後は実行しないことを意味している．

3.3 Omni XcalableMP Compiler

3.3.1 概要

Omni XMP Compiler はソース・プログラム変換を用いたコンパイラである．Omni XMP Compiler の動作を**図 4** に示す．まず，ベース言語（C 言語もしくは Fortran）と XMP 指示文で記述されたコードを，Omni XMP Compiler

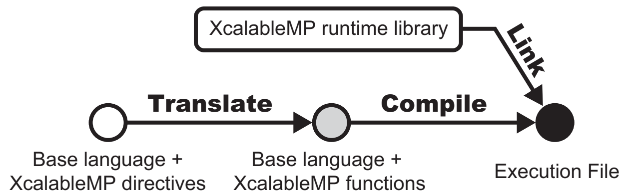


図 4 Omni XcalableMP Compiler の動作

表 1 「京」のスペック

CPU	SPARC64 VIIIfx 2.0 GHz, 8 Cores, 128 GFlops
Memory	DDR3 SDRAM 16 GB, 64 GB/s
Network	Torus fusion six-dimensional mesh/torus network, 5 GB/s x 10
Compier	Fujitsu C/Fortran Compiler K-1.2.0-15
Library	Fujitsu MPI K-1.2.0-15, Fujitsu SSLII K-1.2.0-15

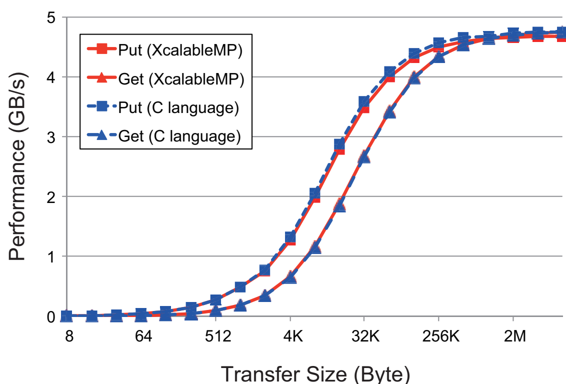


図 5 片側通信の性能比較

はベース言語と XMP ランタイムが用意している関数に変換する．次に，計算システムが用意しているコンパイラ（gcc，PGI や Intel コンパイラなど）を用いて，コンパイルと XMP ランタイムとのリンクを行い，実行ファイルを生成する．

XMP ランタイムの実装では，3.2.2 節で述べた集合通信には MPI を用いており，3.2.4 節と 3.2.5 節で述べた片側通信には GASNet を用いている．GASNet は MPI 実装も存在するため，Omni XMP Compiler は MPI が対応しているシステムであれば動作する．また，「京」においては，より高い性能を達成するため，GASNet の代わりに「京」が提供している RDMA 用の関数を呼び出すようにしている．

3.3.2 「京」における Coarray の性能評価

本節では，「京」における XMP/C の Coarray の性能評価を行う．「京」のスペックを**表 1** に示す．性能評価には，Ohio State University Micro-Benchmarks [26] の latency benchmark を元に作成した put/get 通信による ping-pong プログラムを用いる．XMP/C の Coarray を用いた場合と，C 言語から「京」の RDMA 用の関数を直接呼び出した場合とで比較する．

2 ノード間で ping-pong プログラムを実行させた結果を**図 5** に示す．この結果から，XMP/C の Coarray の性能は，C 言語から「京」の RDMA 用の関数を直接呼び出した場合とほぼ同じ性能であることがわかる．

```
1  #pragma xmp nodes p(*)
2  ...
3  #pragma loop xfill
4  #pragma loop noalias
5  #pragma omp parallel for
6  for (i=0; i<N; i++)
7      a[i] = b[i] + scalar*c[i];
8  ...
9  #pragma xmp reduction(+:triadGBs)
```

図 6 XcalableMP を用いた STREAM の一部

4. HPC Challenge ベンチマークの実装と性能評価

本章では，XMP を用いて HPCC ベンチマークの実装を行い，表 1 に示す「京」を用いて性能評価を行う．STREAM と FFT の性能評価には「京」の全計算ノードを用いており，RandomAccess と HPL の性能評価には「京」の 16,384 ノードを用いている．また，チューニングの効果を測定するため，過去の XMP を用いた実装の性能 [9, 18] と比較する．さらに，STREAM と FFT については，2012 年の HPCC Awards Competition Class 1 [19] における「京」の全計算ノードを用いた HPCC ベンチマークの結果とも比較する．このベンチマークの実装には MPI が用いられており，さらに「京」に対するチューニングも行われている．

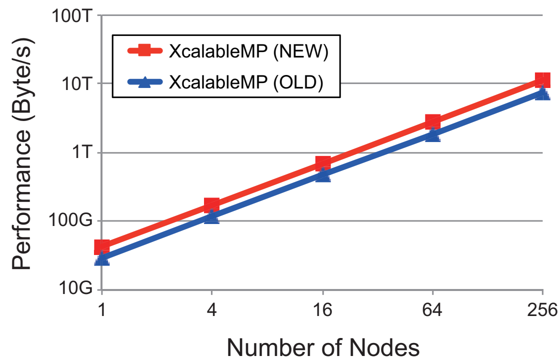
4.1 STREAM

4.1.1 実装

STREAM は実行メモリバンド幅を計測するベンチマークである．STREAM の並列化は非常に簡易であり，プログラマは逐次版 STREAM に指示文を追加するのみで並列化が可能である．

XMP/C を用いて並列化を行ったコードの一部を**図 6** に示す．1 行目は XMP のノード集合を定義しており，プログラムが並列に実行されることを意味している．3〜7 行目は STREAM のカーネルであり，for ループ文は 5 行目の OpenMP 指示文により各ノードでスレッド並列で実行される．9 行目は各ノードの性能結果を集計している．

過去の実装 [9, 18] との差分は，3 行目と 4 行目に「京」で提供されている富士通コンパイラの最適化指示文を追加したことである．“#pragma loop xfill” は書き込み用のキャッシュラインを確保することを指定しており，この指示文によりストア命令を高速化できる．“#pragma loop noalias” は異なるポインタ変数が同一の記憶領域を指さないことを指定しており，コンパイラが最適化を行い易くしている．



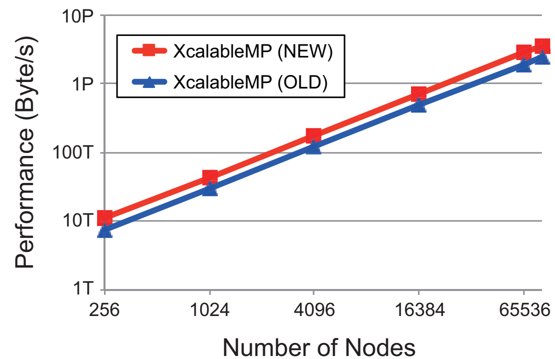


図 7 STREAM の性能結果

4.1.2 性能評価

各計算ノードにつき，1 プロセス 8 スレッドで計測を行った．double 型配列 *a[]*, *b[]*, *c[]* の各サイズ（図 6 の 6 行目の *N*）は 536,870,912 である．3 つの配列の合計はシステムメモリの 75%を占める．

結果を**図 7** に示す．図 7 において，今回の結果は “XcalableMP (NEW)”，過去の結果を “XcalableMP (OLD)” とした．82,944 ノード利用時の XcalableMP (NEW) の性能は 3.58 PB/s であり，XcalableMP (OLD) の性能は 2.44 PB/s である．この結果から，富士通コンパイラの最適化指示文を加えることで，性能が 1.47 倍に向上することがわかる．また，Class 1 の STREAM の性能は 3.86 PB/s であるため，XcalableMP (NEW) は，Class 1 とほぼ同じ性能を達成していると言える．

4.2 RandomAccess

4.2.1 実装

RandomAccess は，複数の計算ノードに分散された Table に存在する整数データに対して，各プロセスがランダムに更新を行う速度を計測するベンチマークである．1 対 1 通信が大量に発生するため，ネットワーク性能が測定結果に強く影響する．RandomAccess の計測単位は GUPS (Giga UPdates per Second) であり，1 秒間に Table の要素を更新した回数を 10^9 で割った値である．

我々の実装は，自ノードが持っている整数データに対して各送信先を計算し，他ノードに対してチャンク毎に転送・更新を行う．この転送パターンは全対全個別通信（all-to-all personalized communication）である．実装では通信回数を減らすため，Recursive Exchange Algorithm を用いている [27]．

XMP/C の Coarray と **post** 指示文と **wait** 指示文を用いて RandomAccess を実装した．**図 8** にコードの一部を示す．1〜2 行目は Coarray *recv[][]* と *send[][]* を宣言している．19 行目は相手に送る要素数（変数 *nsend*）を *send[][]* の先頭に追加している．20 行目は *send[isend][0]* から *send[isend][nsend]* の要素を，ノード番号 *ipartner+1* の *recv[j][0]* から *recv[j][nsend]* に put 通信で送っている．21 行目の関数 xmp_sync_memory() は，直前の put 通信の完了を保証する．22 行目と 28 行目の **post** 指示文と **wait** 指示文は，ノード番号 *ipartner+1* に自ノードの put 通信が完了したことを伝えるために使われている．**wait** 指示文の後は，他ノードからのデータが到着しているので，そのデータを使って自ノードが持っている Table の更新を行う．

```
1  u64Int recv[MAXLOGPROCS][RCHUNK+1]:[*];
2  u64Int send[2][CHUNKBIG+1]:[*];
3  ...
4  for (j = 0; j < logNumProcs; j++) {
5    nkeep = nsend = 0;
6    isend = j % 2;
7    ipartner = (1 << j) ^ MyProc;
8    if (ipartner > MyProc) {
9      sort_data(data, data, &send[isend][1], nkept, ...);
10     if (j > 0) {
11       jpartner = (1 << (j-1)) ^ MyProc;
12 #pragma xmp wait(p(jpartner+1))
13       xmp_sync_memory(&status);
14       nrecv = recv[j-1][0];
15       sort_data(&recv[j-1][1], data, &send[isend][1], nrecv,
             ...);
16     }
17   }
18   else { ... }
19   send[isend][0] = nsend;
20   recv[j][0:nsend+1]:[ipartner+1] = send[isend][0:nsend+1];
21   xmp_sync_memory(&status);
22 #pragma xmp post(p(ipartner+1), 0)
23   if (j == (logNumProcs - 1)) update_table(data, Table,
          nkeep, ...);
24   nkept = nkeep;
25 }
26 ...
27 jpartner = (1 << (logNumProcs-1)) ^ MyProc;
28 #pragma xmp wait(p(jpartner+1))
29 #pragma xmp sync_memory
30 nrecv = recv[logNumProcs-1][0];
31 update_table(&recv[logNumProcs-1][1], Table, nrecv, ...);
```

図 8 XcalableMP を用いた RandomAccess の一部

過去の実装 [9,18] では，Omni XMP Compiler の **post** 指示文と **wait** 指示文の実装に関数 MPI_Send() と関数

MPI_Recv() を用いていた．今回の実装では「京」が提供する RDMA を用いて，**post** 指示文と **wait** 指示文の実装を行った．

4.2.2 性能評価

各計算ノードにつき，8 プロセスで計測を行った．Table のサイズは，システムメモリの半分を使うように設定した．

結果を**図 9** に示す．図 9 において，今回の結果は “XcalableMP (NEW)”，過去の結果を “XcalableMP (OLD)” とした．また，MPI で実装されたリファレンス実装 [15,16] の RandomAccess に対して，Table の更新のためのローカル処理を「京」に最適化させた結果も示す．この結果は図 9 に “MPI-modified” と表示した．なお，このローカル処理の最適化は，両方の XMP の実装でも行っている．16,384 ノード利用時の XcalableMP (NEW) の性能は 254.20 GUPs であり，XcalableMP (OLD) の性能は 162.63 GPUs である．この結果から，**post** 指示文と **wait** 指示文に「京」の RDMA を用いて実装したことで，性能が 1.56 倍に向上することがわかる．また，“MPI-modified” の結果は 243.40 GUPs であることから，XcalableMP (NEW) は，MPI で実装されたリファレンス実装とほぼ同じ性能を達成していると言える．

4.3 Fast Fourier Transform

4.3.1 実装

FFT は 1 次元離散複素フーリエ変換に対する速度を計測するベンチマークであり，計算システムの演算性能と通信性能 (全対全通信) の両方が測定結果に影響する．我々はリファレンス実装と同様に，FFTE ライブラリ [28]（version 6.0）を用いて six-step FFT アルゴリズム [29,30] を実装した．six-step FFT アルゴリズムは，1 次元 FFT を 2 次元表現で計算することにより，キャッシュミスを少なくするとともに，データ通信に対して効率的な集団通信を利用可能であるという特徴を持つ．

FFT の実装には XMP/Fortran と Fortran を用いた．**図 10** にコードの一部を示す．図 10 の配列 a, b, c は，ブロック分割された分散配列である．サブルーチン zfft1d() は，FFTE ライブラリの 1 次元複素 FFT ルーチンであり，3.2.3 節で述べた分散配列の特徴により，分散配列をそのまま代入して用いる．six-step FFT アルゴリズムでは，データの転置のために全対全通信を 3 回実行する必要があるため，XMP が提供している分散配列の転置用のサブルーチンである “xmp_transpose()” を用いた．**図 11** に xmp_transpose() の動作の概要を示す．xmp_transpose() の第 1 引数（出力）と第 2 引数（入力）には，それぞれ 2 次元配列を指定する．第 3 引数はオプションであり，“1” を指定すると入力配列の値が変わる可能性がある代わりに省メモリで動作する場合がある．

また，XMP/Fortran と Fortran との相互運用性の高さ

```
1  complex*16 a(nx,ny), b(ny,nx), w(ny,nx)
2  complex*16 cx(*),cy(*)
3
4  !$xmp template tx(nx)
5  !$xmp template ty(ny)
6  !$xmp distribute tx(block) onto p
7  !$xmp distribute ty(block) onto p
8  !$xmp align a(*,i) with ty(i)
9  !$xmp align b(*,i) with tx(i)
10 !$xmp align w(*,i) with tx(i)
11
12 call xmp_transpose(b,a,1)
13 call zfft1d(b,ny,0,cy) ! init table
14
15 !$xmp loop on tx(i)
16 do i=1,nx
17    call zfft1d(b(1,i),ny,-1,cy)
18 end do
19
20 !$xmp loop on tx(i)
21 !$omp parallel do
22 do i=1,nx
23    do j=1,ny
24       b(j,i)=b(j,i)*w(j,i)
25    end do
26 end do
27
28 call xmp_transpose(a,b,1)
29 call zfft1d(a,nx,0,cx) ! setup
30
31 !$xmp loop on ty(j)
32 do j=1,ny
33    call zfft1d(a(1,j),nx,-1,cx)
34 end do
35
36 call xmp_transpose(b,a,1)
```

図 10 XcalableMP を用いた Fast Fourier Transform の一部

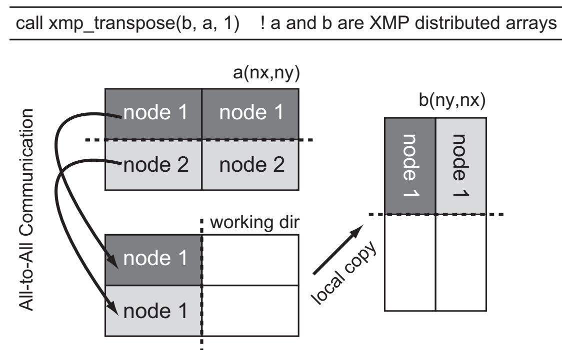


図 11 xmp_transpose() の動作の概要

を生かして，Fortran コードで 1 次元配列として動的に確保して初期化した領域を，図 10 では XMP/Fortran で 2 次元の分散配列として参照している．このような適材適所のプログラミング手段の選択により，性能を落とすことなく生産性を上げることができた．

過去の実装 [9,18] で計測した際は，各計算ノードにつき 1 プロセス 8 スレッドで動作させ，各スレッドが FFTE ライブラリを呼んでいた．本稿では，より効率よく FFTE ライブラリを使うため，スレッド化した FFTE ライブラ

ⓒ 2015 Information Processing Society of Japan

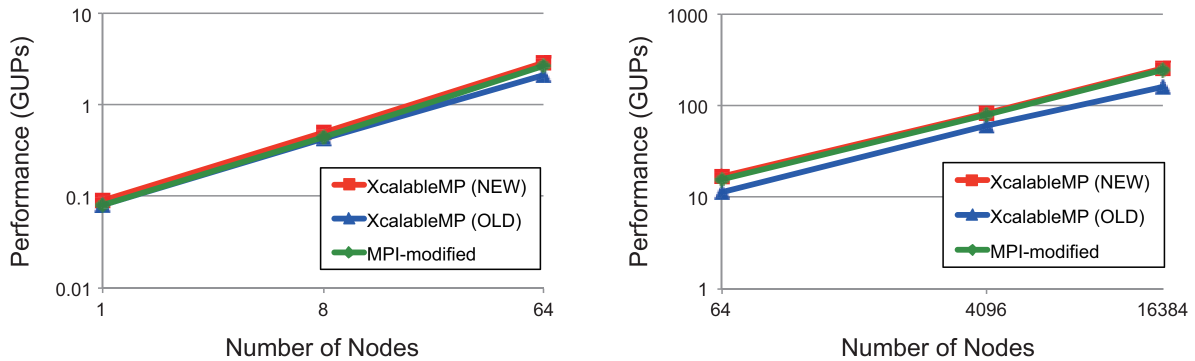


図 9 RandomAccess の性能結果

表 2 FFT におけるノード数と配列のベクトル長

#Nodes	Length of vector	Data size / System memory
36	6,635,520,000	34.3%
144	24,300,000,000	31.4%
576	99,532,800,000	32.2%
2,304	398,131,200,000	32.2%
9,216	1,592,524,800,000	31.0%
36,864	6,522,981,580,800	33.0%
55,296	9,784,472,371,200	33.0%
82,944	12,899,450,880,000	29.0%

リを 1 プロセスが呼ぶように変更した．また，過去の実装では性能の計測範囲に **node** 指示文を用いていたが，本稿の FFT ではプログラムの最初（性能の計測範囲外）に **node** 指示文を移動させた．その理由は，**node** 指示文では，XMP のランタイム内で用いる MPI のコミュニケータを MPI_Comm_dup() を用いて複製しているが，高並列時にその操作の負荷が大きいことがわかったからである．

4.3.2 性能評価

各計算ノードにつき，1 プロセス 8 スレッドで計測を行った．配列 a, b の大きさと利用ノード数は，「京」のキャッシュのヒット率を計算して**表 2** のように決定した．

結果を**図 12** に示す．図 12 において，今回の結果は “XcalableMP (NEW)”，過去の結果を “XcalableMP (OLD)” とした．なお，“XcalableMP (OLD)” は 36,864 ノードまでしか計測していない．36,864 ノードにおけるそれぞれの結果は，XcalableMP (NEW) の性能は 79.45 TFlops であり，XcalableMP (OLD) の性能は 50.08 TFlops である．この結果から，性能が 1.59 倍に向上していることがわかる．また，82,944 ノード利用時の XcalableMP (NEW) の性能は 211.64 TFlops であり，Class 1 の FFT の性能は 205.94 TFlops であるため，XcalableMP (NEW) は，Class 1 とほぼ同じ性能を達成していると言える．

4.4 High Performance Linpack

4.4.1 実装

HPL は密行列の連立一次方程式を解く速度を計測するベンチマークであり，計算システムの演算性能が測定結果に大きく影響する．我々はリファレンス実装と同様に再帰 LU 分解アルゴリズムを BLAS を用いて実装した．Level 3 BLAS と Level 2 BLAS については，「京」が提供している BLAS を用いた．しかし，Level 1 BLAS はスレッド化されておらず性能が低かったため，Level 1 BLAS については，我々が開発したものを用いた．

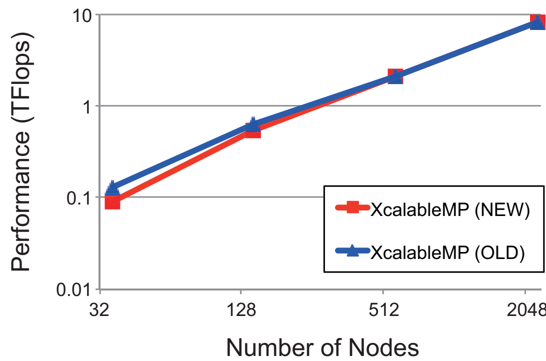
```
1 xmp_desc_t A_desc = xmp_desc_of(A);
2 int ld[2];
3 xmp_array_lead_dim(A_desc, ld);
4 int ld_A = ld[1];
5 cblas_dgemm(CblasRowMajor, CblasNoTrans,
6             CblasNoTrans, local_len_y, local_len_x, NB,
7             −1.0, &A_L[y][0], ld_A_L, &A_U[0][x], ld_A_U,
8             1.0, &A[y][x], ld_A);
```

図 14 分散配列に対する BLAS を用いた行列積

まず，係数行列を格納する 2 次元配列 *A[][]* の各次元を，リファレンス実装と同様にブロックサイクリック分割する．**図 13** に，ブロックサイクリック分割のコード例を示す．2 行目と 3 行目は 2 次元テンプレート t と 2 次元ノード集合 p を定義しており，4 行目は t を $P \times Q$ の上にサイズ NB のブロック毎にサイクリック分散している．5 行目は配列 *A[][]* をテンプレート t に整列させている．

分散配列における行列積の計算を BLAS の DGEMM 関数を用いて行う方法を**図 14** に示す．3.2.3 節と同様に，1〜4 行目は分散配列 *A[][]* の leading dimension を取得し，5 行目以降でその leading dimension を用いて DGEMM 関数を利用している．図 14 では省略しているが，分散配列 *A_L[][]* と *A_U[][]* についても同様に leading dimension を取得している（それぞれ，*ld_A_L* と *ld_A_U* である）．また，分散配列 *A[][]*, *A_L[][]* と *A_U[][]* は，関数の適用範囲の各ノードにおける先頭アドレスの位置をそれぞれ計算し，DGEMM 関数の引数として用いている．

過去の実装 [9, 18] では，**図 15** のようにパネル転送に **gmove** 指示文を用いた．2 次元配列 *A_L[][]* は最初の次元のみブロックサイクリック分割されている．そのため，*A[j+NB:N-j-NB][j:NB]* の要素は *A_L[j+NB:N-*



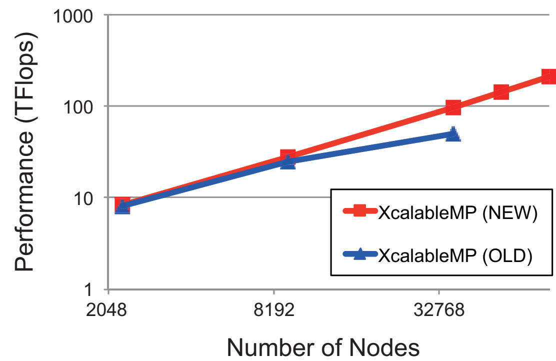


図 12 Fast Fourier Transform の性能結果

```
1  double A[N][N];
2  #pragma xmp template t(0:N−1, 0:N−1)
3  #pragma xmp nodes p(P,Q)
4  #pragma xmp distribute t(cyclic(NB), cyclic(NB)) onto p
5  #pragma xmp align A[i][j] with t(j,i)
```

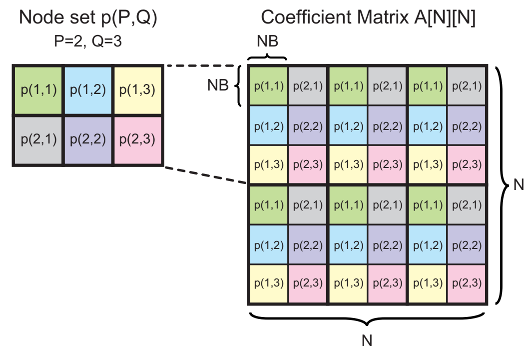


図 13 ブロックサイクリック分割のコード例

```
1  double A[N][NB];
2  #pragma xmp align A_L[i][*] with t(*,i)
3    :
4  #pragma xmp gmove
5    A_L[j+NB:N−j−NB][0:NB] = A[j+NB:N−j−NB][j:NB];
```

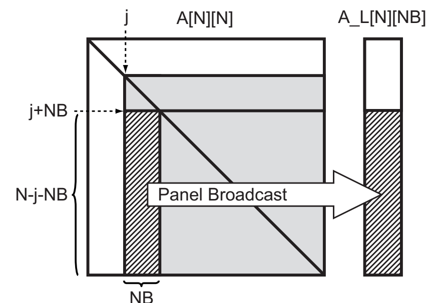


図 15 gmove を用いたパネル転送

```
1  double A_L[N][NB];
2  #pragma xmp align A_L[i][*] with t(*,i)
3    :
4  #pragma xmp gmove async(1)
5  A_L[k:len][0:NB] = A[k:len][j:NB];
6    :
7  for(m=j+NB;m<N;m+=NB){
8    for(n=j+NB;n<N;n+=NB){
9      cblas_dgemm(&A[m][n], ..);
10     if(xmp_test_async(1)){
11       // receive A[k:len][j:NB];
12         :
```

図 16 gmove を用いた非同期パネル転送

j-NB][0:NB] にブロードキャストされる．なお，このパターンの **gmove** 指示文では，XMP ランタイム内で関数 MPI_Bcast() が用いられる．

今回の実装においても，パネル転送には **gmove** 指示文を用いているが，**図 16** のように **gmove** 指示文に **async** 節を加えることで，非同期通信を実現している．なお，図 14 の DGEMM 計算は，図 16 の 9 行目のようにブロック毎に DGEMM 計算を行うように書き換えている．10 行目の関数 xmp_test_async() は非同期通信の到着を確認しており，もし到着していなければ，DGEMM 計算を少しずつ行いながらデータの到着を待つ．すなわち，通信と計算のオーバラップを実現している．なお，**gmove** 指示文と **async** 節を用いた場合に発生する通信には，文献 [31] の Increasing-ring アルゴリズムと同等のものを実装している．

4.4.2 性能評価

各計算ノードにつき，1 プロセス 8 スレッドで計測を行った．配列 *A[][]* のサイズは，システムメモリの 70%を占めるように設定した．なお，4.4.1 節で述べたいくつかの機能は未実装であるため，その箇所は手動でコード変換を行った．

結果を**図 17** に示す．図 17 において，今回の結果は “XcalableMP (NEW)”，過去の実装の結果を “XcalableMP

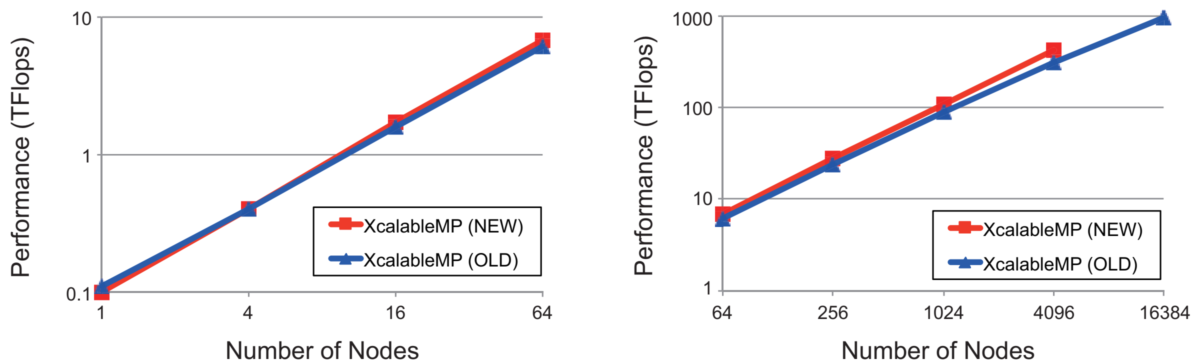


図 17 High Performance Linpack の性能結果

(OLD)” とした．バジェットの都合上, XcalableMP (NEW) は 4,096 ノードまでしか計測できなかった．4,096 ノードにおける XcalableMP (NEW) の性能は 422.94 TFlops（ピーク性能の 81%）であるのに対し，XcalableMP (OLD) の性能は 309.64 TFlops（ピーク性能の 59%）である．この結果から，性能が 1.37 倍に向上していることがわかる．なお，16,384 ノード利用時の XcalableMP (OLD) の性能は 970.97 TFlops（ピーク性能の約 46 %）であり，ノード数の増加に伴い，並列化効率が悪くなるのがわかる．しかしながら，1 ノードのおける XcalableMP (NEW) の性能はピーク性能の 80%程度であるため, XcalableMP (NEW) の並列化効率は高いことがわかる．なお，Class 1 の 82,944 ノード利用時の HPL の性能は 9,796TFlops であり，ピーク性能の 92%を達成している[*1]．このことから，XcalableMP (NEW) の結果のピーク性能比が Class 1 の結果と比較して低い原因は，ローカルのチューニングにあると考えられる．

5. 考察

4.1 節で述べた XMP を用いた STREAM の実装の特徴は，C コンパイラが提供している最適化指示文を利用したことである．XMP の利点として，C 言語および Fortran における既存の資産（e.g. 最適化指示文，コンパイラ，ライブラリ，最適化のための技法 など）をそのまま利用できる点が挙げられる．4.3 節と 4.4 節で述べた外部ライブラリを用いた FFT と HPL の実装についても，この利点は活かされている．外部ライブラリを利用する際は，分散配列のローカルメモリを意識したプログラミングを行う必要があるため，可読性は落ちる．しかしながら，場合に応じてこのような性能チューニングを行えることは XMP の利点であり，性能と生産性のバランスの取れた HPC アプリケーションを作成するための良いデザインであると考えている．

*1 Top500 リストに登録されている「京」の結果には，計算ノードに加え I/O ノードも計算に参加しているため，計 88,128 ノードが利用されている．また，Top500 リストに登録されている結果は 10,510 TFlops であり，ピーク性能の 93%である．

6. まとめと今後の課題

本稿では，データ並列言語に対して，プログラミングの生産性を高くするための要件について述べ，各要件に対する XMP のデザインについて紹介した．次に，XMP の生産性と性能を評価するため，HPCC ベンチマークの 4 つのベンチマークを作成した．その結果，XMP の生産性は高く，さらに MPI で記述された実装と同等の性能を発揮できることを示した．

今後の課題として，省電力性に優れたアクセラレータクラスタへの対応が挙げられる．X10 と Chapel では，各言語の構文を用いて，アクセラレータクラスタへの対応がなされている．XMP においても，OpenACC [32] と XMP を組合せた言語である XcalableACC [33, 34] が提案されているが，アクセラレータ間の片側通信などは未実装であるため，今後も開発を続けていく必要がある．また，XMP は C99 と Fortran95 の拡張であるため，他の PGAS 言語（e.g. UPC++，X10，Chapel など）では可能なオブジェクト指向プログラミングを行うことはできない．アプリケーションの種類やその規模によっては，オブジェクト指向が有効な場合があるため，我々は C++言語および Fortran2003 に対する XMP の拡張を計画している．さらに，今後の計算環境では，生産性と性能に加え，フォールトトレランスやエネルギー消費の明示的なコントロールなども重要になってくると考えられる．それらを言語レベルで対応，もしくは外部ライブラリと組合せて対応していく必要があると考えている．

謝辞 本研究は JST-CREST 研究領域「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」，研究課題「ポストペタスケール時代に向けた演算加速機構・通信機構統合環境の研究開発」による．

参考文献

[1] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, Vol. 17, No. 2, pp. 1–31, August 1998.

[2] Nowicki, M. and Gorski, L. and Grabrczyk, P. and Bala, P. PCJ - Java library for high performance computing in PGAS model. In *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pp. 202–209, July 2014.

[3] A publication of the UPC Consortium, November 2013. https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf.

[4] Yili Zheng and Kamil, A. and Driscoll, M.B. and Hongzhang Shan and Yelick, K. UPC++: A PGAS Extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1105–1114, May 2014.

[5] Kumar, Vivek and Zheng, Yili and Cavé, Vincent and Budimlić, Zoran and Sarkar, Vivek. HabaneroUPC++: A Compiler-free PGAS Library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pp. 5:1–5:10, New York, NY, USA, 2014. ACM.

[6] Charles, Philippe and Grothoff, Christian and Saraswat, Vijay and Donawa, Christopher and Kielstra, Allan and Ebcioglu, Kemal and von Praun, Christoph and Sarkar, Vivek. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, Vol. 40, No. 10, pp. 519–538, October 2005.

[7] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, Vol. 21, No. 3, pp. 291–312, August 2007.

[8] XcalableMP Specification Version 1.2.1, November 2014. http://xcalablemp.org/download/spec/xmp-spec-1.2.1.pdf.

[9] Masahiro Nakao and Hitoshi Murai and Takenori Shimosaka and Mitsuhisa Sato. Productivity and Performance of the HPC Challenge Benchmarks with the XcalableMP PGAS Language. In *7th International Conference on PGAS Programming Model*, pp. 157–171, 2013.

[10] Jinpil Lee. *A Study on Productive and Reliable Programming Environment for Distributed Memory System*. PhD thesis, University of Tsukuba, 2012.

[11] Masahiro Nakao and Jinpil Lee and Taisuke Boku and Mitsuhisa Sato. Productivity and Performance of Global-View Programming with XcalableMP PGAS Language. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pp. 402–409, Washington, DC, USA, 2012. IEEE Computer Society.

[12] Omni Compiler. http://omni-compiler.org.

[13] GASNet Communication System. http://gasnet.lbl.gov.

[14] Hiroyuki Miyazaki, Yoshihiro Kusano, Naoki Shinjou, Fumiyoshi Shoji, Mitsuo Yokokawa, Tadashi Watanabe. Overview of the K computer. *FUJITSU SCIENTIFIC and TECHNICAL JOURNAL*, Vol. 48, No. 3, pp. 255–265, 2012.

[15] Dongarra, J. and Luszczek, P. Introduction to the HPC-Challenge Benchmark Suite. Technical report, ICL Technical Report, ICL-UT-05-01, (Also appears as CS Dept. Tech Report UT-CS-05-544), 2005.

[16] HPC Challenge Benchmarks. http://icl.cs.utk.edu/hpcc/.

[17] HPC Challenge Awards Competition. http://www.hpcchallenge.org.

[18] HPC Challenge Award Competition Class 2 Awards in 2013. http://www.hpcchallenge.org/custom/index.html?lid=103&slid=263.

[19] HPC Challenge Award Competition Class 1 Awards in 2012. http://www.hpcchallenge.org/custom/index.html?lid=103&slid=257.

[20] HPC Challenge Award Competition Class 2 Awards in 2014. http://www.hpcchallenge.org/custom/index.html?lid=103&slid=272.

[21] P. Jones. Parallel Ocean Program (POP) user guide. Technical Report Technical Report LACC 99-18, Los Alamos National Laboratory, March 2003.

[22] Tomoya Adachi, Naoyuki Shida, Kenichi Miura, Shinji Sumimoto, Atsuya Uno, Motoyoshi Kurokawa, Fumiyoshi Shoji, and Mitsuo Yokokawa. The design of ultra scalable mpi collective communication on the k computer. *Comput. Sci.*, Vol. 28, No. 2-3, pp. 147–155, May 2013.

[23] PC Cluster Consortium. http://www.pccluster.org/en/.

[24] Charles H. Koelbel and David B. Loverman and Robert S. Shreiber and Guy L. Steele Jr. and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[25] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pp. 7–1–7–22, New York, NY, USA, 2007. ACM.

[26] MVAPICH Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[27] Ponnusamy, R. and Choudhary, A. and Fox, G. Communication overhead on the CM5: an experimental performance evaluation. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pp. 108–115, Oct 1992.

[28] FFTE: A Fast Fourier Transform Package. http://www.ffte.jp.

[29] David H. Bailey. FFTs in external or hierarchical memory. *Journal of Supercomputing*, Vol. 4, pp. 23–35, 1990.

[30] Van Loan, C. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, 1992.

[31] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl/.

[32] OpenACC. http://www.openacc-standard.org.

[33] Hitoshi Murai and Masahiro Nakao and Takehiro Shimosaka and Akihiro Tabuchi and Taisuke Boku and Mitsuhisa Sato. XcalableACC - a Directive-based Language Extension for Accelerated Parallel Computing. In *SC14 poster*, 2014.

[34] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Bokut, and Mitsuhisa Sato. Xcalableacc: Extension of xcalablemp pgas language using openacc for accelerator clusters. In *Proceedings of the First Workshop on Accelerator Programming Using Directives*, WACCPD '14, pp. 27–36, Piscataway, NJ, USA, 2014. IEEE Press.

[35] Guohua Jin and Mellor-Crummey, J. and Adhianto, L. and Scherer, W.N. and Chaoran Yang. Implementation and Performance Evaluation of the HPC Challenge Benchmarks in Coarray Fortran 2.0. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 1089–1100, May 2011.

ⓒ 2015 Information Processing Society of Japan

表 A·1 PGAS 言語およびリファレンス実装との行数の比較

Langauge	STREAM	Random Access	FFT	HPL
XcalableMP	69	253	205	313
Coarray Fortran	63	409	450	786
PCJ	180	146	498	N/A
X10	60	143	236	708
Chapel	72	112	N/A	658
Reference (MPI)	329	787	1,904	8,800

[36] Andrew I. Stone and John M. Dennis and Michelle Mills Strout. Evaluating coarray fortran with the cgpop miniapp. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS)*, October 2011.

付　　録

A.1 PGAS 言語およびリファレンス実装との行数の比較

生産性評価の指標としては，定量的に表現できることから，行数もしくは行数をベースとした値（e.g. 性能を行数で割った値 [35]，逐次コードから並列コードを作成するのに要した行数 [36] など）が用いられているが，行数だけで上記のコストを測ることはできない．行数は評価の 1 つとしてのみ用いることが妥当と考える．

参考として，XMP と HPCC Awards Competition Class 2 に投稿された他の PGAS 言語における各ベンチマークを実装するのに要した行数（空行とコメント行を除く）を**表 A·1** に示す [17]．また，MPI で記述されたリファレンス実装の行数についても示す．注意点として，各実装によって採用しているアルゴリズムが異なるため，単純な行数の比較はできない．特に，リファレンス実装の FFT と HPL は，1 つのベンチマークに複数のアルゴリズムが実装されている．

ⓒ 2015 Information Processing Society of Japan