情報処理学会論文誌　コンピューティングシステム　Vol. 3　No. 3　153–165 (Sep. 2010)

# 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

李　珍泌[†1]　朴　泰祐[†1,†2]　佐藤三久[†1,†2]

分散メモリ型並列計算機における標準的なプログラミングモデルである MPI は高いプログラミングコストが問題として指摘されている．並列プログラミングをより簡単にするため，C と Fortran をベース言語として，指示文により拡張した並列プログラミングモデル XcalableMP が提案されている．XcalableMP は OpenMP-like な指示文を提供し，典型的なデータ並列化手法が有効なアプリケーションに対して逐次コードからのシームレスな並列化を可能にする．また，CAF-like な言語拡張を取り入れることにより，ノード内のメモリイメージとノード間通信を意識した効率的な並列化の記述が可能である．また，性能チューニングのため，OpenMP や MPI を XcalableMP と併用することもできる．本論文では，XcalableMP について述べ，そのコンパイラの実装と性能評価について述べる．性能評価には HPC Challenge Benchmark の並列化を行い，XcalableMP が少ないプログラミングコストで分散メモリ向け並列性記述を実現できることを確認した．

# Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory System

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

Although MPI is a de-facto standard for parallel programming on distributed memory systems, writing MPI programs is often a time-consuming and complicated process. XcalableMP is a language extension of C and Frotran for parallel programming on distributed memory systems that helps users to reduce those programming efforts. XcalableMP provides two programming models. The first one is the global view model, which supports typical parallelization based on the data and task parallel paradigm, and enables parallelizing the original sequential code using minimal modification with simple, OpenMP-like directives. The other one is the local view model, which allows to use CAF-like expression to describe internode communications. Users can even use MPI and OpenMP explicitly in our language to optimize the performance explicitly. In this paper, we introduce XcalableMP, the implementation of the compiler, and the performance evaluation result by global view parallelization in XcalableMP. For the performance evaluation, we parallelized HPC Challenge Benchmark in XcalableMP. It shows that users can describe the parallelization for distributed memory system with a small modification to the original sequential code.

## 1. はじめに

現在，PC クラスタは高性能コンピューティングの分野で広く使われている．そのプログラミングモデルとして普及しているのが Message Passing Interface（MPI）である．ノード間のデータの送受信やブロードキャストなどの集団通信を行う API 関数が提供され，データの分散やノード間通信などのすべての並列処理をユーザが明示的に記述しなければならない．その結果，プログラミングコストが膨大になり PC クラスタの利用を妨げる一因となっている．

一方，近年その普及が目覚ましいマルチコアプロセッサにおいては OpenMP による指示文ベースの並列プログラミングが可能である．逐次のソースコードに対して並列化に関する情報を記述した指示文を挿入することで，コンパイラによる並列化が行われる．そのため，OpenMP は少ないプログラミングコストで逐次コードからのシームレスな並列化を可能にする．

分散メモリ上で利用できるよりシンプルで強力な並列プログラミングモデルを目指して，XcalableMP [1),⋆1]（以後，XMP）が提案されている．XMP は C と Fortran 言語をベースに OpenMP-like な指示文の追加や言語構文の拡張を行うものである．まず，分散メモリ上での並列化を記述するための指示文をユーザに提供する．XMP はコンパイラやランタイムによる自動的な並列化を提供せず，そのための処理はすべてユーザによって明示的に記述されるものとする．これによりコンパイラによるコード変換のイメージがユーザに対して明確

†1 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
†2 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba

⋆1 XcalableMP の言語仕様は，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」プロジェクトの次世代並列プログラミング言語検討委員会にて検討されている．本論文で述べられる言語仕様（指示文の名前や記号，記法など）は今後変更される可能性がある．

になり，性能のチューニングを容易にする．性能チューニングのため，XMP はノード間通信を記述できるようにベース言語を拡張する．または，MPI 関数の呼び出しを直接挿入することで通信を記述することも可能である．これらの要素を利用することで分散メモリ型並列計算機において効率の良い並列化を，少ないプログラミングコストで行うことができる．

本論文の構成は以下のとおりである．2 章では今まで提案されてきた並列言語モデルと XMP の比較を行い，3 章では XMP の概要と設計におけるコンセプトを延べる．4 章では XMP による並列プログラミングモデルについて説明を行う．5 章ではコンパイラの実装について述べ，6 章でベンチマークによる性能評価を行う．7 章ではマルチコアクラスタ上でのハイブリッド並列化について検討を行い，8 章で本論文をまとめる．

## 2. 関 連 研 究

分散メモリ型並列計算機での並列プログラミングのために様々な言語モデルやライブラリが提案されてきた．その中でも代表的なものとして，Unified Parallel C（UPC），Co-Array Fortran（CAF）のような Partitioned Global Address Space[2]（PGAS）言語や，High Performance Fortran（HPF）があげられる．

HPF は指示文によって並列化を記述する．HPF ではデータの分散はユーザによって指示文の形式で与えられるが，ループの並列化や通信はコンパイラによるプログラムの解析によって生成，挿入される．分散されたデータのイメージや通信のタイミングがユーザに対して不明瞭であり，それを操作する手段も提供されないため，性能をチューニングすることは困難である．

XMP は HPF のような自動的な通信の生成を行わない．コンパイラやランタイムによって自動的に行われる並列化はその実体をユーザに隠すことでプログラミングコストを減らすというメリットがある．しかし，すべてのアプリケーションに対して最適な性能を提供することは不可能である．XMP は通信を含めたすべての処理がユーザによって明示的に与えられる．したがって，変換されるコードや分割されるメモリのイメージがユーザに対して明確である．それによってユーザは外部ライブラリを用いた並列アルゴリズムを自由に記述することができる．このようなプログラミングモデルは自動並列化の言語モデルよりプログラミングコストが高いものの，チューニングによる性能向上の余地をユーザに与える．

UPC はノード内メモリの一部をすべてのノードで参照できる「shared メモリ」として宣言することができる（対称的な概念として，ノード独立の領域である「private メモリ」がある）．shared メモリへの参照はローカルメモリアクセスと同様に行う．コンパイラは shared メモリへの参照を解析し，ノード間通信を生成する．しかし HPF と同様，自動的な通信だけでつねに最適な性能を達成することは困難である．また，shared メモリはランタイムによって管理されるため，（MPI のような通信ライブラリの使用が禁止されているなど）その利用に制約があり，そこで行われる通信を操作する手段はユーザに提供されない．したがって，shared メモリのみを用いる並列化は低コストで記述できる反面，実用的なパフォーマンス達成することは困難である．UPC を用いて性能を出すためには，shared メモリから private メモリにデータを転送し，なるべく private メモリのみで計算を行うことが推奨される．データの転送には UPC のリモートメモリアクセス API 関数が用いられる．したがって，逐次コードから並列化を行っていくことは困難であり，MPI と同様，並列コードを一から記述していく必要がある．

XMP は逐次コードに指示文を追加することで並列化を行う．分散メモリ上の並列化の中でも典型的かつ，最適な性能で実装できるものを指示文で記述できるようにしている．したがって，並列アルゴリズムが XcalableMP の指示文で十分に記述できるものであるならば，逐次コードに対してわずかな修正（指示文の挿入）を行うだけで高い性能を達成することが保証される．

CAF は変数に「Co-Array 次元」を導入することでノード間通信を実現している．Co-Array 次元にノード番号を指定することで他のノードのデータを参照することができ，代入文と組み合わせることで get や put などの片側通信を記述することができる．CAF は MPI と同等のレベルで並列処理を記述する．したがって，ユーザの努力次第で高い性能を得ることができるが，データの分割やループ並列化などのすべての処理を手動で記述する必要があり，プログラミングコストが問題となる．

CAF は単体で用いるにはプログラミングコストが高いという問題点があるが，通信を記述する手段としては MPI より直感的である．XMP では指示文によるプログラミングモデルを提供することでプログラミングコストをおさえつつ，CAF をベースとした言語機能を提供することで言語の記述性を高めている．

既存の並列言語が持つ問題を解決するため，著者らは過去に指示文による並列言語，OpenMPD[9] を提案した．OpenMPD は OpenMP や HPF と同様，指示文による並列化の記述を行う．しかし，コンパイラによる自動的な通信の生成を行わず，ユーザによる明示的な通信の記述が必要である．データ並列化で行われる典型的な通信を指定する指示文をユーザに提供する．しかし，まだ実験段階の言語モデルであったため，実際のアプリケーションを記述するには機能が不十分であるという問題点があった．たとえば，配列の一次元分割のみを

サポートしているため，多次元分割による並列化ができない．また，外部ライブラリとの併用を可能にするため，ノード間で分散される配列であっても，全領域を各ノードで重複して宣言するような仕様になっている[⋆1]．また，指示文ベースのプログラミングモデルの限界として，不規則なデータの分割や通信パターンなど，指示文が対象としていない処理を記述できないという問題点がある．これは VPP/XPF Fortran など指示文をベースにした他の言語モデルでも共通した問題である．

XMP は OpenMPD のコンセプトを受け継ぎ，明示的な並列化を行うが，多次元分割をサポートし，通信を自由に記述する手段を提供するなど，新しい言語機能が多数追加されている．また，3 章で述べるように分散される配列のローカルイメージをユーザに公開することで，配列を重複宣言することなく外部ライブラリの利用を可能にした．また，ローカルメモリイメージの利用に MPI のような外部ライブラリを用いるだけでなく，言語構文でリモートメモリアクセスを記述できるようにして記述性を高めている．これらの機能を用いることによって，OpenMPD や VPP/XPF Fortran などの指示文ベースの言語では不可能だった並列アルゴリズムの自由な記述が可能となる．

既存の並列言語モデルのどれもがプログラムのしやすさと記述性のどちらかを優先しており，両立には成功していない．3 章，4 章で述べるように，XMP は 2 つのプログラミングモデルを 1 つの言語で提供することによってこの問題点を解決している．

## 3. XcalableMP の概要

XMP は逐次の手続き型言語である C と Fortran をベースにして，分散メモリ上での並列プログラミングのための独自の言語拡張を行っている．言語拡張の大半は OpenMP-like な指示文であり，ベース言語の仕様や構文の変更は最小限にとどめる．XMP の言語モデルの大原則は自動的な並列化を行わず，すべての処理はユーザによって明示的に与えられるということである．このような言語設計は自動並列化を提供する他の言語モデルよりプログラミングコストが高くなる傾向がある．その反面，変換されるコードのイメージがユーザに対して明確であるため，外部ライブラリを用いた性能の最適化が容易であるというメリットがある．

### 3.1 実行モデル

XMP は分散メモリをターゲットとした並列言語である．実行単位であるプロセスを XMP では「ノード」と定義する．XMP の実行モデルは MPI と同様，Single Program Multiple Data（SPMD）である．つまり，通常実行時には各ノードで同じ処理が実行される．ソースコード上で宣言されたデータは，それが XMP の言語機能によって分割されると宣言されない限り，各ノードで重複して確保される．

XMP では自動的な通信の生成を行わないため，メモリアクセスはつねにローカルメモリのデータに対する参照である．他のノードのデータにアクセスするためには，XMP が提供する通信記法を用いて明示的なノード間通信を行わなければならない．

各ノードでのスレッド数は実行開始時には 1 つである．7 章でマルチスレッド化によるハイブリッド並列化に関する考察を行う．

### 3.2 プログラミングモデル

XMP では少ないプログラミングコストと記述力を両立させるため，1 つの言語の中で「グローバルビューモデル」と「ローカルビューモデル」という 2 つのプログラミングモデルを提供する．ここでは言語モデルの概要を与え，各々の言語機能に関しては 4 章で説明を行う．

#### 3.2.1 グローバルビューモデル

グローバルビューモデルは分散メモリ上で OpenMP-like な指示文による並列プログラミングを可能にするものである．XMP の指示文は HPF をベースにしたものが多く，template，shadow などの概念を HPF から取り入れている．しかし，データ分散のみを与える HPF と異なり，すべての処理をユーザが明示的に与える必要があるため，データの分散以外にもループ文のワークシェアリング，バリアやリダクションなど集団通信のような典型的な並列化手法を指示文として提供する．グローバルビューモデルで記述できる処理は分散メモリ上の並列化の中でも典型的なもので，最適な性能で実装できるものに絞られている．これらの処理は多くのアプリケーションの中で活用できるもので，コンパイラによって最適な並列コードが生成されることを保証される．たとえば，shadow 領域の宣言と同期によって配列の隣接領域の交換を記述した姫野ベンチマークの並列コードが，MPI 版と同程度の性能を示すことが確認されている[9),⋆2]．グローバルビューモデルを用いることで逐次コードから

⋆1 配列のローカルメモリイメージが逐次コードと同じであるため，コンパイラによる index 変換が不要である．したがって，逐次コードでの index のまま，配列を外部ライブラリの引数として利用することができる．

⋆2 参考文献 9) の中で登場する sleeve 領域は XMP における shadow 領域と同じ概念であり，指示文の挿入によって領域の宣言と同期を記述する．OpenMPD では同期を行う領域のサイズを指定できず，sleeve 領域全体の同期が行われたが，XMP では同期を行う領域のサイズを指定するように指示文の拡張を行った．

指示文を挿入するといったインクリメンタルな並列化が可能である．

しかし，指示文ベースのプログラミングモデルには記述能力に限界がある．特にノード間通信の場合，典型的な集団通信で記述できないものに対しては何らかの手段によってデータのやりとりを直接記述しなければならない．XMP ではローカルビューモデルというもう 1 つのプログラミングモデルを提供することでこの問題点を解決する．

#### 3.2.2　ローカルビューモデル

従来の MPI ではデータの分散や通信の記述をユーザが手動で行う．このようなモデルではデータの局所性やノード間通信を意識した並列化を行うことで高い性能を引き出すことができる．XMP ではこのようなプログラミングモデルをローカルビューモデルと呼ぶ．ローカルビューモデルではデータの分割をユーザが手動で行うため，メモリ領域のイメージがグローバルビューモデルと比べ，より明確である．このような明確なメモリイメージは外部ライブラリを利用するときに有用である．たとえば，データ配列を直接 MPI 関数の引数として与えて，ノード間通信を記述することも可能である．

ローカルビューモデルでの通信をより簡単に記述できるよう，XMP では Co-Array Fortran をベースにした言語拡張を提供する．変数宣言や代入文を拡張した構文を用いることで片側通信を簡単に記述することができる．

#### 3.2.3　2 つのモデルの切替え

XMP の他の言語モデルにない特徴として，同じ変数に対してグローバルビューモデルとローカルビューモデルの切替えを行うことが可能であることがあげられる．たとえば，グローバルビューモデルで分割割当てが宣言された配列に対して，そのローカルイメージをユーザに提供することで，ローカルビューモデルでのより自由な並列アルゴリズムの記述を可能にする．したがって，同一の配列に対して，最初はグローバルビューモデルによる簡単な並列化を行い，性能のチューニングのためローカルビューモデルに切り替えるということも可能である．

## 4.　XcalableMP による並列プログラミング

本章では XMP が持つ言語機能を紹介し，それらを用いた並列プログラミングについて述べる．

### 4.1　グローバルビューモデル

XMP のグローバルビューモデルは指示文の挿入による並列化を行うものである．**図 1** にソースコードの例を示す．C 言語で記述された逐次コードに並列化を指定する指示文を追加

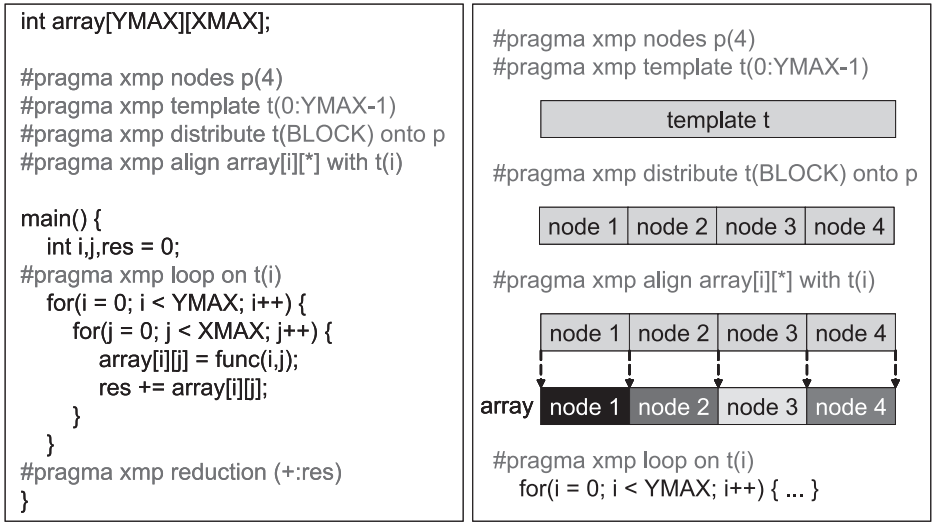


**図 1**　template によるデータ並列化
Fig. 1　Data parallelization using template.

している．すべての指示文は「#pragma xmp」で始まる．

#### 4.1.1　template を用いた index 空間の分割

nodes 指示文はプログラムを実行するノード集合の名前と形状を宣言する．図 1 の例では 4 つのノードで構成される一次元のノード集合に p という名前をつけている．template とは index の集合を表現する仮想的な配列である．ここでの index とはデータ配列やループ文の反復の添え字の総称である．template の属性として名前，各次元の上限値と下限値が与えられえる．

XMP では配列の分散やループ文のワークシェアリングは template を用いて行われる．仮想的な index 空間である template を distribute 指示文で各ノード上に割り当て，align 指示文と loop 指示文を用いることで配列とループ文の並列化を行う．template による並列化のイメージを図 1 に示す．仮想的な index 空間である template を用いて配列の分散やループ文の並列処理を記述している．

distribute 指示文は template を各ノード上に分散配置することを宣言する．その属性として各次元の分割方式を指定することができる．ブロック分割やサイクリック分割などが利用で

きる．align 指示文は配列の分割を記述するものである．template 空間と配列の index 関係を宣言することで配列の割り当て方を指定する．図 1 の例は配列の要素 a[i][0]〜a[i][XMAX-1]（以後，このように index が連続な領域を array section を用いて a[i][0:XMAX-1] のように表す）が t(i) が割り当てられたノード上に宣言されることを記述したものである．その結果，配列 array は二次元方向でブロック分割される．align によって分割された配列は各ノードで割り当てられた分だけがメモリ上に確保される．

#### 4.1.2　ループ文の並列実行

ループ文の並列化には loop 指示文を用いる．loop 指示文の直後に記述されたループ文は各ノードで処理を分担するように並列化される．ループ文の反復をどう分割するかは template やノード集合を指定することで決定する．図 1 の例ではループ文が配列 array を処理している．array は align 指示文と template t によって分割されているため，ループ文の並列化も配列の分散と整合しなければならない．したがって loop 指示文で template t を指定し，index の分割の仕方を指定している．図 1 の例では t(i) を所有するノードがイテレーション i を実行するようにしてループ文がつねにローカル配列にアクセスするようにしている．

このように，distribute や loop 指示文は直接配列やループ文を並列化することはできない．これらの並列化は template を経由して行わなければならない．

#### 4.1.3　配列の重複宣言と通信の記述

XMP ではループ文の並列化と配列の分散の整合はユーザ責任である．XMP におけるメモリアクセスはつねにローカルメモリに対するものであるため，他のノードに割り当てられた領域にアクセスした場合はエラーとなる．割り当てられていない（他のノードに割り当てられた）領域にアクセスする場合は領域を参照側と被参照側で重複宣言し，指示文により明示的な同期を行う必要がある．

配列の各要素に対する計算が，すぐ隣の要素に依存するようなパターンは，たとえば偏微分方程式の空間差分など，多くのアプリケーションで見られる．そのような場合は各ノードで割り当てられるデータ領域の境界要素を重複宣言し，必要に応じて通信を行わなければならない．XMP はこのような並列化を記述するための指示文 shadow と reflect を提供する．指示文の記述と動作を**図 2** に示す．shadow 指示文の属性として，重複宣言する領域の次元と大きさ（下端と上端の要素数）が指定される．図 2 の例では第二次元方向の両端に要素 1 個分の領域が余分に宣言される（以下 shadow 領域）．shadow 領域の index は隣接するノードで宣言された境界要素に対応する．しかし，その実体はローカルに確保されたメモリ領域であるため，正しい値を参照するためにはノード間通信による同期が必要である．

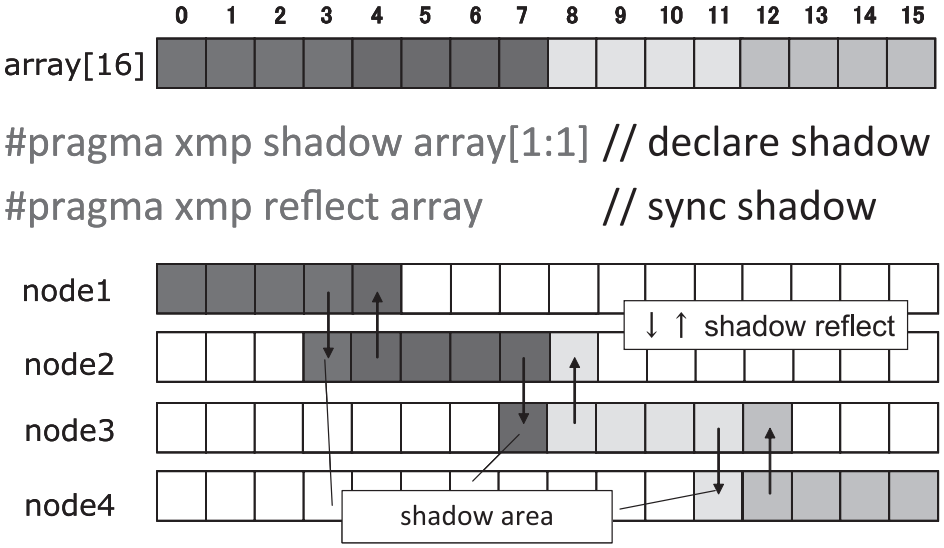


**図 2**　shadow 領域の宣言と同期
Fig. 2　Shadow declaration and synchronization.

shadow 領域の同期には reflect 指示文が用いられる．XMP コンパイラは，reflect 指示文を記述した場所に，shadow 領域に対する通信を挿入する．その結果，reflect 指示文の直後では shadow 領域が隣接ノードの境界領域と同じ値を持つことが保証される．

必要なデータの index が不明なときは，分散されたすべての要素を集約することで配列の同期を行い，問題を解決することができる．shadow 領域のサイズを「*」と指定することですべての領域の重複宣言を行うことができる．このような領域を full shadow と呼ぶ．full shadow に対する reflect 指示文は MPI における MPI_Allgather() と同等である．したがって，full shadow の宣言と同期においてはメモリの消費や通信のコストに注意しなければならない．

配列間のデータ移動のためにも特別な構文が用意されている．Fortran では配列間のデータ移動を簡潔に記述するため，部分配列構文を提供する．XMP は Fortran だけでなく，C 言語バージョンにも部分配列構文を導入する．**図 3** の代入文は配列 A の A[0][0] から A[0][N-1] までの要素を L[0] から L[N-1] までの領域に順次代入する．各ノードに分散された配列にこのような操作を行う場合，必要なデータがすべてローカルメモリに存在するとは限らない．

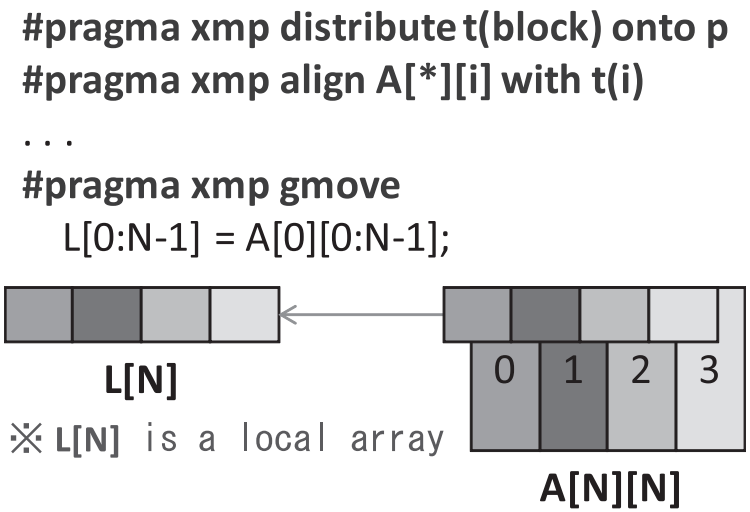


**図 3**　gmove 指示文による通信の記述
Fig. 3　Global data movement using gmove directive.

図 3 では対象となるデータが各ノードに分散されているため，ノード間通信によりデータを集めなければならない．XMP では代入文においてノード間通信が必要な場合，その通信を生成するように指示することができる．直前に gmove 指示文を記述することで，続く代入文がノード間通信を必要とすることをコンパイラに知らせる．コンパイラは代入文の左辺と右辺の変数がどのように分割されているかを調べ，正しい通信を生成する．

shadow 領域の同期や gmove 指示文による通信のほかに，典型的な通信のパターンとしてブロードキャスト，リダクションなどを行う指示文が提供される．

### 4.2　ローカルビューモデル

ローカルビューモデルでは各ノードでのローカル配列のイメージとノード間通信を強く意識した並列プログラミングを行う．XMP ではローカルビューモデルの記述手法として CAF の仕様も用いる．CAF の実行モデルは XMP と同じ SPMD であるため，2 つのモデルが違和感なく混在するように言語の設計を行うことができる．

**図 4** に Co-Array の宣言と言語拡張による通信の記述を示す．XMP の Fortran バージョンでは CAF と同じ書式で CoArray の宣言と通信を記述する．C 言語バージョンにおいては独自の書式を導入し，Fortran バージョンと同等の機能を提供する．Co-Array は変数に

```
<Fortran version>
real dimension A(N)[*]      ! declare co-array
B(1:N) = A(1:N)[1]          ! get A(1:N) from node1
<C version>
double A[N];
#pragma xmp coarray A[N]:(*)
B[0:N-1] = A[0:N-1]:(1);
```

**図 4**　Co-Array の宣言と通信の記述
Fig. 4　Co-Array declaration and notation.

Co-Array 次元という新しい次元を持たせたものである．Co-Array 次元とはデータが宣言されたノード集合を表すものである．Co-Array 次元をノード番号で参照することで他のノードの値を参照することができる．それを代入文と組み合わせることで片側通信によるノード間通信を記述する．図 4 の例では Co-Array として宣言された配列 A からローカル配列 B にデータを代入している．Co-Array 次元に 1 が指定されることにより，ローカル配列 B にはノード 1 が持つ配列 A のデータが代入される．

CAF の言語拡張を用いることで，MPI が提供する片側通信と同等の処理を，MPI に比べて少ないプログラミングコストで記述する．したがって，グローバルビューモデルで記述しきれないアプリケーションに対して，ローカルビューモデルを用いた独自の並列アルゴリズムを用いることも可能である．

ローカルビューモデルのもう 1 つのメリットはメモリイメージが明確なゆえ，外部ライブラリとの親和性が高いということである．たとえば，MPI 関数の引数として配列を渡し，ノード間通信を記述することが可能である．

一方，グローバルビューモデルで分割された配列を外部ライブラリで利用するためにはローカルな領域の先頭を指す index を取得し，ノードに割り当てられた配列の先頭アドレスを計算する必要がある．また，関数引数としてオフセットなどが要求される場合，ローカルなメモリイメージに合わせて index の変換を行わなければならない．こういった場合，最初から分割後のローカルイメージで配列を扱った方が便利である．

XMP では同一の配列に対して，グローバルビューモデルとローカルビューモデルの切替えを行うことができる．**図 5** に local_alias 指示文によるモデルの切替えを示す．local_alias 指示文はグローバルビューモデルで分割された配列の実体に別名を与えてローカルビューモデルで操作できるよう，ユーザに提供するものである．配列 a は 4 ノードで分割され，大きさ 1 の shadow 領域を持つ．したがって，1 ノードあたり 5 個の要素が割り当てられる．たとえば，ノード p(2) に割り当てられた配列 a の要素は a[3]，a[4]，a[5] であり，両端に要素 1 個分の shadow 領域を持つ．local_alias の宣言によってノード p(2) の a の実体である a[2:6]（shadow 領域含む）に b という名前が与えられる．C 言語ベースの XMP では配列はすべて index 0 から始まるため，b は 0 から 4 までの index を持つ．Fortran ベースの XMP は開始 index を指定することが可能である．local_alias によって分割後のイメージ b を得ることができた．このインタフェースを利用することによって，グローバルビューで並列化したプログラムをローカルビューでチューニングする，または外部ライブラリを用いて処理のモジュール化を行うなどが可能となる．

今まで述べたものと逆のアプローチ，つまり，ローカルビューモデルからグローバルビューモデルの切替えも考えられるが，現在の言語仕様では考慮されていない．

```
double a[12];
#pragma xmp nodes p(4)
#pragma xmp template t(0:MAX-1)
#pragma xmp align a[i] with t(i)
#pragma xmp shadow a[1:1]
double *b;
#pragma xmp local_alias b => a
```

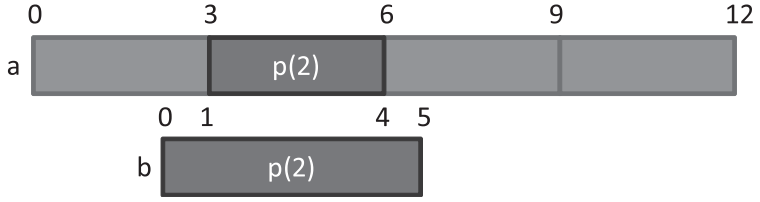


**図 5**　local_alias 指示文の記述例
Fig. 5　Example of local_alias directive.

## 5. XcalableMP コンパイラの実装

XMP コンパイラは C または Fortran のベース言語と XMP の言語拡張で記述されたソースコードを読み込み，並列コードに変換する．XMP はユーザに指示された並列化のみを行うため，指定外の自動的な通信の生成や処理の並列化などは行わない．指示文などの言語拡張で得られた明示的な並列化情報をもとに並列コードを生成する．

コンパイラによるコード変換の例として，図 1 から生成される並列コードを**図 6** に示す．配列の分散割当てや index 分割を行うため，ランタイムライブラリが呼び出される．ランタイムライブラリが生成した情報を収納するために内部変数が宣言される．loop 指示文が記述された場合，コンパイラはプログラムの構造を変更（ループ文の処理範囲や実行するノードを制限）することで並列コードを生成する．通信を記述する指示文（barrier，reduction など）はノード間通信を行うランタイムライブラリ関数呼び出しに置き換えられる．

```
int **__array_addr;
__xmp_array_handle_t *__array_handle;                    // array descriptor

__xmp_main() {
  int i,j,res = 0, __local_i_lower, __local_i_upper;
  __local_i_lower = __xmp_get_lower(__array_handle, ...);  // calc lower bound
  __local_i_upper = __xmp_get_upper(__array_handle, ...);  // calc upper bound
  for(i = __local_i_lower; i < __local_i_upper; i++) {     // work-sharing
    for(j = 0; j < XMAX; j++) {
      *__XMP_GET_ADDR_2(__array_addr, i, j, ...) = func(i, j);
      res += *__XMP_GET_ADDR_2(__array_addr, i, j, ...);
    }
  }
  __xmp_allreduce(&res, ...);                              // reduction
}

__attribute__((constuctor)) static void __xmp_consturtor() {
  // consturctor functions (nodes, templates, arrays, ...)
}
```

**図 6**　コンパイラによって生成される並列コードの例
Fig. 6　Example of translated code.

現在，C 言語ベースの XMP コンパイラの実装を進めている．テンプレートの宣言と分割，配列の align，ループの並列化など，データ並列に関わる基本的な機能の実装を終えている．性能評価で行った並列化作業のほとんどはコンパイラによるものであるが，gmove 指示文の変換のみは実装が完了しておらず，手動で通信関数の挿入を行った．試作したコンパイラはノード間通信のためにランタイムライブラリの中で MPI を利用している．Co-Array のリモートメモリアクセスも MPI2 の片側通信によって実装される．

## 6. 性 能 評 価

ここでは試作した XMP コンパイラを用いたベンチマークの並列化と，その性能評価の結果について述べる．評価対象は HPC Challenge Benchmark [3] の Linpack と FFT である．両者について，XMP のグローバルビューモデルによる並列化を行った．

### 6.1 評 価 環 境

評価には T2K-Tsukuba [10] システムの実行（物理）ノードを 2 ノードから最大 32 ノード利用した．評価環境のノード構成を**表 1** に示す．

### 6.2 Linpack の並列化

Linpack は LU 分解に基づき，連立方程式の解を求めるベンチマークである．行列やベクトルに対する操作をいかにして並列化するかが問題となる．また，ピボット選択と行の交換の際にデータの分散の仕方によってノード間通信が必要となる．

#### 6.2.1 並列化の記述

**図 7** に XMP による Linpack の並列コードの要点を示す．データ並列化を記述するために，グローバルビューモデルの指示文を逐次コードに挿入している．2 次元行列やベクトルを表すデータ配列は 1 つの次元のみ cyclic に分割する．分割された配列を処理する daxpy のような関数を並列化するためには，関数内変数の宣言部で指示文を用いて配列の分散を記述する．その場合，引数として渡される配列の分散と関数内の宣言が整合しなければならない．Linpack では LU 分解の際にピボットの選択と行の交換を行う．ピボットはローカルメモリとして宣言されるため，行列から代入を行うときにはノード間通信が必要になる．gmove 指示文と部分配列の構文を利用することでこれらの操作を簡単に行うことができる．ユーザはグローバル index から見たデータの移動を部分配列の代入として記述し，gmove 指示文を挿入することで配列の分散を気にする必要なく，コンパイラに通信を生成させるこ

**表 1**　評価環境のノード構成
Table 1　Evaluation environment.

| | |
|---|---|
| CPU | AMD Opteron Quad-core 8000 series<br>2.3 Ghz × 4 sockets（16 cores） |
| Memory | 32 GB |
| Network | Infiniband DDR（4 rails） |
| OS | Linux kernel 2.6.18 x86_64 |
| MPI | MVAPICH2 1.2 |

```
#pragma xmp nodes p(*)
#pragma xmp template t(0:N-1)
#pragma xmp distribute t(CYCLIC) onto p
double a[N][N], pvt_v[N];
#pragma xmp align a[*][i] with t(i)
...
void dgefa(double a[N][N], int n, int ipvt[N]) {
#pragma xmp align a[*][i] with t(i)
 ...
 for (k = 0; k < nm1; k++) {
  ...
#pragma xmp gmove
   pvt_v[k:n-1] = a[k:n-1][l];
  if (l != k) {
#pragma xmp gmove
    a[k:n-1][l] = a[k:n-1][k];
#pragma xmp gmove
    a[k:n-1][k] = pvt_v[k:n-1];
  }
  ...
  for (j = kp1; j < n; j++) {
   t = pvt_v[j];
   A_daxpy(k+1, n-(k+1), t, a[k], a[j]);
  }
  ...
}
void A_daxpy(int b, int n, double da, double dx [N], double dy[N]) {
#pragma xmp align [i] with t(i) :: dx, dy
 ...
#pragma xmp loop on t(i)
 for (i = b; i < b+n; i ++)
  dy[i] = dy[i] + da*dx[i];
}
```

**図 7**　Linpack のソースコード
Fig. 7　Source code of Linpack.

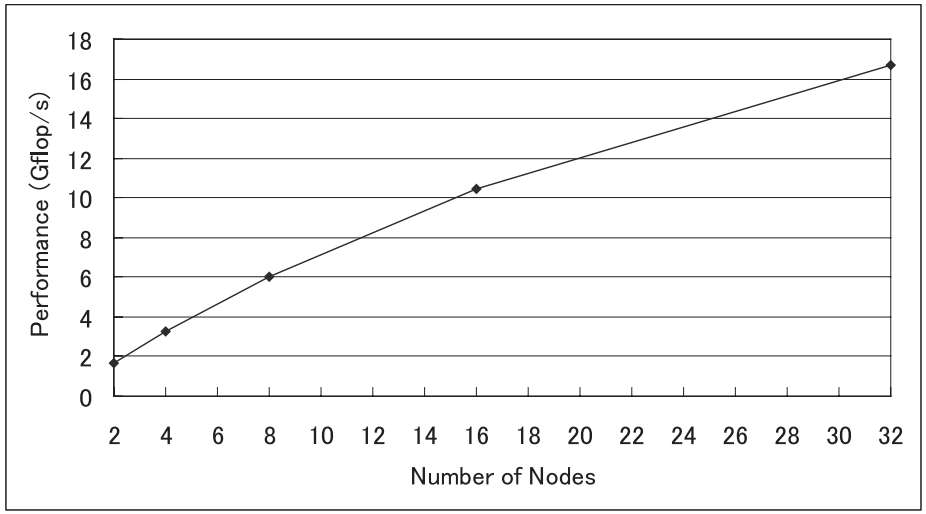


**図 8**　Linpack の評価結果
Fig. 8　Performance of Linpack.

とができる．Linpack の逐次コードの行数は 208 行である．並列化のために 35 行の指示文を逐次コードの中に記述した．

#### 6.2.2　評 価 結 果

**図 8** に Linpack の評価結果を示す．フラット MPI 実行による実行ノード内並列化を行っており，1 つの実行ノードにコア数と同じ 16 個の XMP ノードを生成，各コアに割り当てる．1 次元分割による単純な並列化であるため，Linpack の性能は最適とはいえない．より良い性能を達成するためには，2 次元分割によって並列度を引き出すなどの工夫が必要である．2 次元分割版のコードは現在の 1 次元分割版のコードから指示文を追加することで記述する．新しい次元に対してテンプレートの宣言や配列の分割，それにともなう通信の記述するなど，1 次元版でやってきた処理と同程度のコードを新しい次元に適用することで 2 次元分割のコードを得ることが可能である．

また，dgesl において行列の対角要素が計算中にブロードキャストされることで並列化の効率が下がっているという問題点がある．この問題に対しては計算の前に対角要素をローカルバッファに集めて通信を減らす工夫が必要と考えている．ローカルバッファの宣言，パック，アンパックのためのループ文の記述，MPI 関数の挿入は MPI で並列化を行う場合と同じプログラミングコストが求められる．しかし，それ以外の部分では XMP の指示文によって簡潔に並列化を記述できているため，性能が大きく影響するところで効率的にプログラミングコストをかけることができる．

### 6.3　FFT の並列化

FFT は高速フーリエ変換の性能を測定するためのベンチマークである．six-step FFT を用いて逐次版を記述し，並列化を行った．six-step FFT の並列化ではデータの転置にともなうノード間通信をどのようにして記述するかが問題となる．

#### 6.3.1　並列化の記述

**図 9** に XMP による FFT の並列コードの要点を示す．Linpack と同様，グローバルビューモデルに基づく並列化を行う．今回の評価では 1 つの次元のみに対し，block 分割を行った．six-step FFT では 2 次元の行列の各次元方向に対して 1 次元 FFT の処理を行う．その際，3 回の行列の転置が必要になるが，行列はノード間で分散されるため，通信が発生する．部分配列構文では次元の転置を直接記述することはできない．データ配列と異なる次元で分割される a_work を宣言し，それを経由することで転置を行う．まず，gmove 指示文と部分配列構文を用いることで転置に必要な配列のデータを a_work に収納する．次に，代入先の配列とローカルバッファの a_work でループ文による代入を実行し，転置を完了させる．配列の転置処理以外は独立した要素の処理が続くため，ループ文の実行を loop 指示文で並列化するだけで並列化を記述できる．FFT の逐次コードの行数は 186 行である．並列化のために 31 行の指示文を逐次コードの中に記述した．

#### 6.3.2　評 価 結 果

**図 10** に FFT の評価結果を示す．FFT は 1 個の XMP ノードを 1 台の実行ノードに割り当てる．Linpack と同様，少ないプログラミングコストで並列化を実現している．FFT のオーバヘッドのほとんどは配列の転置と，それにともなう通信によるものである．したがって，gmove 指示文の効率的な実装により性能が上がることが期待される．また，より高い性能を目指すときには gmove の記述からより明示的な通信の記述が必要である．たとえば，all-to-all コミュニケーションとローカルメモリでの転置をオーバラップさせることが考えられる．この場合，XMP の指示文のみで記述することはできず，ローカルビューモデルの CAF 記法や MPI 関数の利用が求められる．したがって，転置処理の記述コストは MPI 版とほぼ同等となる．しかし，MPI 版と比べて全体のプログラミングコストは低く，ユーザは行列の転置のみに集中することが可能である．

```
#pragma xmp nodes p(*)
#pragma xmp template t0(0:(N1*N2)-1)
#pragma xmp template t1(0:N1-1)
#pragma xmp template t2(0:N2-1)
#pragma xmp distribute (BLOCK) onto p :: t0, t1, t2
fftw_complex in[N1*N2], out[N1*N2], a_work[N2][N1];
#pragma xmp align [i] with t0(i) :: in, out
#pragma xmp align a_work[*][i] with t1(i)
...
int zfft1d(fftw_complex a[N], fftw_complex b[N], ...) {
#pragma xmp align [i] with t0(i) :: a, b
  ... zfft1d0((fftw_complex **)a, (fftw_complex **)b, ...); ...
}
...
void zfft1d0(fftw_complex a[N2][N1], fftw_complex b[N1][N2], ...) {
#pragma xmp align a[i][*] with t2(i)
#pragma xmp align b[i][*] with t1(i)
  ...
#pragma xmp gmove
  a_work[:][:] = a[:][:];
#pragma xmp loop on t1(i)
  for (i = 0; i < N1; i++)
    for (j = 0; j < N2; j++)
      c_assgn(b[i][j], a_work[j][i]);
#pragma xmp loop on t1(i)
  for(i = 0; i < N1; i++) HPCC_fft235(b[i], work, w2, N2, ip2);
  ...
}
```

**図 9**　FFT の並列化
Fig. 9　Source code of FFT.

### 6.4　評価結果の考察

今回の性能評価はプログラミングのしやすさに注目したものであったため，実用的な性能を得るには至らなかった．しかし，その性能はプログラミングコストを考えると妥当なものである．XMP のグローバルビューモデルでは指示文で記述された処理が，同じ処理を記述

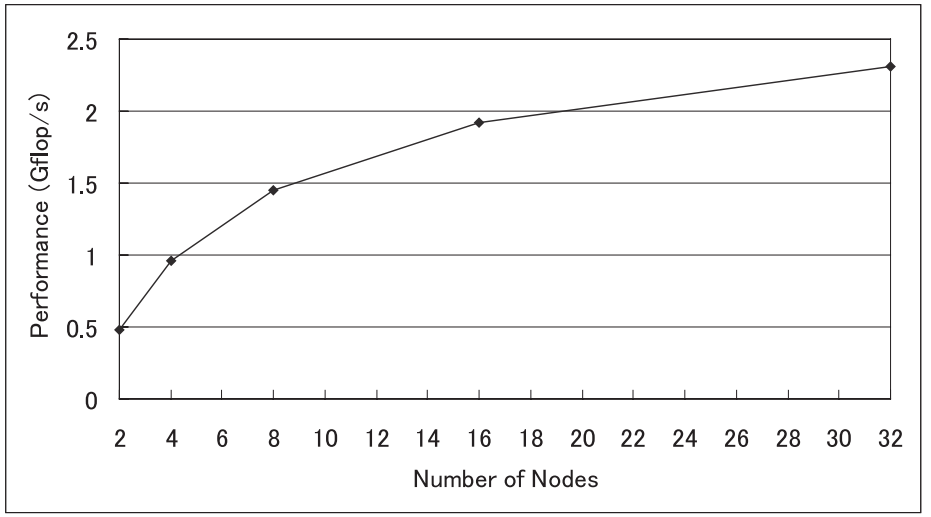


**図 10**　FFT の評価結果
Fig. 10　Performance of FFT.

した MPI コードと同程度のパフォーマンスを達成することを保証する．今回の結果は本来 MPI で記述しなければならなかった単純な一次元並列化をより少ないプログラミングコストで実現していることを示すものである．

XcalableMP を使うメリットとしてはインクリメンタルな並列化が可能であるということがあげられる．評価に用いた Linpack は 1 次元分割のみにとどまっているが，それに新しい指示文を追加することで多次元分割の並列化を行い，性能の向上を図ることができる．または，MPI 関数を直接呼び出すことで最適な通信を記述することも考えられる．自動的な並列化や最適化を提供しない言語の性質上，MPI に近い性能を出すためには MPI に近いコストをかけて並列アルゴリズムを記述しなければならない．しかし，多くのアプリケーションでは典型的な並列化をグローバルビューモデルの指示文で記述することになる．MPI や CAF 記法によるアルゴリズムの記述は性能に大きな影響を及ぼすところのみに集中され，ユーザはその性能のチューニングに集中することができる．

### 6.5　プログラミングコストと性能の考察

他の言語モデルによる並列化では言語機能に強く依存した並列コードの記述が必要である．UPC による並列化で高い性能を達成するためには shared メモリに対するアクセスを

減らし，なるべくローカルメモリで処理を行うことが求められる．したがって，shared メモリとローカルメモリの間のデータ移動が性能向上の鍵となる．HPCC Awards の過去のエントリ[3] からも shared メモリとローカルメモリ間のデータ移動関数を用いた並列化がなされていることが確認できる．このような場合，逐次コードから配列化を始めることは難しく，UPC の言語機能を熟知したうえで一から並列コードを作成しなければならない．これは並列プログラミングや UPC の初心者には大きな障害となり，経験者によってもプログラミングコストの面で負担が大きい．CAF ではこの問題はより顕著である．CAF で記述できるものは Co-Array によるデータ転送のみで，データの分散や処理の並列化はユーザは手動で行うものとしている．Linpack を CAF で並列化した文献 7) でも分かるように，転送するデータのローカル index やノード番号をユーザが計算し，通信を記述する必要があり，グローバルビューで指示文によって簡単に記述できるループ並列化に関しても手動で行われている．

逐次からの並列化に最も成功しているのは HPF である．文献 11) によると，逐次コードに指示文を挿入することで一部のアプリケーションに対しては少ないプログラミングコストで高い性能を達成している．しかし，性能が得られなかったアプリケーションに対して性能をチューニングする手段が提供されないということが HPF のかかえる問題点である．多くの研究では新しい指示文の導入による機能拡張により，ケースバイケースで当面の問題を解決しているが，自動並列化という言語設計で，すべてのパターンにつねに最適な通信を生成することができない現状では，根本的な問題点は解決できずにいる．

XMP の並列化はユーザが明示的に指定した場所でのみ行われる．グローバルビューモデルにおける指示文による通信は典型的かつ性能が十分に実用的であるとされるもの（リダクション，ブロードキャストなど）に限られる．したがって，一般的にノード間の定型的なデータ交換および通常の集団通信で記述される並列化であれば，MPI 版と比べて性能を落とすことなく，XMP の指示文による簡単な記述が可能であると考えられる．現在のコンパイラの実装はランタイムの通信レイヤとして MPI を利用しており，これらの通信は MPI_Allreduce() や MPI_Bcast() などの MPI 関数呼び出しに変換される．したがって，XMP が生成する並列コードは上記の MPI 関数を用いて直接作成したものと同等である．コンパイラによる余分な処理の追加がほとんど存在しないため，（MPI が記述したとおりの性能を示すのと同じように）記述した処理が予想されるとおりの性能を示す．これはユーザがパフォーマンスを予測，チューニングするために重要な要素である．XMP のローカルビューモデルを用いれば，MPI レベルでのより詳細なチューニングが可能である．その際は記述オーバヘッドが増えるものの，MPI と同じ程度まで性能を上げることが可能である．

## 7. マルチコアクラスタへの対応

マルチコアのプロセッサを持つクラスタシステムではプロセスレベルとスレッドレベルの並列化を組み合わせたハイブリッドプログラミングがより良い性能を引き出す可能性がある．また，大規模システムにおいて膨大になる MPI プロセスの数を減らす効果も期待できる．ここでは XMP によるスレッド並列化について検討を行い，マルチコアクラスタに対応した言語モデルを提案する．

### 7.1 コンパイラによるマルチスレッド化

まずはコンパイラによる自動的な並列化について考える．**図 11** に XMP のコードとコンパイラによって変換されたコードのイメージを示す．Pattern 1 の左辺のコードでは XMP の loop 指示文を用いてループ文の並列実行を記述している．loop 指示文によって並列化が指定されるループ文は各イテレーションの実行が独立であること，ローカルメモリのみにアクセスすることがユーザをユーザが保証する．このようなケースでは，プロセス内で実行されるイテレーションの集合を，スレッドを用いることで並列に処理することができる．したがって，Pattern 1 の左辺のコードを右辺のコードに変換することができる．右辺のコードではプロセスレベルでスケジュールしたイテレーション（lb:ub）をスレッドレベルで並列するため，OpenMP の指示文を挿入している（スレッド並列化のために用いる機構は OpenMP とは限らない，見やすさのため）．

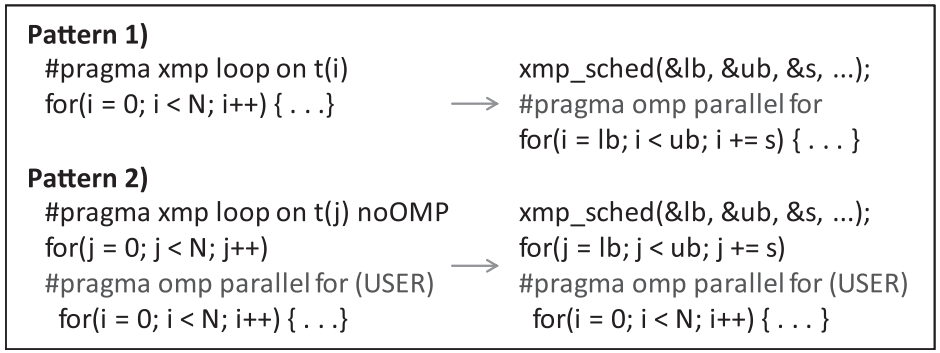


**図 11**　XcalableMP と OpenMP によるハイブリッド並列化
Fig. 11　Hybrid parallelization using XcalableMP and OpenMP.

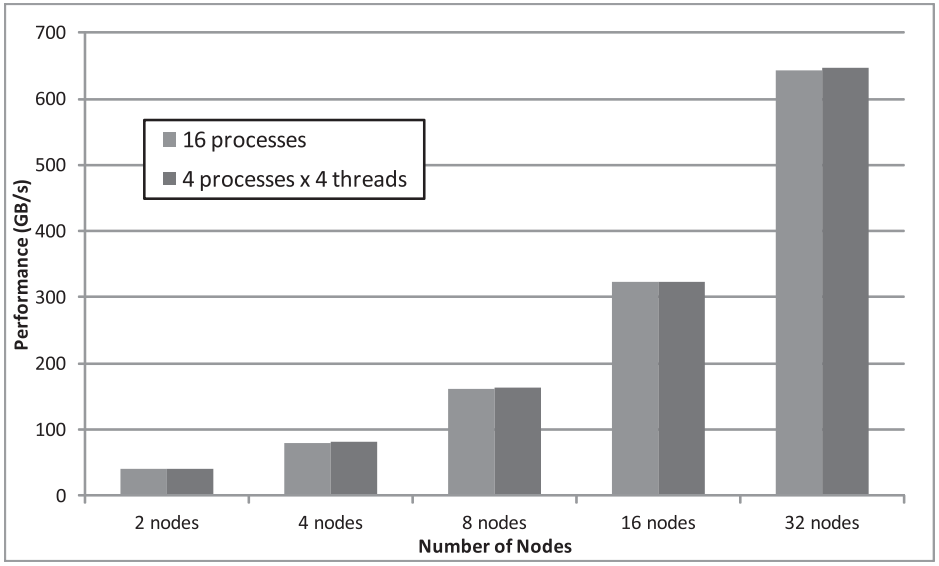


**図 12**　ハイブリッド並列化の予備評価結果
Fig. 12　Performance of hybrid-parallelized STREAM.

XMP は OpenMP-like な構文を持つため，OpenMP との親和性が高い．ハイブリッド並列化のために，ユーザが OpenMP の指示文を直接記述することも考えられる．Pattern 2 の右辺のコードでは外側のループ文を XMP で並列化し，内側のループ文を OpenMP で並列化している．ユーザが OpenMP のコードを直接記述する場合，コンパイラの自動並列化を抑制する機能が必要になる．loop 指示文の新しい節 noOMP はイテレーションがスレッドレベルでスケジュールされることを抑制する．

### 7.2 予 備 評 価

HPC Challenge Benchmark の STREAM を用いて，ハイブリッド並列化の予備評価を行う．STREAM は 1 次元のデータ配列をループ文でアクセスする単純な構造のプログラムである．並列化は図 11 の Pattern 1 のように loop 指示文の挿入で記述する．予備評価ではコンパイラによる自動並列化の性能について検討する．XMP のランタイムライブラリと GCC OpenMP を用いて，想定される並列コードの変換を手動で行った．

**図 12** にフラット MPI とハイブリッド並列化による評価結果を示す．評価に用いた T2K-Tsukuba システムは Quad-core の 4 ソケット構成であるため，ハイブリッド並列化では各ソケットに 1 つの MPI プロセスを割り当て，ソケットの中でスレッド並列化を行った．評価の結果ではハイブリッド並列化とフラット MPI で同程度の性能を示している．これは STREAM がほとんど通信を行わずローカルメモリのみにアクセスするため，当然の結果であり，ハイブリッド並列化によって性能が低下しないことを表す．

## 8. お わ り に

本論文では PC クラスタなど分散メモリ環境のための並列プログラミングモデルとして C や Fortran 言語の並列拡張である XMP が提案されている．XMP は 2 つのプログラミングモデルを同一の言語で併用できるようにして，プログラミングコストと記述力を両立させている．

グローバルビューモデルでは典型的な並列化手法を OpenMP-like な指示文で記述する．MPI と比べてプログラミングコストが少ないというメリットがあり，逐次コードからのシームレスな並列化を行うことができる．HPC Challenge Benchmark の Linpack，FFT を用いた性能評価では，少ないプログラミングコストで並列化を記述できることが分かった．

グローバルビューモデルで十分な性能を達成できないアプリケーションに対してはローカルビューモデルを利用することができる．CAF-like な記述により，MPI と同等に自由度が高く，より直感的な通信の記述が可能である．

今後はローカルビューモデルによる最適化を行ったベンチマークの評価を行う．また，今回手動で変換を行った gmove 指示文など，コンパイラの実装が完了していない部分を完成させる．最後に，現在 HPC プラットフォームとして標準的な環境であるマルチコアクラスタの性能を十分に引き出すため，コンパイラによるマルチスレッド化の検討を深め，実装を行う．

**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．XcalableMP の仕様は，検討している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. http://www.xcalablemp.org/
2) Partitioned Global Address Space. http://www.pgas-forum.org/

3) HPC Challenge Benchmark. http://icl.cs.utk.edu/hpcc/
4) High Performance Fortran 言語仕様書 Version 2.0. http://www.hpfpc.org/jahpf/spec/hpf-v20-j10.pdf
5) UPC Language Specifications V1.2. http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf
6) Numrich, R.W. and Reid, J.: Co-array Fortran for parallel programming, *ACM SIGPLAN Fortran Forum*, Vol.17, Issue 2, pp.1–31 (1998).
7) Reid, J.K., Rasmussen, J.M. and Hansen, P.C.: The LINPACK Benchmark in Co-Array Fortran. http://www2.imm.dtu.dk/documents/ftp/tr00/tr14_00.pdf
8) 岩下英俊，進藤達也，岡田　信：VPP Fortran：分散メモリ型並列計算機言語，情報処理学会論文誌，Vol.36, No.7, pp.1542–1550 (1995).
9) Lee, J., Sato, M. and Boku, T.: OpenMPD: A Directive Based Data Parallel Language Extensions for Distributed Memory Systems, *The 37th International Conference on Parallel Processing* (*ICPP08*), pp.121–128 (2008).
10) 高橋大介，後藤和茂，朴　泰祐，建部修見，佐藤三久，三上和徳：T2K 筑波システムにおける Linpack 性能評価，情報処理学会研究報告，Vol.2008-HPC116，pp.55–60 (2008).
11) 太田　寛，西谷康仁，小林　篤，布広永示：HPF 処理系 Parallel FORTRAN による NAS Parallel ベンチマークの並列化，情報処理学会論文誌，Vol.38, No.9, pp.1830–1839 (1997).
12) 蒲池恒彦，草野和寛，末広謙二，妹尾義樹，田村正典，左近彰一：HPF 処理系の実現と評価，情報処理学会論文誌，Vol.37, No.7, pp.1255–1264 (1996).
13) 村井　均，岡部寿男：地球シミュレータ上の HPF による NAS Parallel Benchmarks の実装と評価，*Proc. SACSIS2004*, pp.389–396 (2004).
14) 岩下英俊，青木正樹：HPF トランスレータ fhpf における分散種別を一般化したコード生成手法，情報処理学会論文誌コンピューティングシステム（ACS15），pp.329–339 (2006).
15) 太田　寛，西谷康仁：データ並列言語における多重ループの計算分散方式，並列処理シンポジウム JSPP99，pp.79–86 (1999).
16) 太田　寛，西谷康仁：データ並列言語の通信生成方式とマルチグリッド法での最適化評価，情報処理学会論文誌，Vol.42, No.4, pp.868–878 (2001).
17) Gupta, S.K.S., Kaushik, S.D., Huang, C.-H. and Sadayappan, P.: Compiling array expressions for efficient execution on distributed-memory machines, *Journal of Parallel and Distributed Computing*, Vol.32, pp.155–172 (1996).

(平成 22 年 1 月 26 日受付)
(平成 22 年 4 月 30 日採録)

**李　　珍泌**（学生会員）

昭和 58 年生．平成 19 年筑波大学第 3 学群情報学類卒業．平成 21 年同大学大学院システム情報工学研究科コンピュータサイエンス専攻博士前期課程修了．現在，同大学院システム情報工学研究科コンピュータサイエンス専攻博士後期課程在学中．分散メモリ向け並列プログラミング言語に関する研究に従事．

**朴　　泰祐**（正会員）

昭和 35 年生．昭和 59 年慶應義塾大学工学部電気工学科卒業．平成 2 年同大学大学院理工学研究科電気工学専攻後期博士課程修了．工学博士．昭和 63 年慶應義塾大学理工学部物理学科助手．平成 4 年筑波大学電子・情報工学系講師，平成 7 年同助教授，平成 16 年同大学大学院システム情報工学系助教授，平成 17 年同教授，現在に至る．超並列計算機アーキテクチャ，ハイパフォーマンスコンピューティング，クラスタコンピューティング，グリッドに関する研究に従事．平成 14 年度および平成 15 年度情報処理学会論文賞受賞．日本応用数理学会，IEEECS 各会員．

**佐藤　三久**（正会員）

昭和 34 年生．昭和 57 年東京大学理学部情報科学科卒業．昭和 61 年同大学大学院理学系研究科博士課程中退．同年新技術事業団後藤磁束量子情報プロジェクトに参加．平成 3 年通産省電子技術総合研究所入所．平成 8 年新情報処理開発機構並列分散システムパフォーマンス研究室室長．平成 13 年より筑波大学システム情報工学研究科教授．平成 19 年より同大学計算科学研究センターセンター長．理学博士．並列処理アーキテクチャ，言語およびコンパイラ，計算機性能評価技術，グリッドコンピューティング等の研究に従事．IEEE，日本応用数理学会会員．