PGAS モデルに基づく大規模並列クラスタ向け高生産並列プログラミングモデルに関する研究

2018 年 3 月

津金 佳祐

PGAS モデルに基づく大規模並列クラスタ向け高生産並列プログラミングモデルに関する研究

津金 佳祐

システム情報工学研究科

筑波大学

2018 年 3 月

概要

高性能計算の分野では，エクサスケールに向けて計算機環境の大規模化や，使用可能な電力量の制限から消費電力性能比の良いメニーコアプロセッサの採用が進んでいる．計算機環境は複雑さを増す一方で，ユーザに求められるプログラミングコストも増加しており，高性能かつ高生産な並列プログラミングモデルが必要とされる．分散メモリ環境での並列プログラミングモデルは MPI が普及しているが，その記述の複雑さから生産性の低さが問題とされている．近年では，Partitioned Global Address Space（PGAS）モデルが登場し，分散メモリ環境における大域的な名前空間による簡易な並列処理や通信記述を可能としている．PGAS モデルはグローバルビューとローカルビューの 2 種類のプログラミングモデルに分類される．グローバルビューは，簡易な指示による並列実装を可能とするモデルであり，逐次の実装に対して指示を挿入するのみでの並列化を実現する．また，実装されたプログラムは逐次実装のイメージを維持しており，アプリケーションの生産性を高めることが可能である．しかし，プログラミングの制約も多く典型的な通信や演算にしか適用できない場合が多い．一方で，ローカルビューでは，簡易な記述による片側通信と各ノード固有の名前空間を用いたプログラミングにより多くのプログラムに適用可能である．しかし，データ分散や並列実装のための記述は MPI と同様陽に記述する必要があり，プログラム全体での生産性が高いとは言えない．また，近年注目を集めているメニーコアプロセッサの代表例である Intel Xeon Phi では，OpenMP のループ並列によるワークシェアリングを用いた実装が主流である．しかし，コア数の増加によりロードインバランスが発生しやすく，ワークシェアリングが内包する全体同期のコストが増加するなどの性能低下が問題となっている．OpenMP では仕様 4.0 よりデータ依存によるタスク並列モデルを提供しており，データ依存によるタスク間の細粒度な同期により，大量のコアを持つメニーコアな環境において性能向上が期待される．しかし，OpenMP は単一ノードを対象としており，分散メモリ環境におけるノードを跨る依存関係の簡易な記述方法が求められる．そこで，本研究では大規模並列クラスタにおけるプログラムの生産性を向上させるため，PGAS 言語 XcalableMP（XMP）を対象とし，PGAS モデルのグローバルビューとローカルビューを組み合わせたハイブリッドビューを提案する．提案モデルで核融合シミュレーションコード GTC-P を実装し，オリジナルの MPI 実装と比較をすることで，提案モデルによる実装の性能と生産性を示す．また，ハイブリッドビューはノード間での通信，データ分散や並列実行を簡易に記述可能とするモデルであり，メニーコアシステムではノード間並列に加えて，ノード内並列の性能，生産性も考慮に入れる必要がある．そこで，メニーコアシステム上での高性能，高生産を両立させるプログラミングを可能とすべく，PGAS モデルによるタスク並列プログラミングモデルの提案を行う．XMP を対象にノード内/間を含むタスク依存に基づく並列実行を可能とする指示文を提案し，提案指示文によるベンチマークプログラムを用いてメニーコアシステム上で評価をすることで，提案モデルによ

る実装の性能，生産性を示す．ハイブリッドビューによる GTC-P の実装では，グローバルビューによる簡易記述で逐次実装を維持しつつ，ローカルビューによるスケーラブルな通信記述により，オリジナル MPI 実装に近い性能で高い生産性を示した．一方，PGAS モデルに基づくタスク並列モデルは，ノード内/間を含むタスク依存を指示文による簡易な記法で実現可能とした．また，従来のループ並列による実装との比較では，XMP によるタスク並列による実装の高い性能と生産性を示した．

目次

図目次

表目次

ソースコード目次

第 1 章

序論

1.1 研究背景

計算科学の分野では，気象，宇宙物理，量子物性や素粒子物理など様々な分野において，シミュレーションや実験データの解析にスーパーコンピュータが用いられており，処理時間の短縮や大規模な問題を解くために多くの計算リソースに加えて高い演算性能が必要とされる．そうした要求から，High Performance Computing（HPC）の分野では，2020 年あたりを目標に 1ExaFLOPS に向けた大規模な計算機環境の開発が進められている．スーパーコンピュータの性能ランキングである Top500[1] より現在稼働中のスーパーコンピュータを見ると，その上位は数千ノードを超える大規模なシステムが占める．しかし，1 システムあたりが使用可能な電力量には制限があるため，今後も単純なノード数の増加による高性能化の実現は困難である．そのような問題から近年では，1 プロセッサあたりの消費電力性能比が良いメニーコアプロセッサが注目を集めている．Intel Xeon Phi や PEZY-SC，申威 26010 など様々な種類のメニーコアプロセッサが開発されており，性能や消費電力要求により今後もメニーコアを搭載するシステムは増加すると考えられる．しかし，1 ノードに数十，数百コアを持つプロセッサを搭載し数万ノード規模で構成されるなど計算機環境の複雑化は進み，アプリケーション開発のためのユーザに求められるプログラミングコストは増加傾向にある．従って，大規模な分散メモリ環境やメニーコアプロセッサを搭載したシステムに向けた高性能，高生産な並列プログラミングモデルが求められる．

1.2 分散メモリ環境における並列プログラミングモデル

分散メモリ環境の並列プログラミングモデルは，Message Passing Interface（MPI）がデファクトスタンダードとなりつつある．しかし，MPI はプロセス毎のデータの分散配置や複雑な通信記述など，並列化のための様々な処理手順を全て明示的に記述する必要があり，プログラミングの学習コストが高くソースコードが煩雑になりやすいという生産性の低下が大きな問題とされている．そこで，分散メモリ環境上での並列プログラミングをより容易にするために提案，開発が進められているのが Partitioned Global Address Space（PGAS）モデル [2] である．PGAS モデルは，分散メモリ環境において各プロセスに分割されたデータを単一の名前空間としてユーザに提供することにより，プロセス間通信や並列処理を容易に記述可能とするモデルである．ノード全体を参照可能な大域的な名前空間を用いたプログラミングとなる

ため，共有メモリ向けのプログラミングをするように，各ノードの局所性を意識した実装が可能となる．PGAS モデルの特性上，グローバルビューとローカルビューの 2 種類のプログラミングモデルに分類される．計算科学の分野におけるアプリケーションの大半は C や Fortran により開発が進められているため，本節では C，Fortran をベースとした PGAS モデルを例に挙げ，2 種類のプログラミングモデルとその生産性に関して述べる．

1.2.1 Partitioned Global Address Space（PGAS）モデル

Unified Parallel C（UPC）[3]/UPC++[4]，Chapel[5]，Global Arrays（GA）[6]，X10[7]，High Performance Fortran（HPF）[8] 及び XcalableMP（XMP）[9, 10, 11] などを代表とするグローバルビューは，各プロセスに分割されたデータを大域的な名前空間にマップし，それを用いて並列プログラミングを行うモデルである．ユーザからは実行プロセス数やデータ分割方法などを指示することで，逐次実装を維持したままでデータ分散や並列実行，モデルによっては暗黙の通信までも実現する．

グローバルビューの UPC を例にプログラミングモデルの詳細を示す．UPC は C の言語拡張により分散メモリ環境へと対応した PGAS 言語である．*thread* を実行単位としており，MPI と OpenMP の組み合わせのようにノード内/外でプロセス/スレッドと，別モデルによる異なる実行単位で記述するのではなく，システム全体で統一的に扱う．静的に記述された配列を *thread* 間で分散する場合は，定義時に `shared` 修飾子を指定するのみであり，`private` 修飾子か何も記述がなければ *thread* 固有の定義となる．`shared` 修飾子が記述された配列は大域的な名前空間上に配置され，別 *thread* が持つ領域も含めて共有メモリを扱うようにアクセスすることが可能である．データ分散の形状は基本的にストライド分割が適用されるが，`shared` 修飾子とデータ型の間に *[]* を記述し値を指定することで，指定サイズのブロックサイクリック分割となり，“*” を指定することでブロック分割となる．分散された配列は，自 *thread* が持たない値の参照や更新も可能であり，ノード内であれば単純なメモリの read/write，ノード間の場合は暗黙の通信がランタイムによって生成，実行される．また，UPC では *upc_memcpy()* のように大域的な名前空間に基づく明示的な通信も記述可能である．ループの並列実行には，逐次実装の for ループが演算する範囲を維持したまま *upc_forall* と書き換え *affinity* を指定する．*affinity* に記述された条件と自 *thread* が所有する範囲が合致する箇所を実行するため，ループ並列によるワークシェアリングが実行可能である．

Coarray Fortran（CAF）[12]，OpenSHMEM[13] 及び Advanced Communication Primitives（ACP）[14] などを代表とするローカルビューでは，各ノードが持つローカルな名前空間による MPI と同様のデータ分散や並列実行の記述が求められるが，ノード間の通信は片側通信を採用しており，他ノードが持つ固有の名前空間に対して直接メモリの read/write（Get/Put）を行うことが可能である．また，MPI の通信記述と比較してよりスケーラブルな記述を提供するモデルが多い．

ローカルビューの CAF を例にプログラミングモデルの詳細を示す．CAF は Fortran の言語拡張であり，配列代入文形式による片側通信機能（Put，Get）を提供する．実行単位はイメージであり MPI のプロセスと同様に扱われる．Fortran の文法に *[]* の概念が追加されており，片側通信における対象の配列やイメージの指定に用いられる．リモートイメージから Put や Get する配列は，定義時に *[]* で実行するイメージ集合を指定する．また，1 次元で全実行イメージを対象とする場合は “*” を指定する．片側通信は，配列代入文に対して *[]* を追加し，イメージ番号の記述が必要となる．右辺にイメージ番号の指定が

あれば Put，左辺であれば Get が記述されたイメージに対して実行される．通信するデータサイズはコロンを用いたセクションで記述され，ユーザは範囲を指定する．また，MPI のように通信記述にタグ，データ型やコミュニケータなどの記述は必要なく，全てランタイムが自動的に決定し通信の整合性をとる．そのため，逐次の配列代入文とほぼ同等の記述で片側通信が記述可能である．

1.2.2 生産性と問題点

計算科学の分野におけるプログラムの生産性は，アプリケーションの開発時間，性能チューニングやデバッグのしやすさ，コード拡張や再利用性，並列プログラミングモデルの学習コストなど様々な要因から評価をする必要がある．本節では，それらの観点から PGAS モデルの生産性について述べる．PGAS モデルや MPI などを用いて並列プログラムを実装する場合，その多くは逐次のプログラムを実装し，そのコードを基に並列化を行う．そのため，グローバルビューによる実装は，大域的な名前空間によるプログラミングを可能とする簡易な指示による実装であるため，逐次プログラムから段階的，部分的な並列実装が可能であり，デバッグの容易さによるアプリケーションの開発時間の短縮が可能であると言える．また，実装された並列プログラムは逐次の記述とほぼ同等であるため，並列実装後であってもコード拡張や再利用が容易である点も利点と言える．さらに，簡易な指示による実装であるため，プログラミングモデル自体の学習コストも低い．ローカルビューによる実装では，各ノード固有の名前空間によるプログラミングにより，既に MPI 実装されたアプリケーションも含む多くの実装に対して適用が可能である．さらに片側通信は，通信バッファへのコピーや双方向での同期を排除した，よりハードウェアレベルに近い通信としているため，P2P 通信と比較して高速な通信を実現可能な環境が多い．また，CAF のような配列代入文による記述をサポートしているなど，MPI と比較してより可読性の高い記述を提供しており，ユーザのプログラミングミスを減らす．

PGAS モデルには生産性に関する多くの利点がある一方で，プログラミングの制約や問題も多い．グローバルビューでは，XMP による格子 QCD の実装 [15, 16] や HPF による数値流体力学のシミュレーションコード [17]，UPC++ のマルチグリッド法への適用 [18] など，様々なアプリケーションが開発されている．しかし，その多くは典型的なステンシル問題が主な演算であり，通信も隣接ノード間の境界要素同士の通信である袖領域通信を主とする，比較的単純なアルゴリズムへの適用である．グローバルビューの特性上，データ分散は各 PGAS モデルが提供する典型的な分散形状に限られるため，例えば，NAS Parallel Benchmarks（NPB）の Block Tri-diagonal solver（BT）や Scalar Penta-diagonal solver（SP）で用いられる Multipartitioning 法による分割 [17] のように，ロードバランス改善のための不均一で複雑な分散の場合に適用できず，同一アプリケーションであってもグローバルビュー向けのデータ分散アルゴリズムによる実装が求められる．またデータ分散の記述以外にも，全ての通信が大域的な名前空間上で効率的に記述できるわけではない．例えば，通信対象の非連続領域を分割し，部分的なデータパッキングと片側通信をパイプライン的に実行する実装 [19] や， 部分的な集合通信の実装 [20] など，大域的な名前空間上で通信最適化を適用することは難しい．また，暗黙の通信の場合は，並列プログラムを簡易に実装することは可能となるが，ユーザからはプログラムのどの箇所で通信が実行されているかを把握するのは難しく，性能チューニングを困難にする．一方でローカルビューは，片側通信による通信性能の向上を目的としたアプリケーションの開発が進められている．例えば，OpenSHMEM による Graph500 ベンチマー

クの実装 [21]，CAF による地下のイメージング手法であるリバースタイムマイグレーションの実装 [22] など様々ある．通信記述は，MPI と比較して簡易な記述で実装されており，P2P 通信と比較して片側通信による高速化を達成している実装も多い．しかし，データ分散や並列実行は依然として MPI と同様に全て明示的に記述する必要があり，プログラム全体の生産性が高いとは言えない．以上のように各モデルによる問題点も多いため，ユーザにとってより性能や生産性を高めることが容易なプログラミングモデルが求められる．そこで本研究では，グローバルビューとローカルビューの両方を提供する XMP を用いて，グローバルビューの簡易な記述によるデータ分散や並列実行及び通信記述を残しつつ，通信最適化のような複雑な記述を必要とする通信実装にローカルビューを用いる，ハイブリッドなプログラミングを可能とする新たなモデルを提案する．

1.3 メニーコア環境における並列プログラミングモデル

PGAS モデルのプログラミングモデルの改善を行うことで，分散メモリ環境でのプログラムの生産性を向上させることは可能だが，近年登場しているメニーコアプロセッサを搭載するシステムにおいてはノード間並列に加えて，ノード内並列の性能，生産性も考慮に入れる必要がある．メニーコアプロセッサとしては，Top500 リストより Intel Xeon Phi が注目を集めていることがわかる．

Xeon Phi におけるプログラミングには，OpenMP のループ並列によるワークシェアリングを用いた実装が多く行われており，その多くがループの並列実行終了後に全体同期を必要とする．メニーコアプロセッサでは，コア数増加により意図しないロードインバランスが発生しやすく，全体同期のコストが大幅に増加する可能性がある．また，同期自体のコストも大きい [23] ため，ユーザは出来る限り全体同期をしない並列プログラミングが求められる．

OpenMP では仕様 3.0 よりタスク並列が記述可能となり，再帰的構造や while ループなどの各スレッドでの演算が動的に決定する場合の並列実装が可能となった．また，仕様 4.0 からはタスク依存が記述可能となり，依存関係によるタスク間での一対一同期とすることができる．タスク間のデータ依存を記述することで，従来のワークシェアリングによる全体同期を排除したタスク単位での細粒度な同期を実現し，ロードバランスの改善による性能向上が期待される．本節では，Intel Xeon Phi プロセッサである Knights Landing（KNL）[24] の概要を示す．また，メニーコアプロセッサ上での並列プログラミング手法に関してまとめ，現状のメニーコア環境におけるプログラミングの問題点を示す．

1.3.1 Knights Landing（KNL）メニーコアプロセッサ

KNL は Intel Xeon Phi の第 2 世代プロセッサであり，Knights Corner（KNC）の後継機である．KNC は GPU と同様に PCI Express 接続のアクセラレータ型で，Intel Xeon などのホストプロセッサを必要とするが，KNL は自身がブート可能となったため KNL のみをホストプロセッサとするシステムを構築可能となった．また，高バンド幅な 3D 積層メモリである MCDRAM の搭載や AVX512 命令のサポート，さらにコア間はリングバス接続から 2 次元メッシュネットワークになるなど，KNC から多くの変更点がある．

図 1.1 に KNL のチップ内構成を示す．KNL は 2 コアを 1 タイルとして扱う．タイルは 1MB の L2

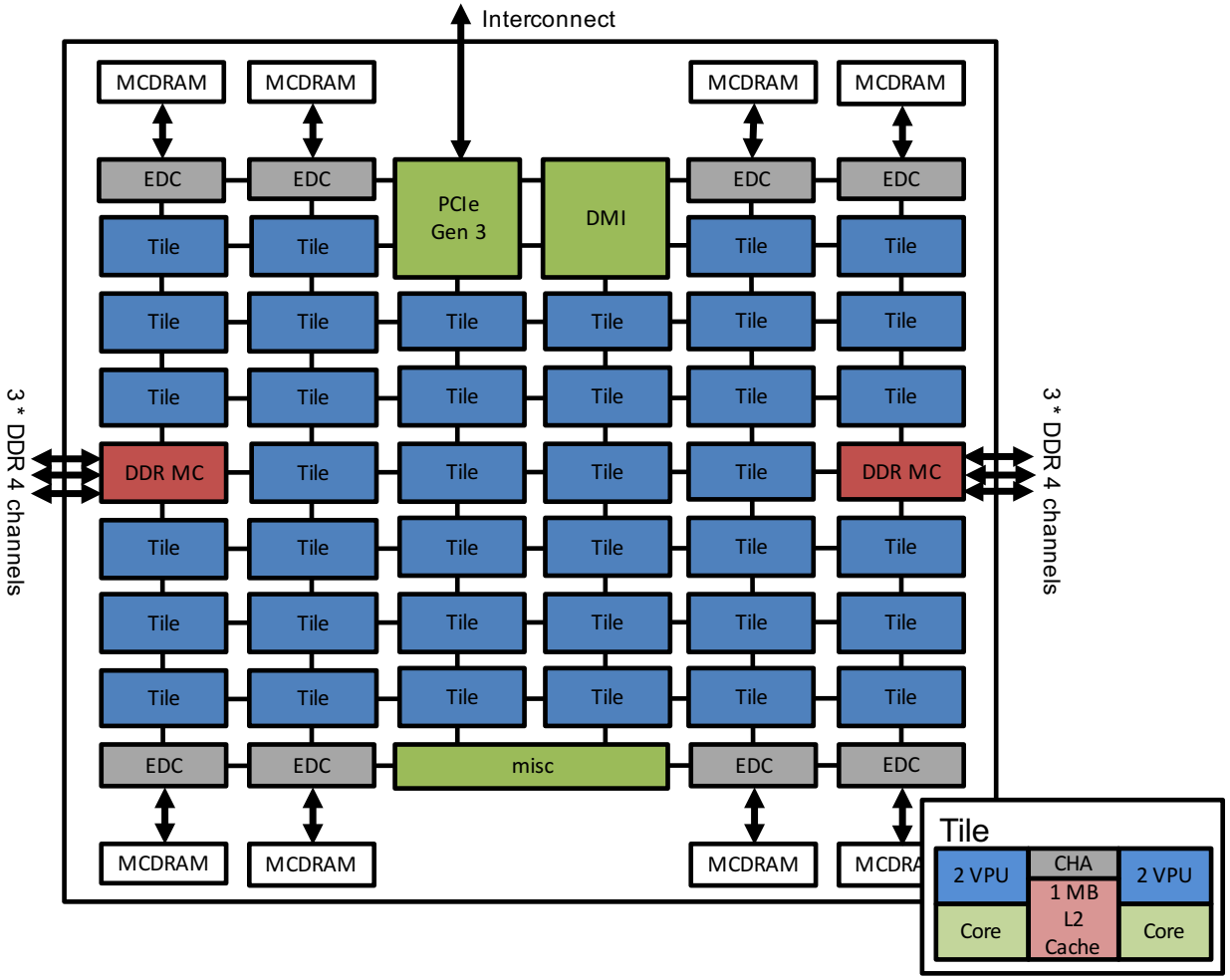


図 1.1 KNL（Knights Landing）プロセッサの構成.

キャッシュとコア毎に 2 つの Vector Processing Unit（VPU）で構成され，タイル間は 2 次元メッシュネットワークで接続される．各コアはハイパースレッディングにより最大 4 スレッドまで動作することから，Oakforest-PACS，Cori システムなどで用いられている Intel Xeon Phi 7250 では 34 タイル，68 コアで最大 272 スレッドによる実行が可能である．MCDRAM は 2GB ずつ合計で 16GB がチップ内に搭載され，チップ両端には DDR4 メモリチャネルが 3 つずつ合計 6 チャネルある．

KNL は 2 次元メッシュネットワークの扱いに All-to-All，Quadrant，Sub-NUMA Clustering（SNC4）の 3 種類のクラスタリングモードを提供している．All-to-All モードは，チップ内の全てのコアを単一プロセッサとして扱うため，メモリアクセスによってはコアとメモリ間のパスが長くなる．Quadrant モードは，All-to-All モードと同様に単一プロセッサとしてコアを扱うが，ネットワークは仮想的に 4 分割され各領域に最も近いメモリにデータが配置される．従って，ユーザはデータのメモリ配置を意識することなくプログラミングすることが可能である．SNC4 モードはチップを仮想的に 4 分割し，4 ソケットの Xeon プロセッサのように 4 つの NUMA ノードとして扱う．基本的に分割された各領域に閉じたメモリアクセスとなるため，numactl コマンドで NUMA ノードを扱う手間はあるが高い性能が期待される．

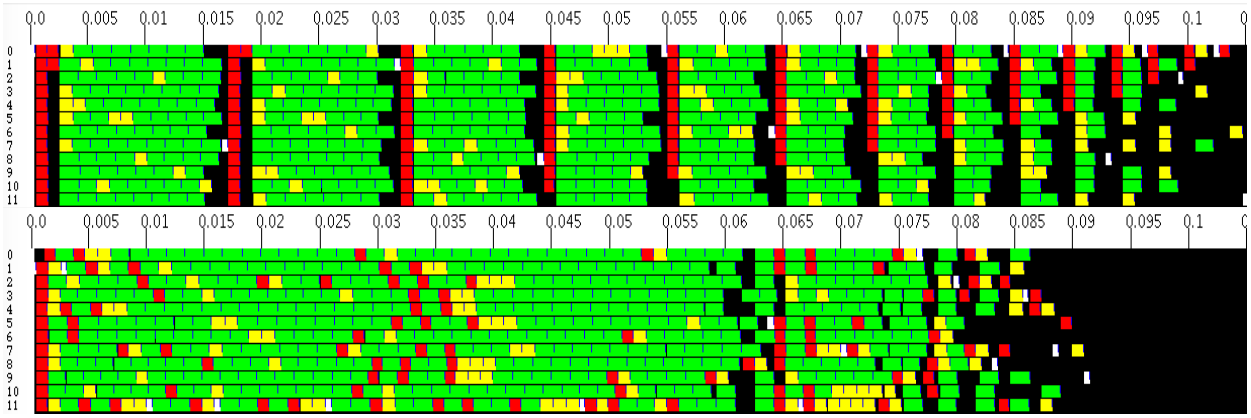


図 1.2　tlog によるブロックコレスキー分解のタイムライン．上はループ並列，下はタスク並列による実装を表し，12 スレッドによる実行を示す．

さらに KNL は，MCDRAM と DDR4 メモリの扱いのために 3 種類のメモリモード（Flat，Cache 及び Hybrid）を提供している．Flat モードは，MCDRAM と DDR4 メモリが異なるメモリアドレス上に配置され，numactl コマンドなどにより明示的に割り当てる．Cache モードは，MCDRAM を DDR4 メモリの L3 キャッシュのように扱う．そのため，明示的に MCDRAM を利用することはできないが，MCDRAM のメモリ量 16GB を超えるプログラムであっても，プログラムに変更を加えることなく実行が可能である．Hybrid モードは，MCDRAM を分割し Flat，Cache モードのそれぞれを利用可能とする．

1.3.2　ループ並列とタスク並列

メニーコアプロセッサにおいて高い性能を得るためには，その莫大なコアを使い切ることが可能な高い並列性や，適切なロードバランシングによりアイドルスレッドを発生させないことが，マルチコアプロセッサ以上に重要とされる．Xeon Phi プロセッサ（KNL）のプログラミングには，OpenMP の `parallel for`，`for` 指示文を用いたループ並列によるワークシェアリングの実装が多く行われており，その多くはループ並列実行終了後にスレッド全体の同期を必要とする．メニーコアプロセッサでは，コア数増加による意図しないロードインバランスが発生しやすく，全体同期のコストが大幅に増加する可能性がある．また，同期自体のコストも大きく [23]，ユーザには出来る限り全体同期をしない並列プログラミングが求められる．

OpenMP では，仕様 3.0 より `task` 指示文でタスク並列が記述可能となり，再帰的構造や while ループなどの各スレッドでの演算が動的に決定する場合の並列実装が可能となった．また，仕様 4.0 からは `depend` 節によるタスク依存の記述を提供し，依存関係によるタスク間での一対一同期が可能となった．そこで，ループ並列とタスク並列のそれぞれを用いて同一のベンチマークプログラムを実装し，同期方法の違いが性能に与える影響を調査する．図 1.2 に OpenMP の `parallel for` 指示文によるループ並列実装と，`task` 指示文と `depend` 節によるタスク並列実装のタイムラインを示す．対象はブロックコレスキー分解であり，1 ノードに Intel Xeon E5-2680 v2 を 2 ソケット（10 コア ×2）持つ環境において 12 スレッドで実行する．コンパイラは Intel Compiler 17.0.1 とし，タイムラインの表示にはプロファイリン

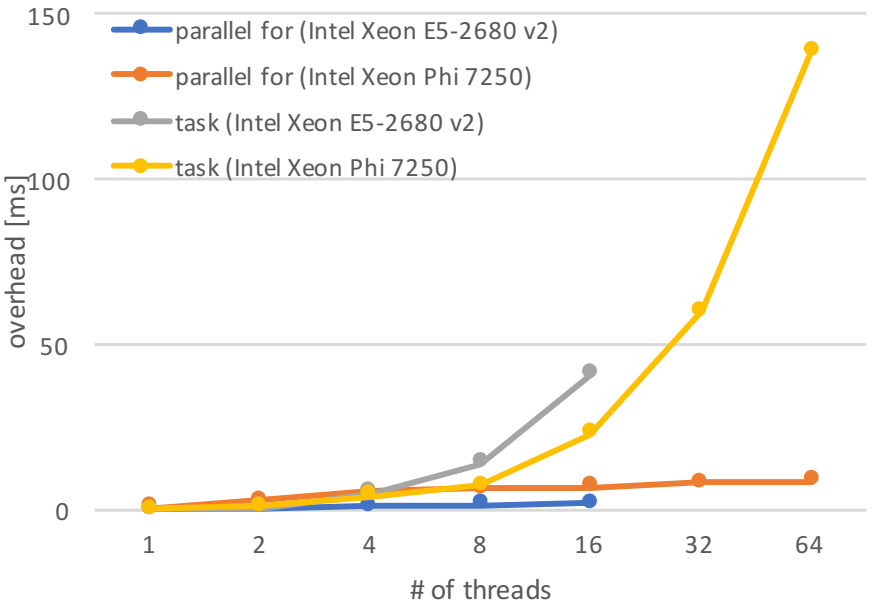


図 1.3 OpenMP `parallel for`，`task` 指示文のオーバヘッド．

グツールである tlog を用いる．縦軸がスレッド数，横軸が時間を表しており，黒い領域はスレッドがアイドル状態であることを表し，その他の領域は演算が実行されている区間を表す．

ループ並列による実行では，0.015，0.03s などにおいて全体同期により多くのスレッドがアイドル状態であることがわかる．一方で，タスク並列では全体同期から一対一同期となったことでアイドル状態のスレッドが減り，性能が向上していることがわかる．よって，タスク間のデータ依存を記述しタスク単位の細粒度な同期とすることで，従来のワークシェアリングによる全体同期を排除し，ロードバランスの改善による性能向上が期待される．しかし，OpenMP はノード内記述のみをサポートしているため，分散メモリ環境でのタスク依存が記述可能なプログラミングモデルが必要とされる．

OpenMP の仕様 4.0 以上をサポートするコンパイラは GNU や Intel など様々あるが，タスク並列に関する実装は未だ改善の余地がある．図 1.3 に EPCC OpenMP micro-benchmark suite[25] による OpenMP の `parallel for`，`task` 指示文のオーバヘッドを示す．コンパイラには Intel Compiler 17.0.1 を使用し，Intel Xeon E5-2680 v2 と Intel Xeon Phi 7250 の 2 種類の環境を用いて，マルチ・メニーコアプロセッサでの傾向の違いを示す．まず，`parallel for` 指示文のマルチ・メニーコアプロセッサにおける性能を比較すると，どちらのプロセッサにおいてもスレッド数増加とともに性能が緩やかに低下し，全体としてはメニーコアプロセッサでの性能が低いことがわかる．次に `task` 指示文の性能は，16 スレッドまではマルチコアプロセッサでのオーバヘッドが大きく，スレッド数の増加とともにメニーコアプロセッサでの性能が極端に低下した．また，マルチ・メニーコアプロセッサのどちらにおいても `parallel for` 指示文と比較して `task` 指示文のオーバヘッドが大きいことがわかった．

タスク実行中にデッドロックが起きうる箇所で，別の実行可能タスクへとスイッチを促す `taskyield` 指示文の挙動の調査も行ったが，GNU（6.4），Intel（17.0.1）コンパイラともに依存付きタスクの場合の

挙動が不定であった．分散メモリ環境でのタスク並列実行では，タスク内で通信を実行することが考えられる．そのため，デッドロックの防止やスレッド内での通信と演算タスクのオーバラップのためにも `taskyield` 指示文によるタスクのスイッチは必要である．そこで，タスク生成やコンテキストスイッチのコストが比較的小さい軽量スレッドライブラリを用いて，高速なタスク生成や一定動作するタスクスイッチング機能を持つタスク並列の実装が必要とされる．

1.3.3 通信

分散メモリ環境における性能向上の手法の一つとして，通信と演算をオーバラップすることが挙げられる．MPI 通信の場合，*MPI_Isend/Irecv()* のノンブロッキング通信を開始し，他の演算を実行することで通信と演算をオーバラップさせる．その後，通信対象のデータの read/write が起きる直前で *MPI_Wait()* により通信完了を保証する実装が一般的である．しかし，複雑な演算や部分的に通信を実行する場合，通信と演算を完全にオーバラップすることは困難であり，並列度の高いメニーコアプロセッサではプログラム全体として見たときに性能ボトルネックとなりやすい．また，ユーザがプログラム中の同期箇所を把握し，通信と演算のオーバラップを記述する必要があるため，プログラミングコストも大きい．

OpenMP のデータ依存に基づくタスク並列モデルでは，タスク内で MPI のような通信を記述することが可能である．通信対象を演算と同じ粒度のデータ依存として記述することで，そのデータが必要となる時点で OpenMP ランタイムが依存関係に基づき実行順序を決定するため，ユーザが同期箇所を明示的に決定する必要がない．従って，通信と演算のオーバラップ率を向上させ，さらにデータ依存のみの簡易な記述が実現できると考えられる．

分散メモリ環境でタスク並列モデルを実行する場合，前述したとおり各ノードが持つスレッドのタスク上で通信が実行される．スレッド上で MPI の通信 API を呼ぶ場合，マルチスレッドでの通信を許可する MPI_THREAD_MULTIPLE を MPI の初期化時に指定する必要がある．しかし，MPI_THREAD_MULTIPLE での通信は，一般的なプロセスレベル（MPI_THREAD_SINGLE, MPI_THREAD_FUNNELED）の通信と比較して性能が低く，その通信性能を改善するために多くの研究が行われてきた．H. Dang らは文献 [26] にて，MPICH の実装をベースとしマルチスレッド通信時の lock/unlock の最適化を行うことで，クリティカルセクションを減らし MPI_THREAD_MULTIPLE での通信性能を向上させた．また，M. Si らは文献 [27] にて，通信のみを実行するゴーストプロセスをユーザに指定させ，そのプロセスのみが通信を実行する実装とした．ユーザはスレッド上で実行するように通信を記述するだけで，自動的にゴーストプロセスが通信を実行する．K. Vaidyanathan らは文献 [28] にて，通信用のスレッドを定義し他のスレッド上で実行される通信を通信用スレッドに自動でオフロードすることで性能向上を行った．従って，MPI_THREAD_MULTIPLE 環境での通信性能の向上のためには，MPI コンパイラの実装を修正し MPI_THREAD_MULTIPLE 自体の性能を向上させるか，ユーザからはマルチスレッドに通信しているように見えるがランタイムが自動的に 1 つの実行単位に通信を集約する 2 種類の方法が考えられる．しかし，上記の研究では MPI コンパイラの実装依存となることや，通信集約用のライブラリが別途必要となる．そこで本研究では，より汎用的な通信最適化の実装を行うため，OpenMP レベルで文献 [28] の通信用スレッドの生成とそのスレッドへの自動的な通信の集約機構を実装し，通信性能向上を目指す．

1.4 PGAS 言語 XcalableMP (XMP)

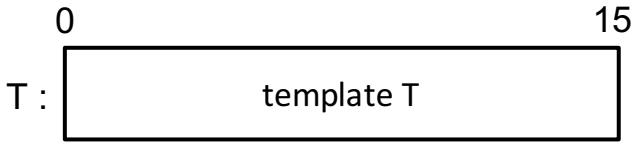
本研究で用いる PGAS 言語 XMP の概要を述べる．XMP は，次世代並列プログラミング言語検討委員会及び PC クラスタコンソーシアム並列プログラミング言語 XcalableMP 規格部会 [29] により，仕様検討及び策定が行われている分散メモリ型 Single Program Multiple Data（SPMD）を実行モデルとする C，Fortran 対応の PGAS 言語である．リファレンス実装である Omni XcalableMP Compiler は，理化学研究所計算科学研究機構プログラミング環境研究チームと筑波大学 HPCS 研究室による Omni Compiler プロジェクト [30] により開発が進められており，XMP 指示文が挿入された C，Fortran コードを MPI で記述されたランタイム呼び出しへと変換する source-to-source なトランスレータである．MPI は 0 オリジンで “プロセス” を実行単位としているが，XMP は 1 オリジンで “ノード” としている．XMP はプログラミングモデルとしてグローバルビューとローカルビューの 2 種類を提供している．グローバルビューモデルは典型的なデータ分散や通信を指示文で提供しており，ローカルビューモデルは Fortran 2008 より正式採用された coarray と互換性がある片側通信をサポートしている．

1.4.1 グローバルビューモデル

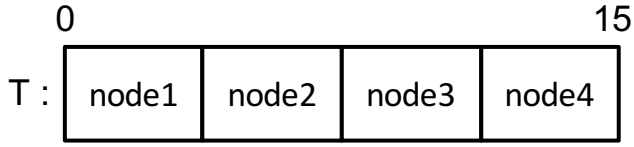
グローバルビューモデルは，問題で扱うグローバルな配列を各ノードに分散する指示文を挿入することで並列実行を可能とするプログラミングモデルである．従って，基本的に逐次プログラムに指示文を挿入するのみで並列プログラムを実装できる．グローバルビューではテンプレートと呼ばれる仮想的なインデックス空間を用いてデータや処理の分散を記述する．図 1.4 にグローバルビューモデルのプログラム例を示す．以降のグローバルビューによるプログラムの説明は，全てサイズ 16 の配列 A を 4 ノードでブロック分割した場合のデータ分散，並列実行及び通信・同期の例となる．まず，`nodes` 指示文により実行ノード集合を定義する．数値の記述により静的に実行ノード数を指定できるほか，“*” とした場合はユーザが実行時に指定したノード数を基に，XMP ランタイムが自動的に実行ノード数を判断する．`template` 指示文はテンプレートを定義する．図 1.4 では分散する配列 A のサイズに合わせてテンプレート長を指定している．次に，テンプレートに対して `distribute` 指示文で分割方法（ブロック，サイクリック，ブロックサイクリック及び不均等ブロック）を指定し，`align` 指示文により対象の配列と分散されたテンプレートを対応付けることで，各ノードへとデータ分散を行う．分散されたデータを用いる for 文に対して `loop` 指示文を挿入することで，ユーザは各ノードへと分散されたデータの配置を意識することなく，並列実行が可能である．基本的にグローバルビューによる並列プログラムは指示文追加による実装であるため，XMP コンパイラが無い環境では逐次実装の C，Fortran のプログラムとして実行できる．

XMP は分散配列に対する通信をサポートするため `shadow`，`reflect` 及び `gmove` 指示文を提供している．`shadow`，`reflect` 指示文は，ステンシル演算などで広く用いられている袖領域通信を実行する指示文である．図 1.5 は，グローバルビューによりブロック分割された 1 次元配列に対する袖領域通信の例である．`shadow` 指示文により，各ノードに分散された配列の上端・下端に任意幅の袖領域を確保する．例の場合は *“A[1:1]”* と記述されているため，分散配列 A の上端・下端にそれぞれ 1 要素ずつ領域が

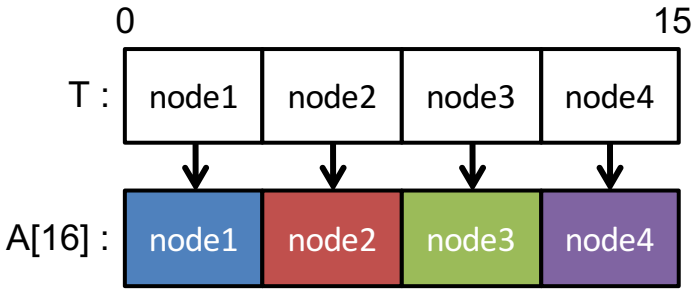
```
int A[16], res;
#pragma xmp nodes P(4)
#pragma xmp template T(0:15)
```




```
#pragma xmp distribute T(block) onto P
```




```
#pragma xmp align A[i] with T(i)
```




```
#pragma xmp loop(i) on T(i) reduction(+:res)
for (int i = 0; i < 16; i++) {
  A[i] = func(i);
  res += A[i];
}
```

図 1.4　グローバルビューモデルのプログラミング例（データ分散，並列実行）．

確保される．`shadow` 指示文で指定された配列を `reflect` 指示文で指定することで各ノードが持つ袖領域の値を更新する．

`gmove` 指示文は分散配列に対する様々な通信が実行可能な指示文である．`gmove` 指示文には変数や分散配列による代入文が記述され，左辺と右辺の所有ノードが異なる場合に通信が発生する．さらに，`gmove` 指示文に対して `in/out` を指定すると，ノード間での通信は片側通信（Put，Get）となる．図 1.6 に `gmove` 指示文により記述可能な通信パターンの一部を示す．左辺と右辺の両方に配列セクションを含む分散配列を指定した場合，図中の *send-recv* のようにノード間でのデータコピーが実行される．複数ノードに跨る配列セクションを記述した場合も同様に実行が可能で，対応するノード全てが実行対象となる．左辺にローカル変数が記述された場合は図中の *broadcast* のように，ノード 1 が持つ *A[0]* の値を各ノードが持つローカル変数 *B* に対して代入するブロードキャストとなる．右辺にローカル変数が記述さ

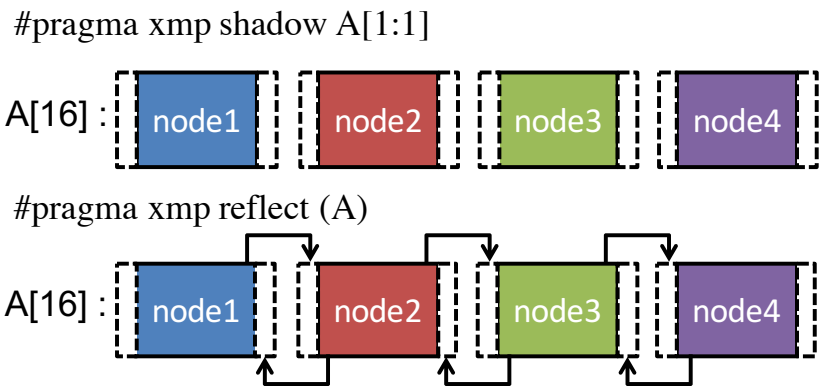


図 1.5 グローバルビューモデルのプログラミング例（shadow，reflect 指示文）.

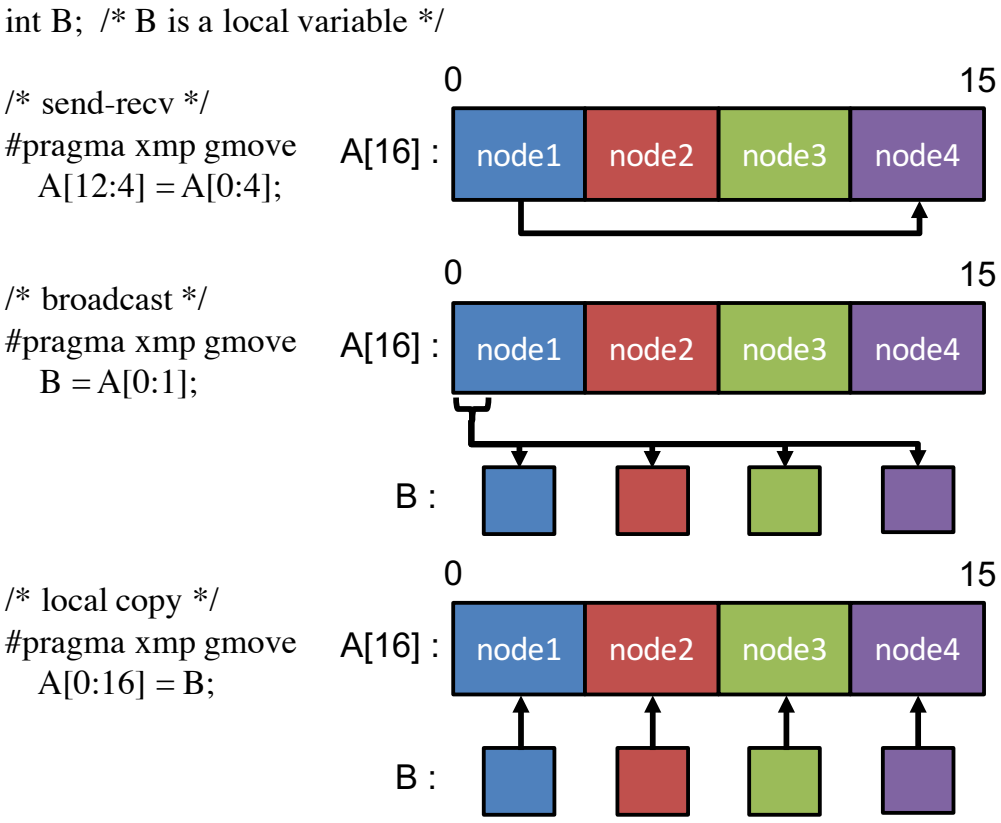


図 1.6 グローバルビューモデルのプログラミング例（gmove 指示文）.

ソースコード 1.1 ローカルビューモデルのプログラミング例.

```
1  int A[25]:[*], B[25], tag, mype, status;
2  #pragma xmp nodes P(4)
3  mype = xmp_node_num();
4
5  /* Example of coarray Get */
6  if (mype == 4) {
7        B[0:25] = A[0:25]:[1];
8  }
9  xmp_sync_all(&status);
10
11 /* Example of coarray Put */
12 if (mype == 1) {
13       A[0:25]:[4] = B[0:25];
14       xmp_sync_memory(&status);
15 #pragma xmp post (P(4), tag)
16 }
17 if (mype == 4) {
18 #pragma xmp wait (P(1), tag)
19 }
```

れた場合は *local copy* が実行される．分散配列 *A* の全ての要素が指定されているため，各ノードが持つローカル変数 *B* を分散配列 *A* の全てインデックスに対して代入する．

1.4.2 ローカルビューモデル

ローカルビューモデルは，各ノードが持つローカルデータに対して通信を行うプログラミングモデルである．XMP では，Fortran2008 から正式採用された CAF をベースとした coarray 記法を提供している．XMP/Fortran における coarray は Fortran2008 の上位互換であり，XMP/C における coarray は XMP の独自拡張である．coarray の実行単位はイメージである．各ノードが持つ固有の名前空間に対して，配列代入文形式でイメージ番号，通信要素数を指定するのみで片側通信（Put，Get）を実行可能であるため，MPI のようにローカルデータの振る舞いを詳細に記述できる．片側通信の対象となる配列は，配列宣言時に角括弧で実行対象のイメージ集合を記述する．複数角括弧により多次元イメージに対する割付も可能である．1 次元空間の全実行イメージを対象とする場合は角括弧に “*” を指定する．片側通信は非同期に実行されるため，ユーザが同期を明示的に記述する必要がある．XMP/Fortran は CAF と同等であり，XMP/C では全体同期の *xmp_sync_all()*，実行主体における通信完了を保証する *xmp_sync_memory()*，一対一同期が記述可能な *xmp_sync_image()* などを提供している．

ソースコード 1.1 に XMP/C の coarray を用いた片側通信の例を示す．配列 *A* を片側通信の対象の配列とするため，1 行目のように角括弧による指定を行う．例では，1 次元空間の全実行イメージを対象とする．7，13 行目がそれぞれ coarray Get，Put を表しており，配列記述後方のコロン以降が通信対象のイメージを示す．7 行目ではノード 4 が実行主体であり，ノード 1 が持つ配列 *A* のインデックス 0 から 25 要素に対して Get を実行する．片側通信は非同期に実行されるため，9 行目の *xmp_sync_all()* にて全体同期により片側通信の完了を保証する．13 行目ではノード 1 が実行主体であり，ノード 4 が持つ配列 *A* のインデックス 0 から 25 要素に対して Put を実行する．この例は，全体同期ではなく `post/wait` 指示文による一対一の同期を示す．14 行目の *xmp_sync_memory()* により，実行主体において片側通信の完了を保証し `post` 指示文で対象ノードを起動する．ノード 4 では `wait` 指示文により `post` 指示文が実行されるまで待機するので，ノード 4 においてノード 1 で実行された片側通信の完了を保証することが可能となる．

1.5 目的

本研究では，大規模な分散メモリシステムにおける並列プログラムの性能と生産性の向上を主な目的とする．その目的を達成するために 2 つの研究課題に取り組む．まず 1 つとして，PGAS モデルのグローバルビューとローカルビューの欠点を補う新たなプログラミングモデルの提案を行う．提案モデルによる実アプリケーションへの適用を通して，実アプリケーションのような複雑なプログラムに対する PGAS モデル適用による知見を示すとともに，オリジナルの並列実装との比較による性能と生産性の評価を行う．本研究では PGAS 言語 XMP を対象に，グローバルビューとローカルビューを組み合わせたモデルであるハイブリッドビューを提案する．ハイブリッドビューモデルにより，実アプリケーションである核融合シミュレーションコード GTC-P[31, 32] を実装し，提案モデルが MPI と同等の性能が得ることが可能であり，さらに生産性の高い実装であることを示す．また，ハイブリッドビューはノード間での通信，データ分割や並列実行を簡易に記述可能としているモデルであり，メニーコアプロセッサを搭載するシステムにおいてはノード間並列に加えて，ノード内並列の性能，生産性も考慮に入れる必要がある．そこで 2 つ目の課題として，メニーコアシステム上で高い性能や生産性を両立させるプログラミングを可能とすべく，PGAS モデルによるタスク並列モデルの提案を行う．XMP を対象としてノード内/間におけるタスク依存を統一的に記述可能とする指示文を提案し，提案指示文によるベンチマークプログラムの実装，評価により，メニーコアクラスタ上で性能，生産性を示す．

本研究の貢献は以下の通りである．

- XMP のハイブリッドビューモデルにより，従来では複雑な通信や演算によりグローバルビューが適用できなったプログラムに対して，部分的にグローバルビューが適用可能となり，生産性の高い実装が可能となった．また，通信はローカルビューによる簡易な記法による実装となり，通信記述の可読性は高い．
- XMP のタスク並列モデルにより，従来のループ並列によるワークシェアリングの全体同期を排除し，データ依存に基づくタスク間の細粒度な同期を分散メモリ環境で実現した．また，ベンチマークプログラムへ提案手法の適用によりメニーコアシステム上での高性能，高生産性を示した．

1.6 本論文の構成

本論文の各章の構成は以下の通りである．第 2 章ではハイブリッドビューモデルによる核融合シミュレーションコード GTC-P の実装と評価を示す．グローバルビューとローカルビューの欠点を補う新しいプログラミングモデルであるハイブリッドビューの提案を行う．提案モデルにより実アプリケーションを実装し評価をすることで，オリジナルの MPI 実装との性能や生産性の比較によるハイブリッドビューの有用性を示す．GTC-P は GPU や Intel Xeon Phi の適用による性能面での改善は頻繁に行われてきたが，プログラムの生産性の観点から評価を行うことは本研究が初めてである．第 3 章では，メニーコア環境における高性能，高生産性を得ることが可能なプログラミングモデルの提案を行う．本研究ではタスク並列モデルに注目し，メニーコア環境でのタスク並列記述の利点を述べるとともに，従来のループ並列によるワークシェアリングの実装との比較を示す．また，PGAS モデルにタスク並列機能を取り入れることで，メニーコア環境での性能，生産性の両立を目指す．最後に第 4 章で本研究全体のまとめと今後の課題を示す．

第 2 章

ハイブリッドビューモデルによる並列プログラミング

2.1　ハイブリッドビューモデル

PGAS モデルのアプリケーションへの適用例を用いて，グローバルビューとローカルビューの利点や欠点を述べる．計算科学のアプリケーションの物理シミュレーションでよく用いられる手法として格子法と粒子法がある．差分法や有限要素法を代表とする格子法では，演算領域が格子上に割り当てられたステンシル演算を主な演算としているため，グローバルビューによる典型的なデータ分散や並列実行が適している．一方で粒子法は，演算は格子に依らず計算点が物理量とともに空間を自由に移動可能であるため，ノードを跨る通信はデータサイズや対象ノードが動的に変更されるような複雑な通信記述が必要となる．そのため，ローカルビューの片側通信による簡易な通信記述が適している．

グローバルビューによる実装は，逐次実装を維持したままで，かつ簡易な指示を与えるだけで並列実装が可能であるため，アプリケーション全体の生産性を高めることが可能である．しかし，プログラミングの制約も多く，粒子法のような複雑な通信を含むアルゴリズムには適用できず，格子法のような典型的なデータ分散や通信を行うアプリケーションにしか適用できない場合が多い．一方で，ローカルビューによる実装は，簡易な記述による片側通信と各ノード固有の名前空間におけるプログラミングであるため，粒子法のような複雑な通信パターンにも対応可能である．しかし，データ分散やループの並列実行は MPI と同様陽に記述する必要があり，プログラム全体での生産性が高いとは言えない．各実行モデルにおいて問題点も多いため，MPI と同等の性能が得ることが可能で，より生産性を高めることが容易なプログラミングモデルが求められる．本研究では，グローバルビューの簡易な記述によるデータ分散や並列実行及び通信記述を残しつつ，粒子法における通信のような複雑な記述を必要とする箇所をローカルビューで実装する，ハイブリッドに記述可能なプログラミングモデルを提案する．

2.2　核融合シミュレーションコード

核融合プラズマ中の乱流現象のシミュレーションを行う代表的な手法として，Particle-In-Cell（PIC）法とモンテカルロ法が挙げられる．このどちらの手法も荷電粒子の集合体としてプラズマを扱う粒子的描像

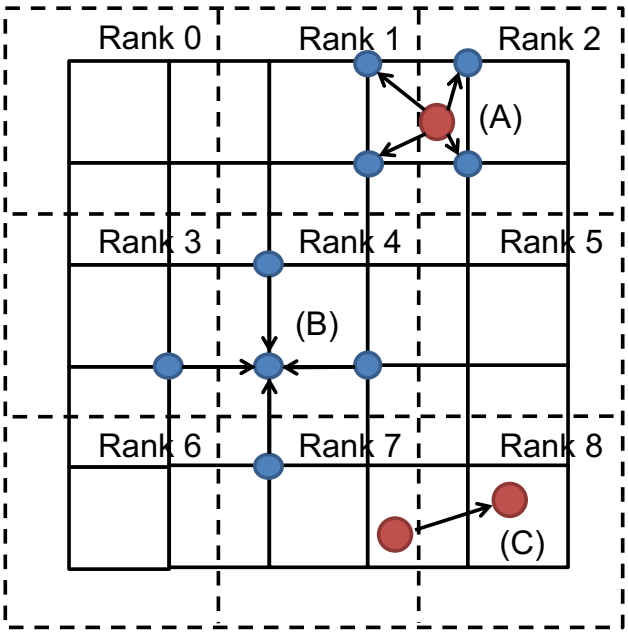


図 2.1　2 次元ブロック分割による 2 次元 PIC シミュレーションの簡易図．（A）は粒子が持つ電荷の近傍格子点への分配，（B）はポアソン方程式を解くための陽解法によるステンシル演算，（C）は粒子軌道計算をそれぞれ表す．

によるが，PIC 法は外部電磁場以上にプラズマ自身が作り出す電磁場の影響が強い現象に適しており，モンテカルロ法はプラズマ粒子や不純物イオンなどの輸送現象の解析に広く用いられている．本研究では，PIC 法による実アプリケーションである GTC-P を対象としたため，PIC 法の説明のみを示す．

2.2.1　Particle-In-Cell（PIC）シミュレーション

PIC 法は，場の計算を行う計算格子と空間を自由に動き回る格子によらない粒子軌道計算の組み合わせで構成される．以下に PIC 法のシミュレーションを示す [33].

1. 各粒子が持つ電荷を近傍格子点に分配し，加算を行う．
2. ポアソン方程式により，近傍格子点上の電荷密度から格子点上の静電ポテンシャルを求め，それを基に電場を求める．
3. 各粒子の近傍格子点から個々の粒子の現在位置での電場を求め，1 ステップ粒子の位置を進める．

図 2.1 に 2 次元 PIC シミュレーションコードに対して 2 次元ブロック分割による並列化を施した例を示す．図中の（A），（B）は隣接格子点情報を用いた場の計算を表し，（C）は粒子の軌道計算を表す．場の計算は大きく分けて 2 種類に分類され，（A）は格子内の粒子が持つ電荷を近傍格子点への分配，（B）はポアソン方程式を解くため陽解法による 5 点ステンシル演算を表す．MPI などを用いて並列実装を行った場合に通信が発生するステップとして，（A），（B）では，計算に用いる隣接格子点が別プロセスの持つ領域にある場合，（C）は粒子の移動先が別プロセスの持つ領域の場合が挙げられる．

ソースコード 2.1　XMP による PIC シミュレーションコードの実装例.

```
1  double f[X][Y], p[N]; /* Electric field and particle data */
2  double send[N], recv[N]:[*];
3  int status;
4
5  #pragma xmp nodes P(3, 3)
6  #pragma xmp template T(0:X−1, 0:Y−1)
7  #pragma xmp distribute T(block, block) on P
8  #pragma xmp align f[i][j] with T(i, j)
9  #pragma xmp shadow f[1][1]
10
11 for (int t=0; t<TIME; t++){
12     /* Calculate the grid−related work */
13 #pragma xmp loop(i, j) on T(i, j)
14     for (int i = 0; i < X; i++) {
15         for (int j = 0; j < Y; j++) {
16             f[i][j] = func(i, j);
17         }
18     }
19 #pragma xmp reflect(f) /* Update the halo region */
20     /* Calculate the particle−related work */
21     /* Pack the communication elements from array ''p'' to array ''send'' */
22     /* Calculate the destination node ''pe'' and communication size ''icount'' */
23     recv[0:icount]:[pe] = send[0:icount];
24     xmp_sync_all(&status); /* Synchronization */
25     /* Unpack the particle data from array ''recv'' to array ''p'' */
26 }
```

PGAS モデルにより PIC 法の実装を行う場合，計算格子のように分割された領域のサイズが変更されず，ステンシル演算や隣接格子間での通信が主な場合はグローバルビューモデルによる実装が適している．しかし，粒子軌道計算のような演算ステップ毎に各プロセスが受け持つデータサイズが動的に変更される演算やそれに伴う通信のグローバルビュー実装は困難であり，プログラム全体を MPI やローカルビューで実装する方法が行われてきた．ローカルビューによる実装では，MPI と比較して簡易な記法による片側通信で実装が可能だが，データ分散や並列実行は MPI 同様に記述する必要があり，プログラム全体の生産性が高いとは言えない．そこで，PGAS モデルのグローバルビューとローカルビューを組み合わせて同一のプログラムで記述するハイブリッドビューを用いることで性能を維持しつつ，生産性が向上可

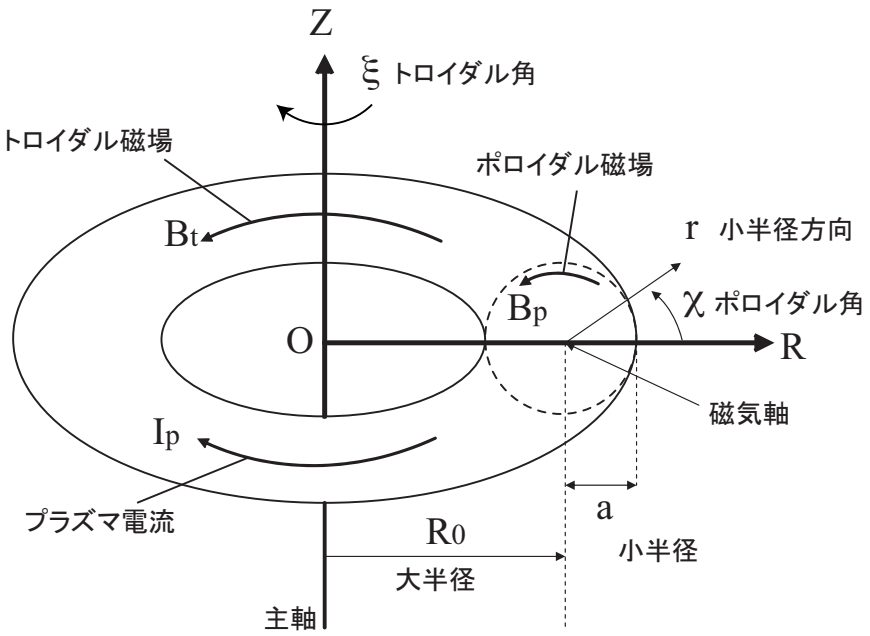


図 2.2 GTC，GTC-P の 3 次元トーラス空間の簡易図 [34].

能であると考えられる．

XMP により PIC シミュレーションを実装した例をソースコード 2.1 に示す．格子データを格納する配列 f をグローバルビューの指示文により分散する．例では，5 行目の `nodes` 指示文により 2 次元の 3×3 ノードによる実行を示す．場の計算である格子計算はステンシル演算を含む典型的なループで構成されるため，14 から 18 行目の for ループに対して 13 行目の `loop` 指示文を追加することで，各ノードに分散されたデータに対して並列実行することが可能である．また，格子計算では隣接ノードが持つ格子情報が必要となるため，9 行目の `shadow` 指示文により 2 次元空間の上下左右の袖領域を確保する．確保された袖領域は，格子計算終了後にある 19 行目の `reflect` 指示文により，隣接ノード間の通信で格子情報が更新される．PIC シミュレーションの場合は，同一アプリケーション内に粒子軌道演算のようなローカルな名前空間での記述を要求する通信が含まれる．そこで 23，24 行目のように，ローカルビューの coarray による通信をグローバルビューのプログラムと組み合わせて記述する．このようなハイブリッドビューの記述により，従来ではグローバルビューによる生産性の高い実装が不可能であったアプリケーションに対しても，グローバルビューによる記述を保ちつつローカルビューの簡易な通信で実装が可能である．

2.2.2 GTC-P

Gyrokinetic Toroidal Code（GTC）[36] は，磁場閉じ込め型核融合装置における核融合プラズマ中の微視的乱流現象の解析を目的として，米国 DoE SCiDAC，UC Irvine などで開発が進められている 3 次元ジャイロ運動論的 PIC シミュレーションコードである．本研究で対象とする実アプリケーションの Gyrokinetic Toroidal Code - Princeton（GTC-P）[31, 32] は，Princeton University や Princeton Plasma Physics Laboratory

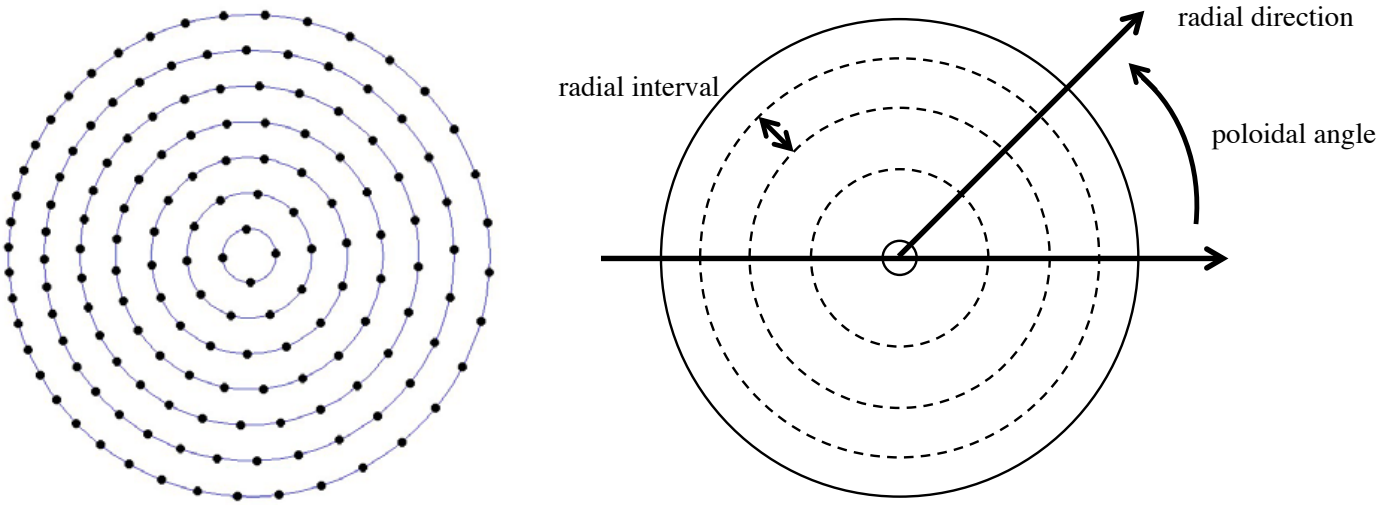


図 2.3　ポロイダル断面の概要．左: ポロイダル断面上の格子点 [35]，右: ポロイダル断面での径方向分割．

で開発が進められている国際熱各融合実験炉 ITER（International Thermonuclear Experimental Reactor）[37] 規模をシミュレート可能な実アプリケーションである．GTC は Fortran で開発されているのに対して，GTC-P は C，Fortran で開発されている．本研究では C による実装を対象とした．

図 2.2 は，GTC や GTC-P の演算領域を表した 3 次元トーラス空間の簡易図を示す．主軸 Z を回るトロイダル方向，磁気軸を回るポロイダル方向，磁気軸からトーラスの外へ伸びる方向を径方向とそれぞれ呼ぶ．また，トロイダル方向に領域分割をした際のトーラスの面をポロイダル断面と呼ぶ．

GTC-P はシミュレーションの大規模化に対応すべく開発された GTC の並列アルゴリズムの改良版である．GTC では，MPI によるプロセスレベルでのトロイダル方向の 1 次元分割と分割された領域が持つ粒子数の分割に加え，各プロセスが持つ領域内での OpenMP によるループのスレッド並列による 3 層レベルでの並列化が施されている．一方で，GTC-P では領域分割がトロイダル方向と径方向の 2 次元へと拡張され，全体として 4 層レベルでの並列実装となった．従って，トロイダル方向の分割数を N_t，径方向を N_r，分割された領域が持つ格子上の粒子数を N_{rp} と分割すると，総 MPI プロセス数は $N = N_t \times N_r \times N_{rp}$ となる．

図 2.3 に GTC-P のポロイダル断面上の格子点の分布と領域分割の方法を示す．図 2.3 の左の面が示すように GTC-P のポロイダル断面上の格子点は，等ポロイダル方向距離ではなく等径方向距離に分布する．そのため，GTC-P ではポロイダル方向ではなく径方向の領域分割を採用している．また，トロイダル方向に対して等トロイダル角での分割は可能な一方で，径方向に対して等間隔に分割を行った場合は，トロイダル断面上の格子点の分布は一定ではないためプロセス毎の演算量に大きな差が生じロードインバランスが発生する．GTC-P では，径方向の分割において各プロセスが持つ格子点数の偏りを減らすために，図 2.3 の右の面が示すように径方向の中心の領域の幅を広く，外側へ向かうにつれて幅を狭く分割することで，格子点数を極力揃えている．

GTC-P は *charge*, *poisson*, *field*, *smooth*, *push* 及び *shift* の 6 種類の演算カーネルで構成される．*charge* は格子内の粒子が持つ電荷を近傍の格子点に分配する．*poisson* はポアソン方程式を解きその結果を基に，

field にて電場を計算し，*smooth* によって電場上で電荷と静電ポテンシャルが平滑化される．*push* は各粒子の近傍格子点から個々の粒子の現在位置での電場を求め，1 ステップ粒子の位置を進める．この時粒子の移動先が別のプロセスが持つ領域の場合，*shift* にて粒子移動の通信が発生する．*poisson*，*smooth* 及び *field* が主に格子点による場の演算となり，*charge*，*push* 及び *shift* が粒子演算となる．ただし，*charge*，*push* は格子による演算も含む．

2.3 XMP ハイブリッドビュー

XMP のグローバルビューとローカルビューを組み合わせたハイブリッドビューの説明を示す．XMP では仕様上，グローバルビューとローカルビューの同一プログラム上での記述を可能としている．グローバルビューとローカルビューを同時に扱うためには，グローバルビューが定義する分散配列をローカルな名前空間でどのように扱うかを考慮する必要がある．XMP では下記の 3 種類の方法が考えられる．

- 関数の引数に分散配列を指定する．関数を呼ぶ側は分散配列であるが，関数内ではローカル配列として扱われる．
- `task` 指示文のブロック内で配列代入文により，自ノードが持つ分散配列の要素をローカル変数や配列へとコピーをする．
- XMP/C では *xmp_array_laddr()*，XMP/Fortran は `local_alias` 指示文を用いて，各ノードが持つ分散配列のローカル領域を取得する．

1 つ目の方法の詳細を示す．XMP では関数内で分散配列を扱う場合，引数に分散配列を指定し関数内で `align` 指示文などを記述することで，分散配列として再定義をする必要がある．再定義をしない場合は，ノードローカルな配列として扱われるので，この機能を利用しローカルビュープログラミングを行うことが可能である．2 つ目の場合は，自ノードが持つ分散配列の要素を直接ローカル変数や配列へと代入する．XMP では `gmove` 指示文を記述しない限り，他ノードが持つ分散配列の要素の参照を許していないため，必ず自ノードが持つ領域のみを記述する必要がある．そこで，XMP の `task` 指示文と `on` 節で，配列代入文を実行するノードをノード集合やテンプレートを用いて制限することで記述が可能とある．3 つ目は，XMP のランタイム関数や指示文を用いた例である．XMP/C は *xmp_array_laddr()* に分散配列を指定することで，分散配列のローカル領域を指すポインタを取得できる．XMP/Fortran では `local_alias` 指示文に形状無指定配列と分散配列を指定することで，分散配列のローカル領域をアクセスするためのエイリアスを定義する．この手法の場合，一度の指定でプログラム中何度でも分散配列のノードローカルな要素を直接扱うことが可能である．

2.4 実装

GTC-P の XMP 実装の方針を示す．XMP，MPI の実行単位をプロセスと統一して今後の説明を示す．GTC-P の逐次実装をベースとして提案手法であるハイブリッドビューによる実装を行う．データの分散配置や演算ループの並列実行はグローバルビューの指示文により実装し，隣接格子点間の通信は `reflect` 指示文による袖領域通信として記述する．また，粒子の移動に伴うプロセス間通信は coarray

ソースコード 2.2　MPI による GTC-P の隣接格子点の更新.

```
1  double ∗sendr, ∗recvl;
2
3  for (i = 0; i < nloc_over; i++) {
4        sendr[i] = phitmp[i ∗ (mzeta + 1) + mzeta];
5  }
6
7  MPI_Sendrecv(sendr, nloc_over, MPI_DOUBLE, right_pe, isendtag,
8                  recvl, nloc_over, MPI_DOUBLE, left_pe, irecvtag, toroidal_comm, &istatus);
```

ソースコード 2.3　XMP ローカルビューによる GTC-P の隣接格子点の更新.

```
1  double sendr[nloc_over], recvl[nloc_over]:[∗];
2
3  for (i = 0; i < nloc_over; i++) {
4        sendr[i] = phitmp[i ∗ (mzeta + 1) + mzeta];
5  }
6
7  recvl[0:nloc_over]:[right_pe] = sendr[0:nloc_over];
8  xmp_sync_memory(NULL);
9  #pragma xmp post(P1(right_pe), mype+1)
10 #pragma xmp wait(P1(left_pe), left_pe)
```

により実装し，動的に変化するデータサイズに対応する．GTC-P のオリジナルの MPI 実装と比較し，同等の性能で高い生産性が得られることを示す．また，比較としてローカルビューのみを用いて実装を行い MPI 実装と比較をすることで，ローカルビューによる通信性能と coarray の記述性の高さも合わせて示す．

XMP の両実装ともに *MPI_Bcast()* や *MPI_Allreduce()* のような全体通信は，グローバルビューが提供する `bcast`，`reduction` 指示文をそれぞれ用いる．また，GTC-P は OpenMP によるスレッドレベルでの並列化も行われているため，本実装においても XMP と OpenMP を組み合わせて記述し，その実装の性能評価も行う．

2.4.1　ローカルビュー実装

GTC-P のローカルビューによる実装を示す．GTC-P におけるプロセス間での隣接格子点の更新や粒子の移動に伴う通信は *MPI_Sendrecv()* または，*MPI_Isend/Irecv()* により記述される．通信先は常に同一であり，一部を除き隣接プロセス間での通信となる．ローカルビュー実装では，全体通信を除き全ての通信

ソースコード 2.4 MPI による GTC-P の粒子移動.

```
1 /* Send # of particles to right neighbor and receive from left neighbor */
2 MPI_Sendrecv(&nsendright, 1, MPI_INT, right_pe, sendtag,
3                 &nrecvleft, 1, MPI_INT, left_pe, recvtag, toroidal_comm, &istatus);
4 /* Send particles to right neighbor and receive from left neighbor */
5 MPI_Sendrecv(sendright, nsendright, MPI_DOUBLE, right_pe, sendtag,
6                 recvleft, nrecvleft, MPI_DOUBLE, left_pe. recvtag, toroidal_comm, &istatus);
```

ソースコード 2.5 XMP ローカルビューによる GTC-P の粒子移動.

```
1 /* Put # of particles to right neighbor */
2 nrecvleft:[right_pe] = nsendright;
3 /* Put particles to right neighbor */
4 recvleft[0:nsendright]:[right_pe] = sendright[0:nsendright];
5 /* Synchronization */
6 xmp_sync_memory(&status);
7 #pragma xmp post(P1(right_pe), mype+1)
8 #pragma xmp wait(P1(left_pe), left_pe)
```

を coarray/Put による片側通信とする.

ソースコード 2.2 に MPI 実装による隣接格子点の更新を示す．配列 *sendr*，*recvl* は通信に用いる送信，受信用のバッファであり，予めサイズ *nloc_over* 分領域が確保されている．MPI で通信をする領域は 3 から 5 行目より，格子点情報を保持する配列 *phitmp* から隣接プロセスへと送信する領域を配列 *sendr* へパッキングする．7 行目以降の *MPI_Sendrecv()* により配列 *sendr* の *nloc_over* 分をプロセス *right_pe* へ送信し，プロセス *left_pe* より送信された隣接格子点は配列 *recvl* に受信される．受信した配列 *recvl* よりアンパッキングを行い格子点を配列 *phitmp* へと戻すが，GTC-P ではアンパッキング時に演算を行いながらバッファへ代入しているため詳細なコードは省略する．

ソースコード 2.3 にローカルビューの coarray による隣接格子点の更新を示す．Omni XMP Compiler の coarray 実装は，coarray の対象となるバッファをグローバル領域に静的に確保しておく必要があるため，1 行目のように送信，受信用のバッファを静的に定義した．本研究では coarray/Put による実装としたため，受信用バッファの配列 *recvl* を coarray 用の配列として宣言した．隣接プロセスへと送信するバッファへのパッキングは MPI 実装と同様に行う．通信は 7 行目の配列代入文であり，配列 *sendr* のインデックス 0 からサイズ *nloc_over* をプロセス *right_pe* の配列 *recvl* のインデックス 0 からサイズ *nloc_over* へ Put を実行する．coarray 構文後はローカルでの通信完了のみ保証されるため，*xmp_sync_memory()* により通信対象のプロセスへの到達の保証が必要となる．また，Put が実行されるプロセスは，Put が完了したかどうかを知ることはできないため，XMP の `post`，`wait` 指示文により片側通信の完了通知を記

述する必要がある．ソースコード 2.3 の場合，9 行目で通信対象のプロセス *right_pe* に対して `post` 指示文を実行し，プロセス *left_pe* より実行される Put が完了するまで 10 行目の `wait` 指示文で待機する．

ソースコード 2.4 に MPI 実装による粒子移動の実装を示す．粒子移動の実装はプロセス *right_pe* に対して配列 *sendright* の粒子データを送信し，プロセス *left_pe* より配列 *recvleft* に粒子データを受信する．通信される粒子サイズはイテレーション毎に異なるため，GTC-P では 1 回の粒子移動に対して 2 種類の通信による実装が取られている．まず，2 行目の通信によりプロセス *left_pe* から送信される粒子データのサイズ *nrecvleft* のみを受信し，5 行目の通信で受信した粒子サイズを基にプロセス *left_pe* より粒子データを配列 *recvleft* に受信する．送信するデータは通信相手に依らずデータサイズが決定されるため，通信する粒子データのサイズ *nsendrecv* と粒子データの配列 *sednright* をそれぞれ送信する．

片側通信の場合は，通信対象に依らずに通信を直接実行可能であるため，対象に通信サイズを前もって知らせるソースコード 2.4 の 2 行目の通信は必要ない．しかし，GTC-P の場合は受信する粒子サイズを用いた演算が行われるため，MPI 実装と同様に 2 回の通信による実装とした．ソースコード 2.5 にローカルビューの coarray による粒子移動の実装を示す．2 行目で粒子サイズ *nsendright* をプロセス *right_pe* の *nrecvleft* へ Put し，4 行目で粒子データである配列 *sendright* のインデックス 0 からサイズ *sendright* をプロセス *right_pe* の配列 *recvleft* のインデックス 0 からサイズ *nsendright* へ Put する．通信の同期は隣接格子点の更新と同様に *xmp_sync_memory()*，`post`，`wait` 指示文を用いた隣接同士のプロセスによる同期とした．

2.4.2　ハイブリッドビュー実装

GTC-P のハイブリッドビューによる実装を示す．GTC-P のトロイダル方向，径方向の領域分割は 2.2.2 節よりトロイダル方向は等間隔に分割されるが，径方向に対しては格子点数を極力揃えるために初期格子点演算時に不均一に分割される．XMP では，グローバルビューが提供する `distribute` 指示文の `gblock` によるデータ分散を行う．`gblock` は各プロセスが受け持つデータ領域のサイズをユーザが指定することが可能であり，プロセス毎に不均一にデータを分散配置することが可能である．ソースコード 2.6 は，演算領域に対して `gblock` 分割を用いた場合のグローバルビューによる GTC-P のデータ分散と並列実行の実装である．18 行目の `distribute` 指示文で `gblock` 指定の際に，各ノードが持つ分割領域のサイズが格納された配列（例では 11 行目の配列 *b* を指す）を指定することで不均一な分割を行う．MPI 実装では，各プロセスが持つ配列サイズは動的に確保されるが，ハイブリッドビュー実装では全て静的に記述している．XMP の仕様では動的にテンプレートや分散配列を定義することが可能であり，Omni XMP Compiler にはそれらの機能が既に実装されている．そこで，今後の課題として動的にテンプレート，分散配列を確保する実装を行うことが挙げられる．また，for ループの演算範囲は各プロセスが持つ配列サイズによって開始，終了インデックスを指定する必要はなく，27 から 31 行目の逐次実装の for ループに対して XMP の `loop` 指示文や OpenMP の `parallel for` 指示文によってプロセス間での並列実行，プロセス内でのスレッド並列を記述可能である．

隣接格子点間の通信にはグローバルビューの `reflect` 指示文を用いる．ソースコード 2.7 は，`reflect` 指示文による実装であり，ソースコード 2.2，2.3 と同様の通信を表す．`width` 節により，定義された袖領域の片側のみを対象とした通信とし，`periodic` 指定により極座標系のような周回データ

ソースコード 2.6 XMP ハイブリッドビューによる GTC-P のデータ分散と並列実行.

```
1  #define n_t 2
2  /* Number of the toroidal domain decomposition */
3  #define n_r 4
4  /* Number of the radial domain decomposition */
5  #define n_rp 2
6  /* Number of the particle decomposition */
7
8  #define nloc_all 107722
9
10 double phitmp[nloc_all][2*n_t];
11 int b [n_r*n_rp] = {10967,10967,14086,14086,16164,16164,12644,12644};
12 /* Block size of each nodes in the ''gblock'' distribution */
13
14 #pragma xmp nodes P2(n_r * n_rp, n_t)
15 /* Number of processes (nodes) */
16 #pragma xmp template T(0:nloc_all-1, 0:2*n_t-1)
17 /* Template length */
18 #pragma xmp distribute T(gblock(b), block) onto P2
19 /* Distribution format of the template */
20 #pragma xmp align phitmp[i][j] with T(i, j)
21 /* Alignment of an array with a template */
22 #pragma xmp shadow phitmp[0][1:0]
23 /* Assignment of the halo region */
24
25 #pragma xmp loop (i, j) on T(i, j)
26 #pragma omp parallel for
27 for (int i = 0; i < nloc_all; i++) {
28       for (int j = 0; j < 2 * n_t - 1; j++) {
29             phitmp[i][j] = func(i, j);
30       }
31 }
```

ソースコード 2.7　XMP ハイブリッドビューによる隣接格子点の更新.

```
1 #pragma xmp reflect (phitmp) width (0, /periodic /1:0)
```

表 2.1　実験環境（HA-PACS Base Cluster）.

CPU	Intel Xeon E5-2670 × 2 (2.6GHz)
	CPU (8 cores/CPU) × 2 = 16 cores
Memory	128GB, DDR3 1600MHz
Interconnect	Mellanox Connect-X3 Dual-port QDR
Software	GNU 4.4.7
	MVAPICH2 2.0
	Omni Compiler 0.9.0
	GASNet 1.24.0

に対して，末端プロセス同士の袖領域の更新を行うことを表す．袖領域として隣接格子点間の通信を行うことで，MPI，ローカルビュー実装で行っていた通信のためのパッキング/アンパッキングは，XMP ランタイム内で自動的に実行される．一方で，粒子移動による通信は coarray で実装を行う．2.3 節に記述された手法を用いて分散配列をプロセスローカルな配列として扱う．GTC-P の計算は関数毎に分けて記述されているため，本研究では，関数の引数として分散配列を渡し，関数内でローカル配列として扱う方法と，*xmp_array_laddr()* による分散配列のローカル領域を指すポインタによる実装の 2 種類の方法を用いた．そのため，グローバルビューで記述された実装に対してローカルビューが使用可能であり，粒子移動による通信はローカルビュー実装であるソースコード 2.5 と全く同じである．

2.5　評価

GTC-P のオリジナルの MPI 実装と本研究で提案するハイブリッドビューによる実装を分散メモリ環境上で実行，比較をすることで，ハイブリッドビューによる実装の性能と生産性を示す．また，ローカルビューのみによる実装の評価も行い，ローカルビューが提供する coarray と MPI を比較したときの性能と生産性も合わせて示す．全ての実装においてストロング・ウィークスケーリングの 2 種類の手法で性能評価を行う．

2.5.1　通信性能の評価

2015 年 3 月現在，Omni XMP Compiler の XMP/C の coarray は，ランタイムライブラリの実装に MPI ではなく GASNet[40] が用いられている．本研究では，ハイブリッドビュー，ローカルビューによる実装を行うにあたり，*MPI_Send/Recv()* などによる一対一通信を coarray による片側通信の Put へと変更を行う．通信方法の違いや GASNet の最適化などにより，アプリケーションの性能に影響を与える可能性があ

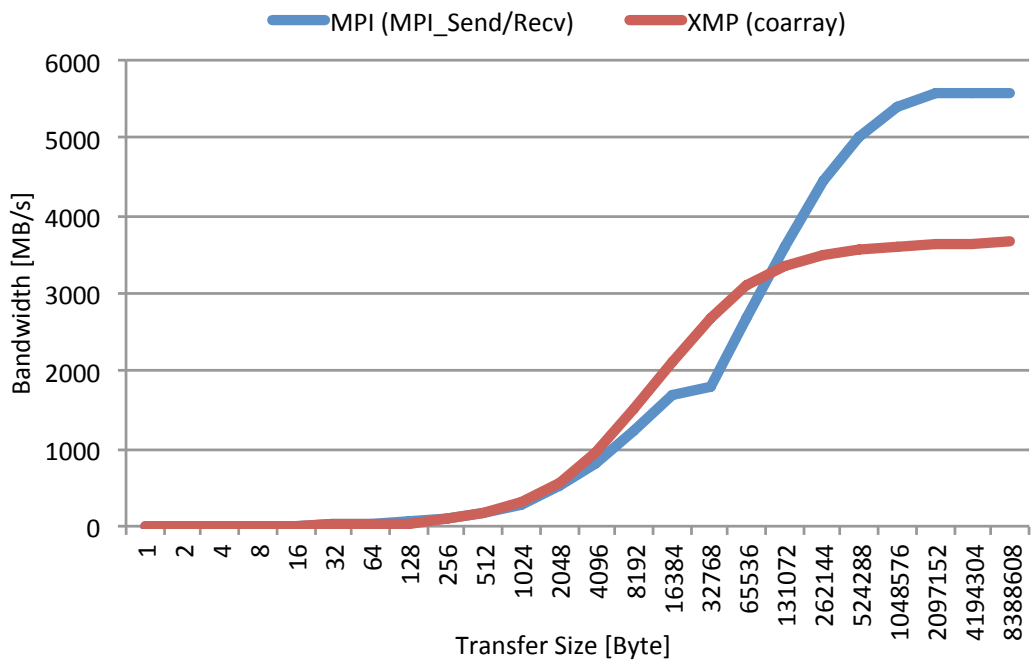


図 2.4 通信性能の評価（HA-PACS Base Cluster）．MPI（*MPI_Send()/MPI_Recv()*）と XMP（coarray/Put）の通信性能の比較．

ることから，Ping-Pong プログラムによる通信性能の評価を示したのちにアプリケーションの評価を行う．実験環境として，筑波大学計算科学研究センターの超並列 GPU クラスタである HA-PACS Base Cluster 部 [39] を用いる．HA-PACS の 1 ノードの計算機環境や使用したソフトウェアのバージョンを表 2.1 に示す．1 ノードは，Intel Xeon E5-2670 の 2 ソケット構成であるため 16 コアを利用可能である．インターコネクトとして Mellanox Connect-X3 Dual-port QDR を採用しており理論ピークバンド幅は 40Gb/s×2 である．1 ノードに 4GPU を搭載しているが，本研究ではホスト CPU のみを対象とする．XMP には Omni Compiler 0.9.0 を利用し，XMP のバックエンドやオリジナルの MPI 実装の実行のための MPI コンパイラには，オハイオ州立大学が研究開発を進めている MVAPICH2[41] 2.0 を使用する．

MPI 通信の評価のための Ping-Pong プログラムは，オハイオ州立大学が提供する OSU Micro-Benchmarks[38] を使用し，coarray の評価には OSU Micro-Benchmark をベースに片側通信による Ping-Pong プログラムを実装した．HA-PACS の InfiniBand は 2 ポートを持つ HCA カードを採用している．そのため，MPI や GASNET の環境変数を MV2_NUM_PORTS=2，GASNET_IBV_PORTS="mlx4_0:1+mlx4_0:2" とし，MPI と XMP ともに 2 ポートを使用する設定とした．図 2.4 に MPI（*MPI_Send/Recv()*）と XMP（coarray）の通信性能を示す．結果として，通信サイズが 64KB までは GASNet 実装による coarray の通信性能が良く，それ以降は *MPI_Send/Recv()* による通信が有利となり，最大で 2GB/s の差が生じることがわかった．

表 2.2　各次元の分割数（$N_t \times N_r \times N_{rp}$）=（トロイダル方向 × 径方向 × 粒子数）.

Processes	Toroidal	Radial	Particle
16	$2 \times 2 \times 4$	$2 \times 4 \times 2$	$2 \times 2 \times 4$
32	$4 \times 2 \times 4$	$2 \times 8 \times 2$	$2 \times 2 \times 8$
64	$8 \times 2 \times 4$	$2 \times 16 \times 2$	$2 \times 2 \times 16$
128	$16 \times 2 \times 4$	$2 \times 32 \times 2$	$2 \times 2 \times 32$
256	$32 \times 2 \times 4$	$2 \times 64 \times 2$	$2 \times 2 \times 64$
512	$64 \times 2 \times 4$	$2 \times 128 \times 2$	$2 \times 2 \times 128$

表 2.3　ウィークスケーリング評価時の GTC-P の問題サイズを決定する各パラメータの値.

Problem Size A	Default	Toroidal	Radial	Particle
mstep	100	20	20	20
mpsi	90	90	90–2880	90
mzetamax	64	2–64	2	2
micell	100	100	100	100–3200

2.5.2　問題サイズ，分割方法

GTC-P のデータ分割やプロセスマッピング，問題サイズを示す．GTC-P は MPI によりトロイダル方向，径方向及び分割された領域内の粒子数の 3 次元分割を行っている．そこで，2 つの次元の分割数を固定し，1 次元のみの分割数を変動させた評価を行う．ストロング・ウィークスケーリングでの評価を行うにあたり，表 2.2 のように分割数を決定した．表の左の列から総プロセス数，トロイダル方向の分割数を変動した場合の各次元のプロセス数，径方向及び粒子数の場合と並ぶ．2 つの次元の分割数をそれぞれ 2 プロセスに固定し，1 次元の分割数を 4 から 128 プロセスと増加させる．例えば，粒子数の分割数を 4 から 128 プロセスへと増加させる場合，トロイダル方向と径方向分割数はそれぞれ 2 プロセスとなる．しかし，トロイダル方向はトロイダル面毎に格子点を保持するため，分割数の増加毎に演算量も増加する．従って，トロイダル方向に限りウィークスケーリングのみの評価となる．また，粒子数や径方向と同様に分割を行った場合，16 プロセスでの分割数を $4 \times 2 \times 2$ と $2 \times 2 \times 4$ とした場合に演算量が倍異なる．そのため，他の分割方法と演算量を揃えるため，トロイダル方向の分割数を変動させた場合の評価に限り，トロイダル方向の分割数を 2 から 64 プロセスまでとし，径方向の分割数は 2 プロセス，粒子数の分割数は 4 プロセスとする．評価には 1 ノードあたり 16 プロセスを配置し，最大 32 ノード 512 プロセスを用いて計測を行った．XMP/MPI+OpenMP 実装の評価には，1 ノード 1 プロセスとして 32 ノードを使用し，1 プロセスあたりのスレッド数を 1 から 16 へと変動させる．分割方法は，トロイダル方向，径方向及び粒子数の分割数をそれぞれ $2 \times 8 \times 2$，$2 \times 2 \times 8$ とした．

GTC-P には演算量を決定するパラメータとして，演算ステップを表す *mstep*，径方向の格子数 *mpsi*，最

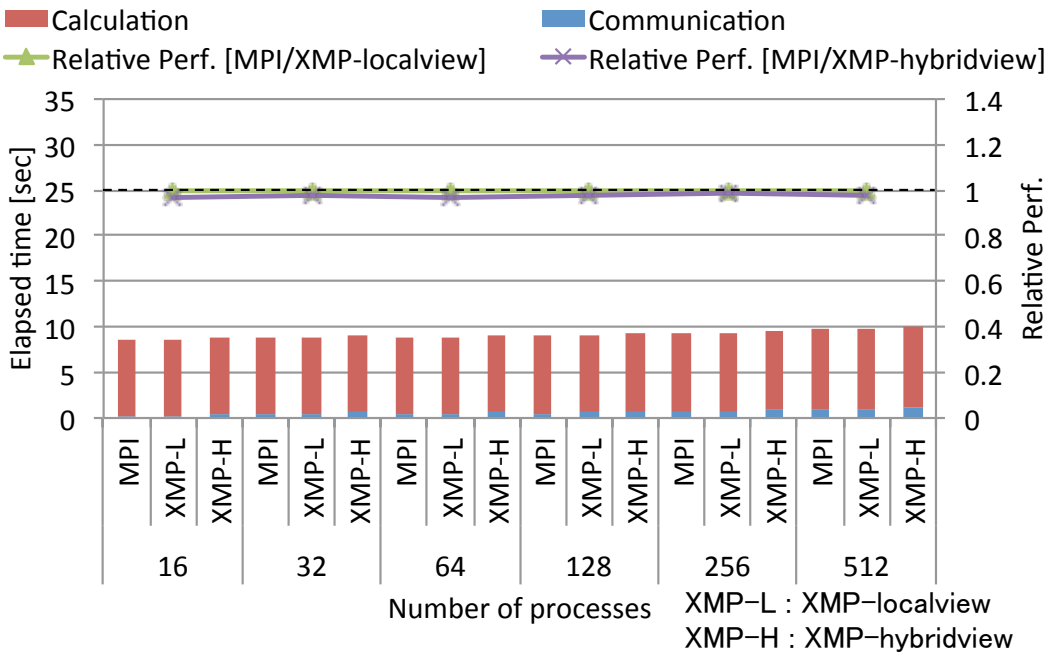


図 2.5 ウィークスケーリングの性能評価（トロイダル方向の分割数を変動）.

外殻でのポロイダル格子数 *mthetamax*，トロイダル格子数 *mzetamax* 及び格子点あたりの粒子数 *micell* がある．GTC-P は問題サイズ A から D の 4 種類を提供しており A が最小の問題サイズとなる．本研究では *mstep* を 20 とし，*mpsi*，*mzetamax*，*micell* は問題サイズ A を基に分割数単位で値を変更し，ストロング・ウィークスケーリングの評価を行う．問題サイズや各パラメータの値を表 2.3 に示す．ウィークスケーリング評価時には各分割数を増加させる毎にパラメータを増加させる．例えば，粒子数の分割数を増加させる場合，*mpsi*，*mzetamax* の値を固定し，*micell* の値を 100 から 3200 へと増加し評価を行う．ストロングスケーリング評価時には，*mstep*，*mpsi*，*mzetamax* 及び *micell* をそれぞれ，20，90，2，100 とした．

2.5.3 性能評価

ウィークスケーリングの評価

図 2.5，2.6 及び 2.7 にトロイダル方向，径方向，粒子数の分割数を変動させた場合のウィークスケーリングによる評価を示す．棒グラフは実行時間，折れ線グラフは MPI 実装の性能を 1 としたときのローカルビュー，ハイブリッドビュー実装の相対性能を表し，XMP-L はローカルビュー，XMP-H はハイブリッドビューによる実装の性能を示す．ローカルビューによる実装は図 2.5，2.7 より，トロイダル方向と粒子数の分割数変動時に MPI 実装とほぼ同等の性能が得られた．一方で，図 2.6 の径方向分割時では最大で 8% の性能差が生じた．ハイブリッドビューによる実装は，ローカルビューと同様にトロイダル方向，粒子数の分割数変動時はほぼ同等の性能が得られたが，径方向分割時には 6 から 25% 性能が低下した．

トロイダル方向，粒子数の分割数を変動させた場合は，MPI，XMP 実装ともに性能がスケールしているが，径方向分割時にはプロセス数が増加する毎に全ての実装において性能が低下している．この問題の

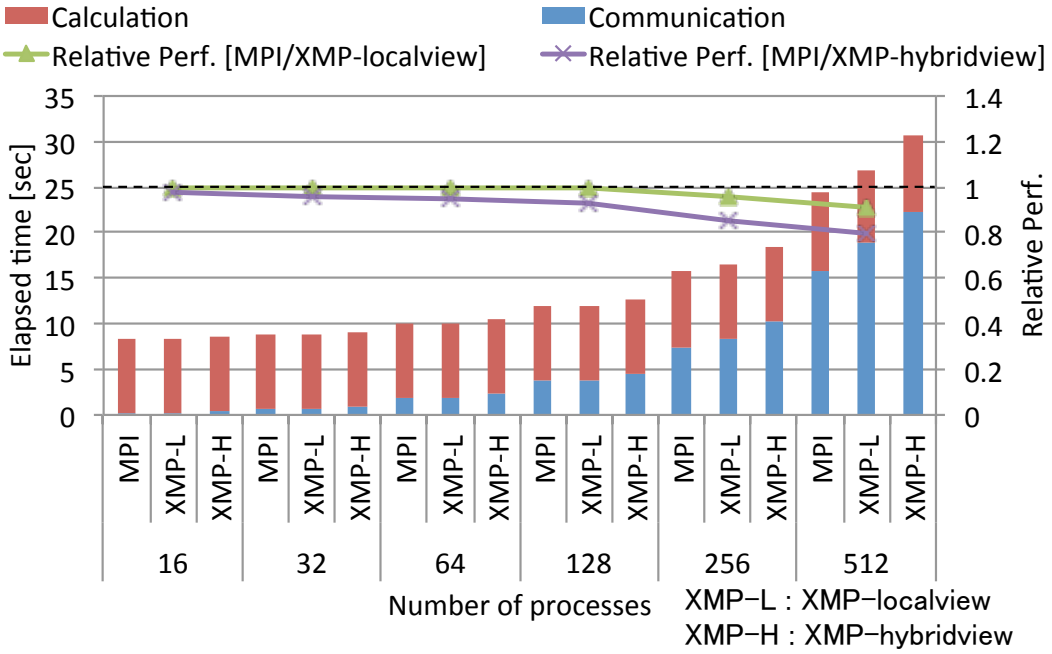


図 2.6　ウィークスケーリングの性能評価（径方向の分割数を変動）.

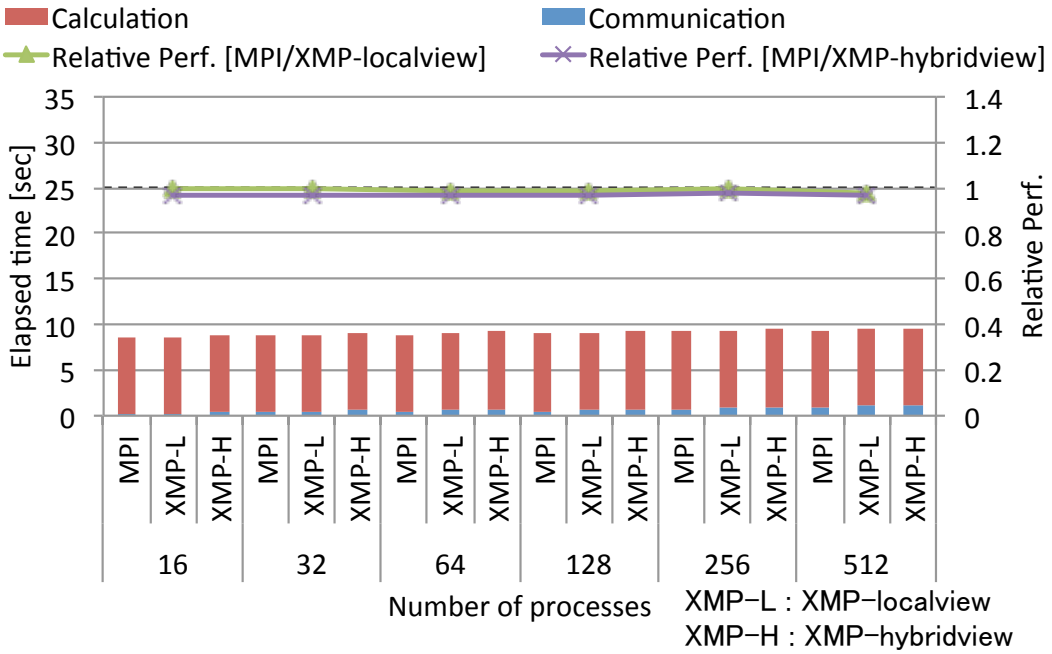


図 2.7　ウィークスケーリングの性能評価（粒子数の分割数を変動）.

表 2.4 MPI 実装におけるトロイダル方向，径方向及び粒子数の分割数を変動させた場合の実行時間．表の値は通信時間を除いた各プロセスの最大と最小の演算時間，括弧の値はそのプロセスが持つ格子点数を示す．

	Toroidal	
Processes	Minimum	Maximum
16	8.408406 (19805)	8.548204 (19916)
32	8.440145 (19805)	8.541321 (19916)
64	8.44846 (19805)	8.631631 (19916)
128	8.511492 (19805)	8.718713 (19916)
256	8.6418 (19805)	8.853517 (19916)
512	8.865397 (19805)	9.109388 (19916)
	Radial	
Processes	Minimum	Maximum
16	8.114932 (10967)	8.270015 (16164)
32	8.083982 (12104)	8.539186 (24200)
64	8.075058 (14130)	9.487029 (33462)
128	8.070919 (17422)	11.014277 (74745)
256	8.232447 (23198)	12.686402 (141700)
512	8.763279 (34522)	16.508915 (270844)
	Particle	
Processes	Minimum	Maximum
16	8.408406 (19805)	8.548204 (19916)
32	8.406107 (19805)	8.558563 (19916)
64	8.394203 (19805)	8.565195 (19916)
128	8.394159 (19805)	8.562974 (19916)
256	8.393343 (19805)	8.591214 (19916)
512	8.390172 (19805)	8.641762 (19916)

原因を調査すべく，プロセス単位での実行時間の調査を行った．表 2.4 に各次元の分割数を変動させた場合の通信時間を除いた実行時間の中で，最小と最大の演算時間であったプロセスを示す．また括弧内の値は，そのプロセスが演算を行った格子点数を示す．結果として，トロイダル方向や粒子数の分割数を変動させた場合は，各プロセスが演算する格子点数に差はなく，それぞれのプロセスの演算時間はほぼ同じである．しかし，径方向の分割数を変動させた場合には，プロセス数の増加とともに各プロセスが演算する格子点数に大きな差が生じている．図 2.8 に径方向の分割数増加時の 512 プロセス実行における，演算時間の最小と最大のプロセスの内訳を示す．図 2.8 より *smooth*，*field*，*poisson*，*push* 及び *charge* の実行時間が増加していることがわかる．これらの演算は 2.2.2 節より格子点に関係する演算を行う関数であるこ

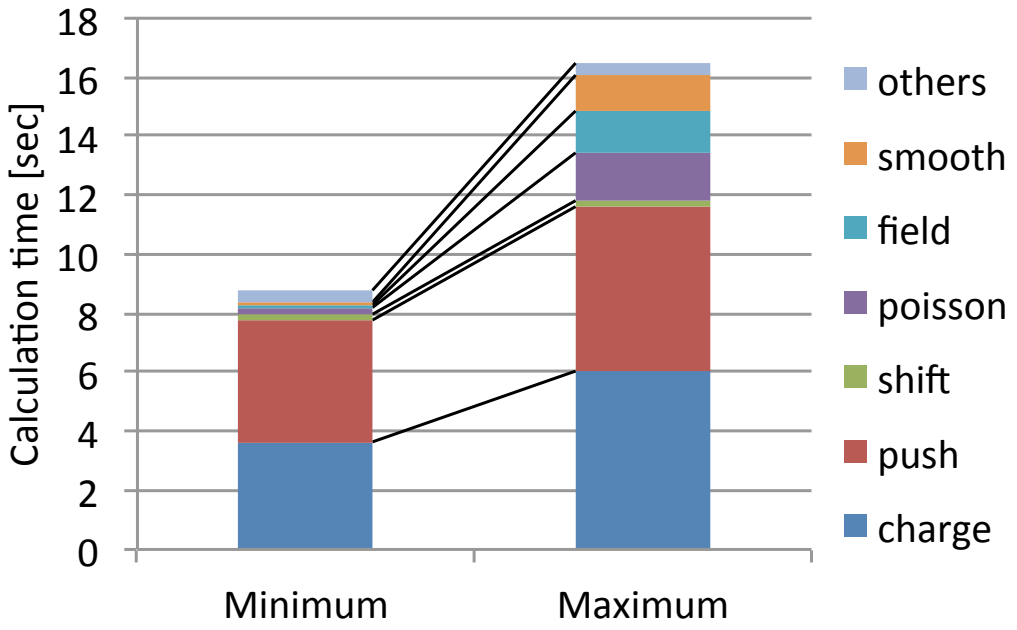


図 2.8　MPI 実装の 512 プロセス実行時における径方向分割時の演算時間の内訳.

とから，格子点数の増加により演算時間が増加していると言える．従って，GTC-P のアプリケーションの設計として，格子点数の均等な分割ができておらず，各プロセスでの演算量の違いによるロードインバランスの発生により性能が低下していることがわかった．

径方向分割時の全ての実装において性能が低下しているが，それ以上に XMP ローカルビュー，ハイブリッドビュー実装の通信時間が長くなっていることが図 2.6 よりわかる．図 2.9 に各次元の分割数を変動させた場合のプロセス 0 の通信回数を示す．図 2.4 より 64KB を境に MPI と XMP の通信性能に差が生じるため，64KB を境界とした通信回数を示す．図 2.9 より，トロイダル方向と粒子数の分割数変動時はプロセス数増加により通信分布に変化はないが．径方向分割時にはプロセス数増加とともに通信回数が増加している．径方向分割時において 64KB を境とした両方の通信回数が増加しているが，図 2.4 より 64KB 以上の MPI と XMP の通信性能の差が大きいため，XMP ローカルビュー，ハイブリッドビューと MPI 実装で通信時間に差がでていると考えられる．

XMP のローカルビューとハイブリッドビュー実装においても図 2.6 より，通信時間に差があることがわかる．GTC-P では，ステンシル演算を行う格子点において，隣接プロセスが持つ格子点を保持する袖領域の一部分のみを更新する通信が存在する．しかし，ハイブリッドビューで用いられているグローバルビューの `reflect` 指示文では，次元単位，もしくは各次元の片側一方のみを通信することは可能だが，各次元の片側一方内の一部分のみを更新することはできない．そのため，MPI やローカルビュー実装と比較して多くの通信を行っている．以上のことがローカルビューとハイブリッドビュー実装の通信時間の差であると考えられる．

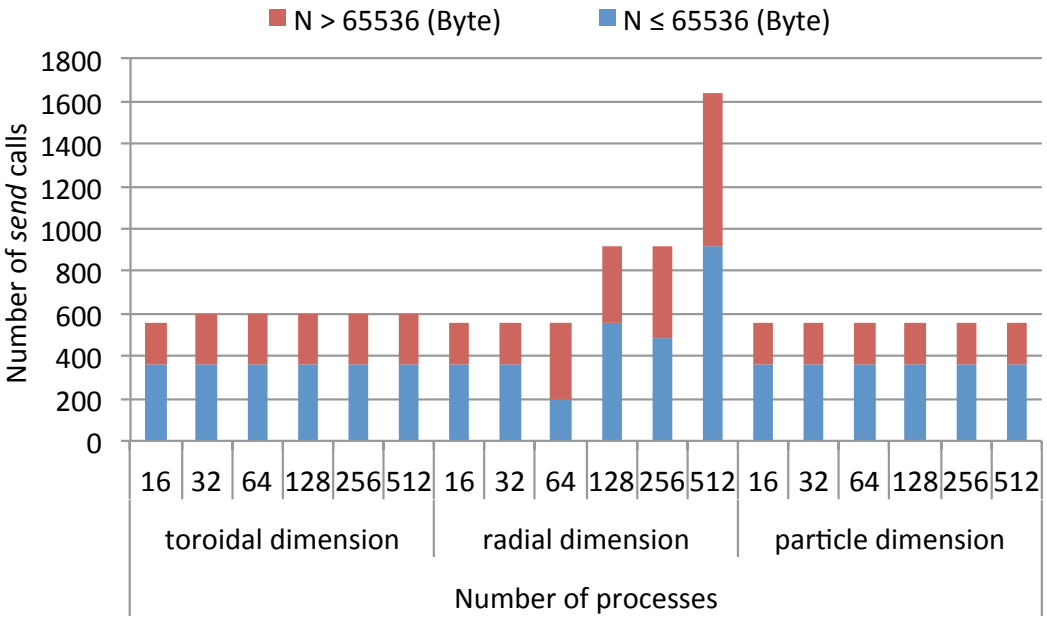


図 2.9 ウィークスケーリングにおける分割数変動時の通信回数の変化.

ストロングスケーリングの評価

図 2.10，2.11 に径方向，粒子数の分割数を変動させた場合のストロングスケーリングによる評価を示す．トロイダル方向の分割数を変動させた場合のストロングスケーリングの評価は，トロイダル方向の分割数を増加させると演算量も増加するというアプリケーションの特性上，評価をすることが不可能であるため，2 種類の評価とした．ウィークスケーリングによる評価と同様に，棒グラフは実行時間，折れ線グラフは MPI 実装の性能を 1 としたときのローカルビュー，ハイブリッドビュー実装の相対性能を表す．

ローカルビュー，ハイブリッドビューによる実装は，図 2.11 よりどちらの実装においても，粒子数の分割数を変動させた場合は，MPI とほぼ同等の性能が得られている．一方で，図 2.10 の径方向の分割数を変動させた場合は，64 プロセス以上で実行プロセス数を増加させる毎に MPI 実装よりも性能が向上している．実行時間の内訳より計算時間は MPI とほぼ同等だが，通信時間が短くなっていることがわかる．ウィークスケーリングのプロセス 0 の通信回数の調査と同様に，ストロングスケーリングにおいても 64KB を境とした通信回数を各次元の分割数増加時のそれぞれの場合において調査を行った．結果を図 2.12 に示す．図 2.12 より，粒子数の分割数変動時はプロセス数を増加しても通信回数の分布に差はない．しかし，径方向の分割数増加時は，プロセス数の増加とともに 64KB 以下の通信回数が大幅に増加している．図 2.4 より，XMP の通信性能は MPI と比較して 64KB 以下の場合に性能が良いため，通信時間が短くなり性能が向上したと考えられる．

XMP/MPI+OpenMP 実装の評価

図 2.13 に，XMP/MPI+OpenMP 実装による性能評価を示す．分割数は $2 \times 8 \times 2$，$2 \times 2 \times 8$ の 2 種類である．径方向の分割数が 8 の場合，ローカルビューは 10%，ハイブリッドビューでは 17% 程度 MPI によ

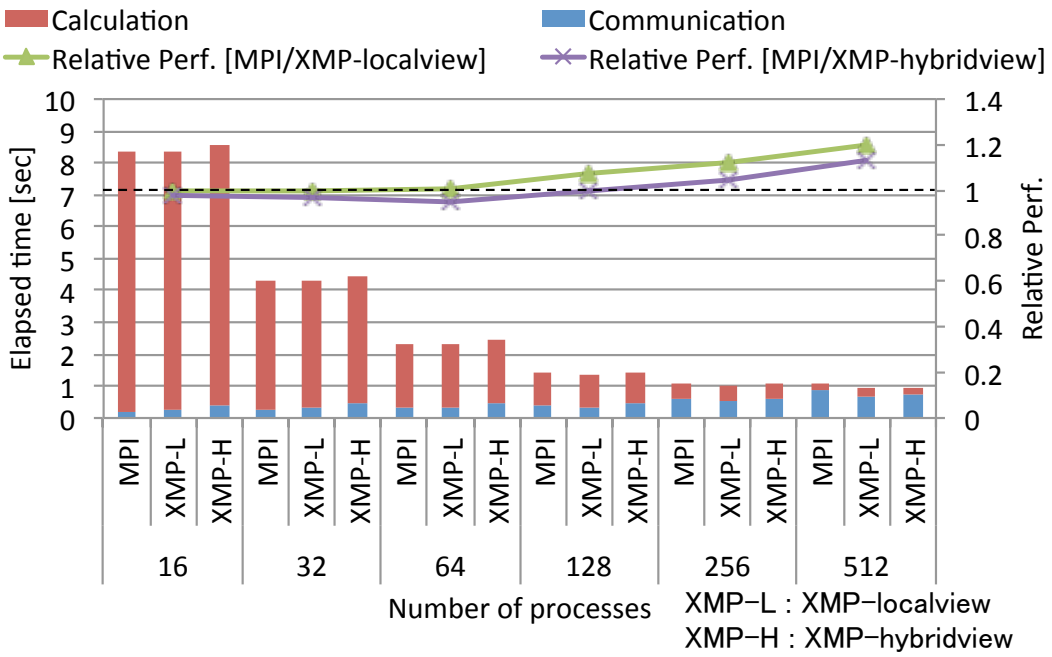


図 2.10　ストロングスケーリングの性能評価（径方向の分割数を変動）.

表 2.5　GTC-P の MPI，XMP 実装の Delta-SLOC.

	Serial	MPI	XMP	
			localview	hybridview
SLOC	4110	5427	5398	5179
modified	-	170	168	158
added	-	1319	1303	1112
deleted	-	2	15	43
Total Delta-SLOC	-	1491	1486	1313

る実装と性能差がある．また，粒子数の分割数が 8 の場合，ローカルビューは 2%，ハイブリッドビューでは 15% の性能差である．以上より，XMP+OpenMP 実装は，各方向の分割数を増加させた場合においても，ローカルビュー，ハイブリッドビューともに性能が極端に低下することはなく，MPI+OpenMP 実装と同様にスケールすることがわかった．

2.5.4　生産性の評価

1.2.2 節で述べた通り，PGAS モデルのグローバルビューによる並列実行を行うことで，逐次プログラムからの少ない変更で並列プログラミングをすることが可能である．また，逐次プログラムの形状を残したままでプログラミングが可能なため，データ分散や並列実行によるループインデックスなどの変更が必要なく，通信も暗黙もしくはグローバルな名前空間上のデータコピー記述となる．従って，並列実装のた

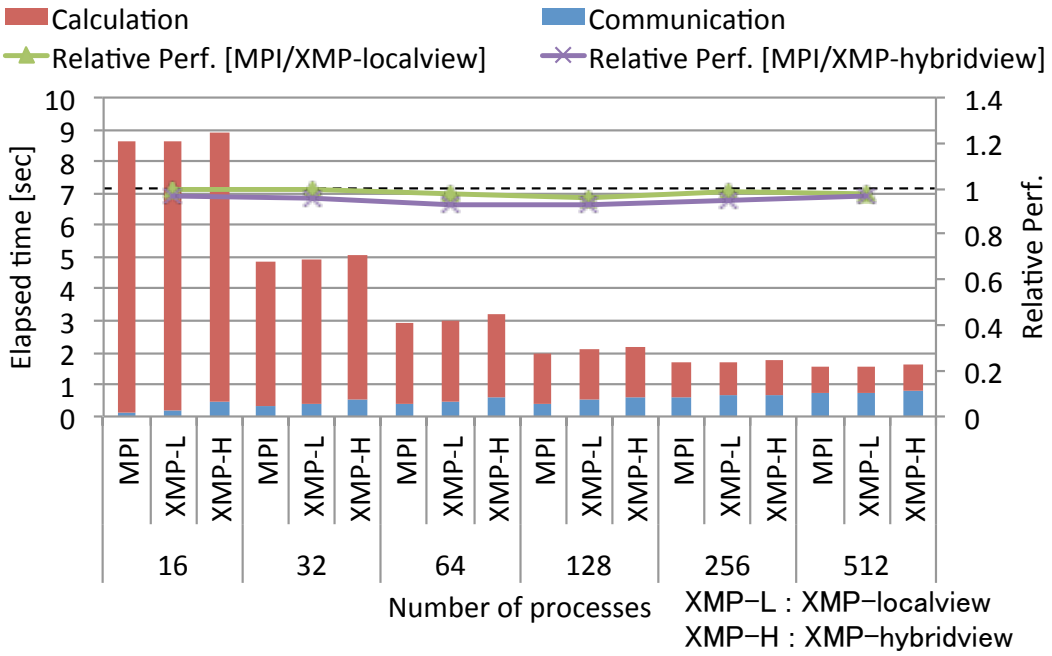


図 2.11 ストロングスケーリングの性能評価（粒子数の分割数を変動）.

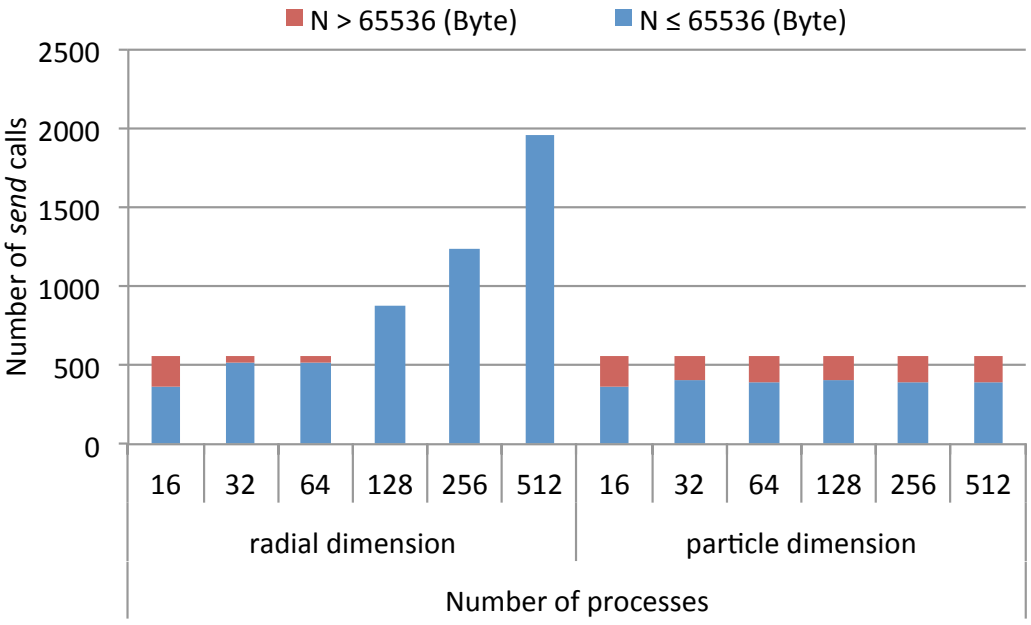


図 2.12 ストロングスケーリングにおける分割数変動時の通信回数の変化.

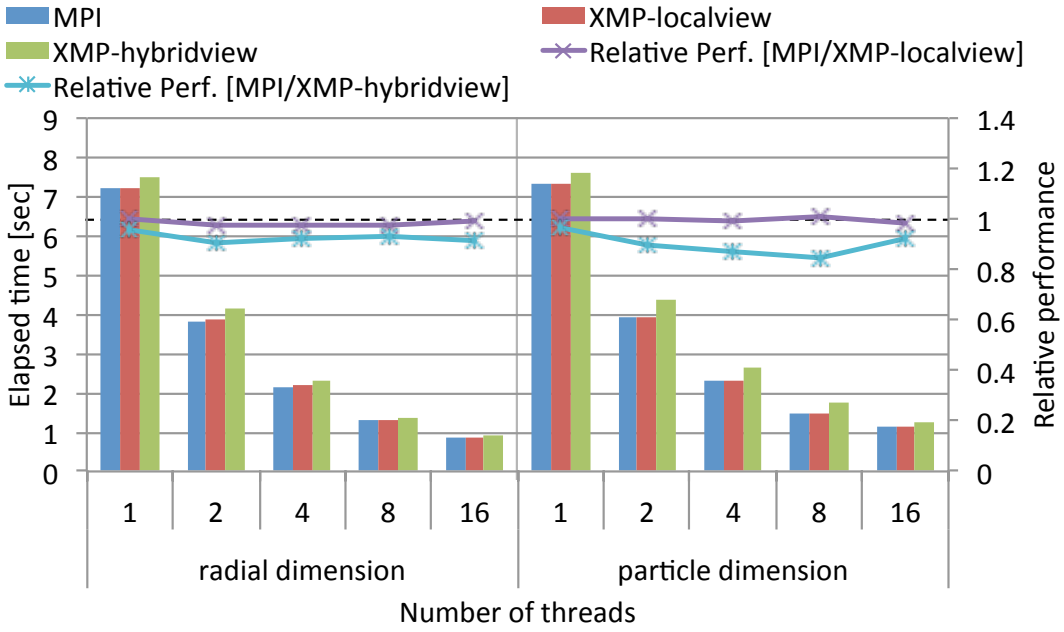


図 2.13　XMP/MPI+OpenMP 実装の評価．分割数を $2 \times 8 \times 2$，$2 \times 2 \times 8$ とした場合の実行時間.

めに逐次プログラム自体の変更は最小限であるため，プログラムの開発コストやバグの発生を低くすることができる．しかし，計算科学の分野で実装される実アプリケーションは複雑であり，全てのプログラムに対してグローバルビューを適用することは難しい．その場合，出来る限りグローバルビューで実装し，グローバルな名前空間上での実装が困難な箇所のみローカルな名前空間を用いて実装をすることで生産性の高い実装とすることが可能であると考える．また，ローカルな名前空間で実装が必要な箇所においてはデータ分散や並列実行は MPI と同様の記述が必要となるが，PGAS モデルのローカルビューを使うことで通信記述の生産性を向上させることが可能である．従って，グローバルビューとローカルビューを組み合わせたハイブリッドビューによる記述が必要となる．

GTC-P の XMP 実装における生産性の考察を述べる．提案モデルであるハイブリッドビューによる実装では，演算領域の分割を XMP グローバルビューの指示文で実装を行い，並列実行する for ループには逐次プログラムに対して `loop` 指示文の追加のみの実装とした．また，隣接プロセスが持つ格子点の参照には，隣接格子点を保持するバッファを `shadow` 指示文を用いて袖領域として確保し，`reflect` 指示文による指示文 1 行での袖領域通信を行う実装とした．このように，指示文は入るものの逐次プログラムには大きな変更がないため，プログラムの可読性は高いと言える．また，グローバルビューの指示文によるデータ分散としたため，データ分散の範囲や分散方法の変更による性能チューニングを簡易に実現可能とした．XMP は指示文ベースなプログラミングモデルであるため，部分的，段階的な並列実行が可能であり，デバッグが容易と言える．`reflect` 指示文による通信では，不連続領域に対して通信を実行する場合，XMP ランタイム内で *MPI_Type_vector()* による派生データ型かパッキング/アンパッキングによる通信が自動で行われる．従って，ユーザは指示文 1 行を記述するだけで袖領域通信が実行され，XMP ランタイムが自動でデータや通信の整合性をとるため，通信のプログラミングミスを減らすことが可能である．

粒子軌道計算では，各プロセスが持つ格子内を自由に動き回る粒子演算を伴うため，グローバルな名前空間を用いた実装が困難である．そこで，粒子軌道演算のみをローカルな名前空間での実装とし，MPI と同様にデータ分散や並列実行を記述する実装とした．粒子が別プロセスへと移動する通信は，ローカルビューの coarray による片側通信としたことで，配列代入文形式での見通しの良い通信記述とした．coarray を用いることで，MPI のように通信マッチングのためのタグの管理やデータ型の指定，コミュニケータや通信情報の管理のための変数を XMP ランタイムが自動で生成，管理を行うため，プログラムから煩雑な記述を排除しユーザに求められる通信のための変数管理コストを減らすことが可能である．

ハイブリッドビューによる GTC-P 実装の生産性を定量的に評価するため，各プログラミングモデルによる実装のコード行数を比較する．比較手法として逐次プログラムからの差分（修正，追加及び削除の行数）で評価を行う Delta-SLOC 方式 [42] を用いる．表 2.5 にそれぞれのプログラミングモデルによる GTC-P の行数と逐次プログラムからの差分を示す．全体の行数を MPI と比較すると，ローカルビュー実装とはほぼ差はなく，ハイブリッドビュー実装では約 250 行削減した．この理由として，ローカルビュー実装は MPI の通信を coarray としただけであり，ハイブリッドビュー実装は指示文によるデータ分散や `reflect` 指示文 1 行による袖領域通信による実装としたためである．Delta-SLOC の評価より，ハイブリッドビュー実装ではグローバルビューの指示文による実装により，修正，追加行数が減少していることがわかる．しかし，依然として追加行数が多い理由として，ローカルな名前空間で記述された粒子軌道演算が追加行数の多くを占めているためである．また，表 2.5 よりローカルビューとハイブリッドビュー実装の削除行数が増加していることがわかる．ローカルビューやハイブリッドビューで用いられる coarray のバッファは，XMP の仕様上，静的にグローバル領域に確保しておく必要がある．しかし，MPI 実装では動的に確保しているため，バッファ毎に確保と解放の行数が削除されているためである．ハイブリッドビューにおいては分散配列も静的な確保としているためローカルビューと比較して削除行数が増加している．分散配列はテンプレートや分散配列を動的に確保可能な `template_fix` 指示文や *xmp_malloc()* が実装されたため，今後の課題として動的確保した分散配列による実装と評価を行う予定である．

2.6 関連研究

PGAS モデルによる PIC 法の実装例を示す．R. Preissl らは文献 [43] にて，3 次元 PIC コードである Gyrokinectic Tokamak Simulation（GTS）を PGAS+OpenMP で実装し，オリジナルの実装と比較して高い性能を示した．GTS は MPI+OpenMP で実装されており，MPI 通信の一部を CAF による片側通信の実装とした．対象は粒子移動時の通信であり，オリジナルの P2P 通信による実装から片側通信への変更と通信アルゴリズムの最適化を通して高速化を行った．CAF のコンパイラとして Cray Fortran 2008 を使用し，Opteron 6172 を搭載する Cray XE6 上で最大 13056 プロセスを用いた評価によると，オリジナルの実装と比較して 52% 性能が向上したと報告されている．MPI 実装の一部をローカルビューの CAF で部分的に実装しているため，CAF の配列代入文形式の見通しの良い記述による通信実装ではあるが，データ分散や並列実行は MPI と同様陽に記述されているため，プログラム全体の生産性が高いとは言えない．

H. Sakagami らは文献 [44] にて，2 次元 PIC コードである ESPAC2 を HPF による実装と性能評価を示している．データ分散は粒子と電磁場の 2 次元分散としており，粒子情報はブロック分割だが電磁場は各プロセスがデータを重複して持つ．各プロセスが更新する粒子情報により演算された電磁場を，集合通信

によりデータを集約し全プロセスの値を更新する．そのため，データ分散や通信は単純なブロック分割や集合通信となるため，グローバルビューに適したアルゴリズムと言える．VPP800，SR8000，SX-4 及び SX-5 上で各ベンダーが提供する HPF コンパイラによる性能評価を行っており，コンパイラのバージョンによる通信性能の低下は見られたが，概ね 8 プロセスまでは良い性能が得られていると報告されている．グローバルビューモデルである HPF を用いた実装により，大域的な名前空間による逐次実装とほぼ同等の記述で並列実装が施されている．しかし，グローバルビューで実装するために重複したデータ領域が必要となるなど，大規模実行時のスケーラビリティに問題が生じる可能性がある．そこで，スケーラビリティを向上させつつもプログラムの生産性を維持するために，本研究で提案するハイブリッドビューのように，部分的にグローバルビューを用いることが可能なプログラミングモデルが必要であると考えられる．

GTC や GTC-P は様々なプラットフォーム向けの実装，最適化が行われている．下坂らは文献 [45] にて京コンピュータ上での GTC-P の大規模実行や性能解析ツール Scalasca によるコード解析結果を示している．京コンピュータによる性能評価では，本研究とは異なり全ての分割次元数を同時に増加させた場合の評価であり，合計で 512，2048，8192 及び 32768 ノードを用いた 4 種類のウィークスケーリングの性能を示している．結果として，本研究での性能評価と同様に性能がスケールしていない．Scalasca によるコード解析の結果によると，ノード毎にロードバランスが取れておらず，通信待ち時間により性能が低下していると報告されており，本研究の考察でも同様に示されている．X. Liao らは文献 [46] にて Tianhe-2 上で様々な HPC アプリケーションの実装，性能評価を報告している．GTC がその中の一つであり，Tianhe-2 が持つ Intel Xeon Phi（KNC）向けにオリジナルの MPI+OpenMP の実装を，オフロードモデルによる実装へと修正を行った．Tianhe-2 の 1 ノードは 2 ソケット構成の Intel Xeon E5-2692v2 と 3 枚の KNC により構成されており，性能比較は 2 ソケットの Xeon 対 KNC としている．結果として 2 ソケットの Xeon と 3 枚の KNC を比較すると，1.67 倍の性能向上が報告されている．K. Madduri らは文献 [47] にて GPU 向けの GTC の実装と性能評価を示している．Fermi 世代の GPU（Tesla C2050）向けに CUDA を用いた実装が行われ，オリジナルの MPI+OpenMP の実装と比較して 1.34 倍の性能向上を報告している．筑波大学と理化学研究所では XMP とデバイス向け指示文ベースモデル OpenACC を垂直統合した XcalableACC（XACC）[48] の仕様検討及びプロトタイプ実装が進められている．今後の課題として，XACC を用いてデバイス向けの実装を行い，GPU や Intel Xeon Phi における性能や生産性の評価を行うことが挙げられる．

2.7　まとめ

大規模並列クラスタにおける並列プログラムの生産性の向上を目的として，PGAS モデルを基にしたハイブリッドビューの提案を行った．ハイブリッドビューは，グローバルビューの簡易なデータ分散，並列実行及び通信・同期を記述可能としつつ，ローカルな名前空間でのプログラミングが求められるような複雑な通信に対してはローカルビューの簡易な記述による片側通信を記述可能とする．提案モデルにより，従来ではグローバルビューの適用が困難なプログラムに対しても部分的に適用可能となり，プログラム全体としての生産性を向上させることが可能となった．本研究では，PGAS 言語 XMP を対象とし，XMP のグローバルビューとローカルビューを組み合わせたハイブリッドビューを用いて核融合シミュレーショ

ンコード GTC-P の実装を行い，オリジナルの MPI 実装と比較をすることで性能と生産性の評価を行った．ハイブリッドビューによる実装では，オリジナルの実装を含むロードインバランスが発生する一部の評価を除き，MPI 実装に近い性能を達成した．生産性の観点からは，グローバルビューによる領域分割により，逐次プログラムに指示文を追加するのみでの並列化や，隣接格子点間の通信を `reflect` 指示文 1 行で記述することが可能であるため簡易な実装と言える．また，ローカルビューの coarray は，配列代入文形式で通信を記述可能なことから，MPI と比較してより直感的なため可読性が高く，XMP が自動で通信の整合性をとるため通信記述も容易である．以上のことから，PIC 法に含まれる各プロセスの演算量が動的に変化する粒子軌道演算のような，グローバルビューのみで実装することが困難な複雑なアルゴリズムに対しても，XMP のプログラミングモデルを組み合わせることで実装が可能になり，さらに，一定の性能を保ちつつ，簡便かつスケーラブルに記述できる事が示された．

今後の課題として，XMP/C の coarray のランタイムライブラリの実装に別の片側通信ライブラリを用いて実装し，性能評価をすることが挙げられる．Omni XMP Compiler では，2018 年 1 月現在 coarray のランタイムライブラリの実装に MPI を用いた実装が追加されており，それを用いた評価や，GASPI[49] や ComEx[50] などの他の PGAS 向けの通信ライブラリを用いて実装することが挙げられる．また，本研究では，分散配列は全て静的に確保された配列を対象としたため，`template_fix` 指示文や *xmp_malloc()* を用いた動的な分散配列を用いた実装を行うことが挙げられる．性能評価で用いたサイズは A と小規模であったため，ITER のような大規模な核融合装置に対応した問題サイズを解いた場合の評価や，XACC を用いた GPU クラスタ向けの実装を行うことが考えられる．

第 3 章

PGAS モデルにおけるタスク並列プログラミング

3.1　タスク並列プログラミング

本研究では OpenMP のタスク並列モデルに基づき，XMP における分散メモリ環境向けのタスク並列モデルを提案する．本節では，OpenMP のデータ依存に基づくタスク並列モデルの詳細を説明し，分散メモリ環境におけるノードを跨るタスク間の依存関係の実現方法について述べる．

3.1.1　OpenMP のタスク並列プログラミング

OpenMP には様々なノード内並列化やデバイス利用のための機能が指示文により提供されているが，本節では本研究で用いる `task` 指示文の概要とその記述方法のみを示す．`task` 指示文は OpenMP の仕

```
#pragma omp parallel
#pragma omp single
{
   int A, B, C;
#pragma omp task depend(out:A)
   A = 1;         /* taskA */
#pragma omp task depend(out:B)
   B = 2;         /* taskB */
#pragma omp task depend(in:A, B) depend(out:C)
   C = A + B;  /* taskC */
#pragma omp task depend(out:A)
   A = 3;         /* taskD */
}
```

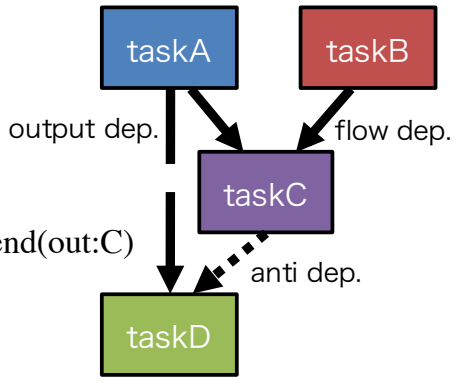


図 3.1　OpenMP `task` 指示文の例．

様 3.0 から登場したタスク並列を記述可能とする指示文であり，再帰的構造（フィボナッチ数の計算や分割統治法など）や while ループなどの各スレッドでの演算が動的に決定する場合に用いられる．また，仕様 4.0 から登場した `depend` 節により，タスクが用いる演算データを基にしたデータ依存を記述することで，全体同期ではなくタスク間の細粒度な同期を記述することが可能となった．`depend` 節には *dependence-type* として *in*，*out* 及び *inout* の 3 種類が指定可能であり，合わせて変数または配列セクションを指定する．以下に OpenMP によるタスク依存モデルの特徴を示す．

- 配列セクションを記述した場合は，配列セクションとその範囲内の単一に記述された要素全てと依存関係があるのではなく，完全一致する配列セクション同士で依存関係が発生する．
- 配列セクションは依存関係毎に独立した範囲を記述する必要があり，範囲を重ねた依存関係を持つタスクを記述してはいけない．
- 依存関係を示す変数や配列セクションは必ずしもタスクブロック内で用いる必要はない．
- タスクの依存関係は兄弟タスク間でのみ考慮され，親子タスク間では無視される．

`depend` 節により生成される依存関係には，プログラムの逐次実行に基づく 3 種類のデータ依存が存在する．

- フロー依存: 同一変数に対する書き込み後の読み込み（RAW: read-after-write）．`depend` 節の *out* と *in* の間に発生する．
- 反依存: 同一変数に対する読み込み後の書き込み（WAR: write-after-read）．`depend` 節の *in* と *out* の間に発生する．
- 出力依存: 同一変数に対する書き込み後の書き込み（WAW: write-after-write）．`depend` 節の *out* と *out* の間に発生する．

本研究で用いるタスク並列モデルは，OpenMP の `parallel` + `single/master` 指示文ブロック内で `task` 指示文が実行されるモデルであり，あるスレッドがタスクを生成し実行待機中のスレッドが生成されたタスクを並列に実行する．`parallel`，`single` 指示文ブロックの出口では暗黙の同期が入るが，任意地点でのタスクの同期や，`single` 指示文ではなく `master` 指示文の場合は，明示的なタスク同期を実行する `taskwait` 指示文が必要となる．

図 3.1 に `task` 指示文の例を示す．*taskA*，*taskB*，*taskC* 及び *taskD* が `task` 指示文により生成され，タスク毎に `depend` 節で指定された依存関係を持つ．`depend` 節による依存関係の記述は基本的にタスク内で使用される変数や配列に対して記述されるのが一般的である．例えば *taskA* と *taskB* の場合は，変数 *A* と *B* に対して書き込みを行うため `depend` 節にて *“out:A”*，*“out:B”* と指定する．*taskC* は変数 *A* と *B* に対して読み込みを行い，それらの値の加算結果を変数 *C* に対して書き込むため，`depend` 節にて *“in:A，B”*，*“out:C”* と記述する．このように依存関係を記述した場合，*taskA* と *taskB* の間には依存関係が無いため並列に実行される．*taskB* と *taskC* の間には変数 *B* によるフロー依存，*taskC* と *taskD* の間には変数 *A* による反依存，*taskA* と *taskD* の間には変数 *A* による出力依存がそれぞれ発生し，各依存関係が解消されるまで後続のタスクの実行は開始されない．

```
int A[2];

if (proc == 0) {
#pragma omp task depend(out:A[0])
   funcA(A[0]);                    /* taskA */
#pragma omp task depend(in:A[0])
   MPI_Send(A[0], ... 1, ...);   /* taskB */

} else if (proc == 1) {
#pragma omp task depend(out:A[0])
   MPI_Recv(A[0], ... 0, ...);   /* taskC */
#pragma omp task depend(in:A[0]) depend(out:A[1])
   funcB(A[0], A[1]);              /* taskD */
}
```

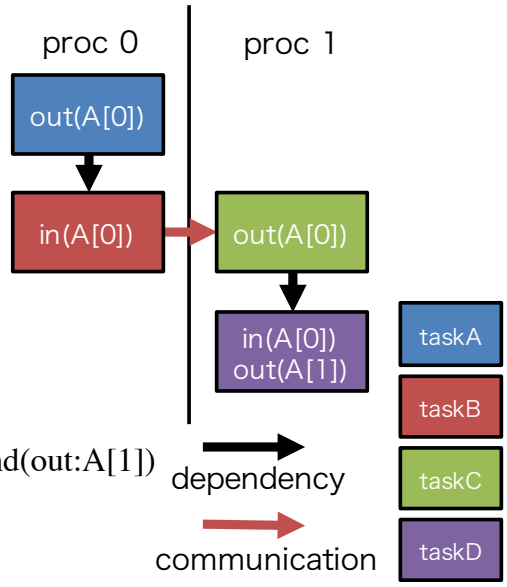


図 3.2　MPI+OpenMP による分散メモリ環境におけるタスク依存のプログラミング例.

3.1.2　分散メモリ環境での OpenMP タスク並列プログラミングモデル

OpenMP は共有メモリ向けのプログラミングモデルであり，分散メモリ環境で実行する場合には MPI など他のプログラミングモデルと組み合わせた記述が必要となる．OpenMP のタスク並列モデルを分散メモリ環境へと拡張する場合，ノードを跨るタスクの依存関係をどのように記述するかを考慮する必要がある．OpenMP の場合，タスク依存の処理は逐次実行に基づいたデータの read/write によるデータ依存であるため，分散メモリ環境における異なるノードのスレッド上で並列実行されているタスク間で依存関係を構築することは困難である．そこで，本研究では MPI の P2P 通信を OpenMP のタスク内から実行し，MPI の通信を依存関係とする手法を用いる [52]．送信側では *MPI_Send()* の通信対象を `depend` 節で *in* とし，受信側では *MPI_Recv()* の通信対象を *out* としてデータ依存を指定する．送信側では反依存が発生するため送信完了までデータの書き換えはできず，受信側ではフロー，出力依存が発生するため対象データを用いるタスクの実行は待機される．

図 3.2 に分散メモリ環境における MPI と OpenMP を用いたタスク依存によるプログラミング例を示す．例では，*funcA()* により *A[0]* を書き換え，その値を用いて *funcB()* にて *A[1]* を更新する．実行プロセス数は 2 であり，*funcA()* をプロセス 0，*funcB()* をプロセス 1 が実行するため，*A[0]* をプロセス間で通信する必要がある．タスクは 4 種類生成され，*funcA()* にて *A[0]* を更新する *taskA*，*A[0]* をプロセス 1 へ送信する *taskB*，プロセス 0 より *A[0]* を受信する *taskC* 及び *funcB()* で *A[0]* を用いて *A[1]* を更新する *taskD* である．プロセス 0 は，*taskA* が *A[0]* を書き換えるため `depend` 節に *out* で指定する．*taskB* は *MPI_Send()* で送信が完了するまで対象である *A[0]* への書き込みを防ぐため，*in* に指定することで後続タスクとの間に反依存を生成する．プロセス 1 は，*taskC* にてプロセス 0 から *A[0]* を *MPI_Recv()* で受信するため，受信バッファである *A[0]* への書き込みがあるとして *out* を指定することで，後続タスクとの間

にフロー，出力依存が生成される．*taskD* では受信した *A[0]* を用いて *A[1]* を更新するため，*in* に *A[0]*，*out* に *A[1]* を指定する．以上より，通信が完了するまでプロセス 0 では *A[0]* は更新されず，プロセス 1 はプロセス 0 が持つ *A[0]* の受信が終わるまで，その値を用いる演算タスクの実行は開始されない．つまり，通信対象を依存関係とした P2P 通信をタスク内で実行することで，プロセス間のタスクの依存関係を記述することが可能である．

MPI と OpenMP を組み合わせることでプロセス間のタスク依存を記述することができる一方で，MPI によるデータの分散を考慮した複雑な通信とタスクの依存関係を合わせた記述が必要とされる．図 3.2 の例を見ても，実行プロセス指定のための if 文や MPI の通信記述があり，オリジナルの逐次プログラムから掛け離れた記述となっている．また，この例では配列 *A* は全てのノードで重複して確保されているが，一般的な MPI による並列プログラミングではデータは分散配置されるため，ローカルインデックスによる演算記述や通信用のローカルバッファが必要となるなど，プログラムはさらに複雑になる．また，OpenMP と MPI は基本的には別の実装となっており，OpenMP ランタイム側からはタスク内で実行される演算と MPI 通信を区別しない．そのため，タスク内で MPI のブロッキング通信や同期処理を行った場合，OpenMP ランタイムが依存関係は無いと判断すれば通信タスクが大量に実行され，デッドロックを引き起こす可能性がある．従って，デッドロックを起こさない通信タスクの実装も必要とされる．

3.2 XMP のタスク並列プログラミングモデル

分散メモリ環境におけるタスク並列プログラムを簡易に記述可能とすべく，PGAS 言語 XMP に拡張を行う．XMP にタスク並列モデルを取り入れることで，PGAS モデルのグローバルビューによる簡易なデータ分散，並列実行及び通信・同期や，ローカルビューのスケーラブルな記述による通信をタスク並列においても記述可能とし，プログラム全体での生産性を向上させることが目的である．また，従来のループ並列とタスク並列による実装をメニーコア環境で比較することで，メニーコア環境におけるタスク並列実装の優位性も示す．

3.2.1 設計

XMP におけるタスク並列をサポートする指示文として，既に `task` 指示文が仕様にある．`task` 指示文は，ブロック内に記述された演算を `on` 節で指定されたノードのみが実行する．この指示文はノードレベルで実行され，`tasks` 指示文ブロック内に `task` 指示文が記述されない限り，複数の `task` 指示文による処理は別ノード上の実行であっても並列に実行されない．本研究では，各ノードが持つスレッド上で実行される細粒度なタスクを想定しており，ノード間で並列に実行されるのはもちろん，スレッドレベルでの並列実行も求められる．そこで，本研究では細粒度タスクを実行するための新たな指示文として，`tasklet` 指示文を提案する．

図 3.3 に `tasklet`，`taskletwait` 及び `tasklets` 指示文のシンタックスを示す．`tasklet` 指示文は各ノードのスレッド上にタスクを生成する指示文である．実行ノードは `on` 節に記述されるノード集合やテンプレートによって決定され，指定がない場合は全てのノードで同一のタスクが生成される．実行モデルとして OpenMP のタスク依存モデルを採用しており，`in`，`out` 及び `inout` 節と合わせて変数ま

#pragma xmp tasklet *[clause[, clause] ...] [on* { *node-ref* | *template-ref* } *]*
(structured-block)

#pragma xmp taskletwait *[on* { *node-ref* | *template-ref* } *]*

#pragma xmp tasklets
(structured-block)

where *clause* is :
{in | out | inout} (*variable[, variable] ...]*)

図 3.3 tasklet，taskletwait，tasklets 指示文のシンタックス．

たは配列セクションを記述することでタスク依存を記述する．依存関係がないタスクは指示文に到達直後に実行が開始されるが，依存関係がある場合は 3.1.1 節で示したフロー，反及び出力依存を持つ先行タスクの実行が終了するまで，タスクの実行は開始されない．in，out 及び inout 節に記述されるタスク依存はノード内に閉じられており，グローバルビューの分散配列で自ノードが持たない領域が依存関係として指定された場合も，全てノード内依存として処理される．

tasklets 指示文は，そのブロック内に記述されたタスクをスレッドレベルで並列実行することを示す指示文である．tasklets 指示文ブロック内に記述された tasklet 指示文のみ生成されたタスクが並列に実行され，tasklets 指示文が記述されていない場合，タスクは各ノードで逐次に実行される．taskletwait 指示文は，生成されたタスクの実行完了を保証する指示文である．この構文はノード毎に生成されたタスクの同期をとる指示文であるため，実行ノード全体で同期をとる場合は barrier 指示文と合わせて記述する必要がある．また，tasklets 指示文のブロックの出口にて各ノード内におけるタスクの暗黙の同期を含む仕様とした．

ノード間のタスクの依存関係は，3.2 節より P2P 通信により表される．XMP ではグローバルビュー，ローカルビューともに様々な通信構文（指示文，coarray）を提供している．本研究では，どちらのモデルにおいてもノード間のタスク依存の記述を可能とするため，各モデルにおける通信構文をタスク実行向けに拡張を行う．本研究で示す tasklet 指示文や XMP の通信指示文の拡張は，XMP の次期仕様である XMP2.0 に向けて，現在 PC クラスタコンソーシアム XMP 規格部会によって検討が進められている．本研究における拡張はその提案の一つである．

グローバルビューでのノード間タスク並列記述

グローバルビューモデルでは，分散配列に対する通信を記述可能な gmove，reflect 指示文をタスク並列で記述可能とすべく拡張を行う．図 3.4 にタスク上で実行される gmove，reflect 指示文である tasklet gmove，tasklet reflect 指示文のシンタックスを示す．

tasklet gmove 指示文は on 節により実行ノードが決定され，記述された配列代入文に現れる分散配列によって通信対象が決定される．また，on 節が無い場合は，配列代入文に記述された分散配列を持

#pragma `xmp tasklet gmove` *[clause[, clause] ...] [on* { *node-ref* | *template-ref* } *]*
(an assignment statement)

#pragma `xmp tasklet reflect` *(array-name[, array-name] ...)*
[`chunksize` *(reflect-chunksize[, reflect-chunksize] ...)]*

where *clause* is :
{`in` | `out` | `inout`} (*variable[, variable] ...]*)

図 3.4 `tasklet gmove`，`tasklet reflect` 指示文のシンタックス．

つノードのみが実行対象となり，ノードローカルな値が指定された場合は全てのノードが実行対象となる．タスク依存は，`tasklet` 指示文同様に `in`，`out` 及び `inout` 節で依存関係を記述することが可能である．`tasklet`，`tasklet gmove` 指示文により生成されたタスクは兄弟タスクとなり，ユーザが記述した依存関係により実行順序が決定される．

`gmove` 指示文は，実行ノード集合全体か `task` 指示文で指定された実行ノード集合内において，配列代入文や実行ノード集合を基に通信を実行するノードが決定される．1.4 節の図 1.6 の通信が実行可能であり，通信完了後には実行ノード集合全体で暗黙の同期がとられる．実行ノード集合全体での同期は，P2P 通信であれば問題はないが，ブロードキャストやリダクションのような集団通信の場合は，データを受信したノードであっても全体で通信が完了するまで実行が止まる．スレッド上のタスクで `gmove` 指示文を実行する場合，全ての通信が完了するまでタスクがスレッド上に残り続ける．通信タスクがスレッドを専有するため，依存関係が解かれたタスクの実行が遅れ，結果としてプログラムの性能に影響を与える可能性がある．そこで，`tasklet gmove` 指示文ではその制約を外し，通信が完了したタスクから実行を終了する仕様とした．

`tasklet reflect` 指示文は，`shadow` 指示文によって袖領域が指定された分散配列に対して，タスク内で隣接ノード間の通信を実行することで袖領域を更新する．XMP では，データ分散は指示文により指定された分散パターンで自動的に各ノードへと分散配置される．つまり，ユーザからは分散されたデータの境界インデックスを知ることはできず，通信される袖領域に対して依存関係を明示的に指定することは不可能である．そこで，`tasklet reflect` 指示文では `tasklet gmove` 指示文と異なり，データ依存は XMP ランタイムによって自動的に生成される仕様とした．

ステンシル演算の最適化手法の一つとしてキャッシュブロッキングがある．キャッシュブロッキングは，演算領域をブロック化しデータアクセス範囲を狭めることでキャッシュミス率を減らし性能を向上させる手法である．XMP プログラムにキャッシュブロッキングを適用すると，XMP のグローバルビューによる指示文で分散されたループをさらにユーザ指定のブロックサイズで分割する．その場合，`tasklet reflect` 指示文により通信される袖領域はブロック化されておらず，ブロック単位での演算と自動で生成される袖領域通信の依存関係が成立しない．そこで，`tasklet reflect` 指示文の節として `chunksize` 節を提案する．`chunksize` 節は依存関係として記述された演算の粒度（この例ではブロックサイズを示す）をチャンクサイズとして指定可能とし，XMP ランタイムに対して自動的に付与さ

#pragma `xmp tasklet` *clause[, clause[, ...]]* on {*node-ref* | *template-ref*}
(structured-block)

where *clause* is :
{`in` | `out` | `inout`} (*variable[, variable, ...]*)
or
{`put` | `get`} (*tag*)
or
{`put_ready` | `get_ready`} (*variable,* {*node-ref* | *template-ref*}*, tag*)

図 3.5 `tasklet` 指示文の `put`，`put_ready`，`get` 及び `get_ready` 節のシンタックス．

れる依存関係がチャンクサイズで分割されていることを知らせる．

ローカルビューでのノード間タスク並列記述

ローカルビューでは，通信構文として coarray を提供している．coarray は片側通信であるため，通信対象のバッファの状況に依らず通信を完了することが可能である．つまり，ノード間のデータの依存関係として coarray 単体で使用することは不可能である．一般的に片側通信を実行する場合，2 種類の同期方法が考えられる．一つは片側通信の実行前後に全体同期をとる方法である．もう一つは，片側通信の実行主体は通信対象ノードより実行可能であるという通知を受け，片側通信を実行する．その後，片側通信の完了を保証した後に完了通知を通信対象ノードに送信する，MPI のアクティブモデルによる同期方法である PSCW（Post-Start-Complete-Wait）モデルに基づく方法である．本研究ではタスク並列による全体同期の削減を目標としているため後者の方法をとる．例えば，タスク内で coarray/Put を実行する場合，Put の通信対象ノードからの通信可能通知，Put，Put の完了通知の 3 種類の通信を記述する必要がある．これら全ての通信をユーザが明示的に記述するのは非常に困難でコードも煩雑になりやすい．そこで本研究では，片側通信を実行するための通知を簡易に記述可能な `put`，`put_ready`，`get` 及び `get_ready` 節を `tasklet` 指示文の節として提案する．片側通信向けの `tasklet` 指示文の拡張のため，ローカルビューの coarray だけではなく，グローバルビューの `gmove in/out` 指示文に対しても同様に使用可能であると考えられる．そこで，3.6 節のベンチマークの実装と評価では，coarray に加えて `gmove in/out` 指示文で記述した実装例も示す．

`put_ready` 節は Put の実行主体の通信対象ノード上で実行され，節には Put を実行される変数または配列セクション，通信対象ノード，タグを記述する．Put によって値の更新が起きるため，この節により生成されるタスクは第一引数に対して *out* の依存関係を持つ．通信対象ノードは `on` 節と同様に実行ノード集合またはテンプレートが記述可能である．`put_ready` 節が記述されたタスクは，通信対象ノードとタグにより指定されたノードのタグと一致する `put` 節が記述されたタスクに対して Put の実行可能通知を送り，通信が完了するまで待機する．通信完了通知を `put` 節が記述されたタスクより受け取った後，`tasklet` 指示文ブロック内に記述された演算の実行を開始する．

```
int A[2], B;
#pragma xmp nodes P(2)
#pragma xmp template T(0:1)
#pragma xmp distribute T(block) onto P
#pragma xmp align A[i] with T(i)

#pragma xmp tasklets
{
#pragma xmp tasklet out(A[0]) on P(1)
   A[0] = 0;       /* taskA */
#pragma xmp tasklet gmove in(A[0]) out(A[1]) on P(1:2)
   A[1] = A[0];   /* taskB */
#pragma xmp tasklet gmove in(A[1]) out(B) on P(2)
   B = A[1];       /* taskC */
}
```

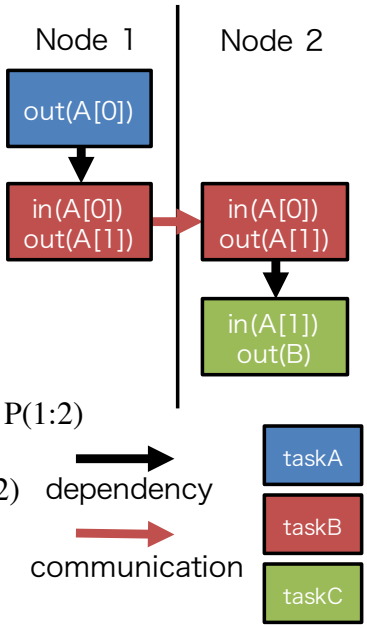


図 3.6 tasklet gmove 指示文のプログラミング例とタスクフロー.

put 節は Put の実行主体のノード上で実行され，節にはタグのみを記述する．タグと一致するノード上の put_ready 節が記述されたタスクより，通信可能通知を受け取るまでタスクの実行は開始されない．通信可能通知を受け取り後，tasklet 指示文ブロック内に記述された片側通信の Put を実行し，片側通信の完了通知を通信可能通知が送られてきた put_ready 節が記述されたタスクに対して送る．

get_ready 節は Get の実行主体の通信対象ノード上で実行され，節には Get を実行される変数または配列セクション，通信対象ノード，タグを記述する．Get によって値の読み込みが行われるため，この節により生成されるタスクは第一引数に対して *in* の依存関係を持つ．通信対象ノードは put_ready 節と同様に実行ノード集合またはテンプレートが記述可能である．get_ready 節が記述された tasklet 指示文ブロック内の処理の実行後に，指定されたノードのタグと一致する get 節が記述されたタスクに対して Get の実行可能通知を送り，通信が完了するまで待機する．通信完了通知を get 節が記述されたタスクより受け取ることでタスクの実行が終了する．

get 節は Get の実行主体のノード上で実行され，節にはタグのみを記述する．タグと一致するノード上の get_ready 節が記述されたタスクより，通信可能通知を受け取るまでタスクの実行は開始されない．通信可能通知を受け取り後，tasklet 指示文ブロック内に記述された片側通信の Get を実行し，片側通信の完了通知を通信可能通知が送られてきた get_ready 節が記述されたタスクに対して送る．

3.2.2 コード例

グローバルビュー

図 3.6 は tasklet gmove 指示文によるプログラミング例とタスクフローである．2 要素を持つ配列 *A* が 2 ノードでブロック分割されており，ノード 1 が *A[0]*，ノード 2 が *A[1]* をそれぞれ保持している．3 種類のタスクが生成され，*A[0]* を更新する *taskA*，tasklet gmove 指示文により *A[1]* から *A[0]* へ

```
int A[4];
#pragma xmp nodes P(2)
#pragma xmp template T(0:3)
#pragma xmp distribute T(block) onto P
#pragma xmp align A[i] with T(i)
#pragma xmp shadow A[1]

#pragma xmp tasklets
{
#pragma xmp loop (i) on T(i)
   for (int i = 0; i < 4; i++) {
#pragma xmp tasklet out(A[i]) on T(i)
      A[i] = i;  /* taskA */
   }
#pragma xmp tasklet reflect(A)
   /* taskB */
}
```

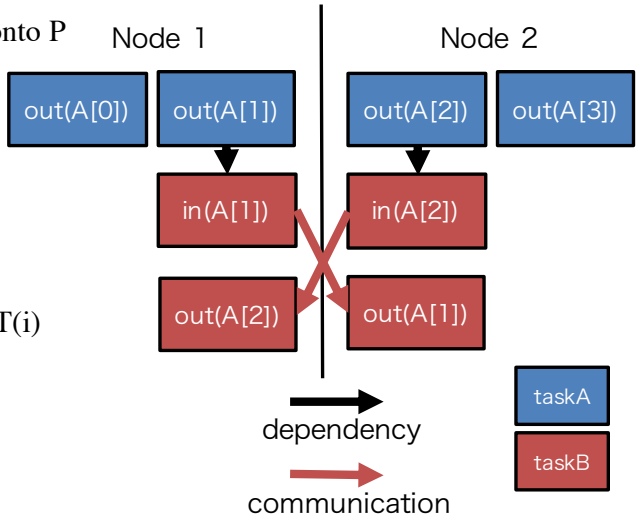


図 3.7　`tasklet reflect` 指示文のプログラミング例とタスクフロー．

とデータコピーを行う *taskB* 及び *A[1]* を用いて変数 *B* を更新する *taskC* である．`tasklet` 指示文は `on` 節により実行ノードが決定されるため，ノード 1 は *taskA* と *taskB*，ノード 2 は *taskB* と *taskC* が生成される．`tasklet gmove` 指示文の配列代入文で指定されている分散配列はそれぞれノード 1 とノード 2 が保持しているため，`gmove` 指示文の通信パターンの一つである *send-recv* による P2P 通信が発生する．ノード 1 で通信を行う *taskB* と *taskA* の間には *A[0]* によるフロー依存があるため *taskA* が終了するまで *taskB* の実行は待機され，ノード 2 では *A[1]* によるフロー依存があるため *taskB* の通信が完了するまで *taskC* の実行は待機される．

図 3.7 に `tasklet reflect` 指示文によるプログラミング例とタスクフローを示す．4 要素を持つ配列 *A* が 2 ノードでブロック分割されているため，ノード 1 が *A[0]*，*A[1]*，ノード 2 が *A[2]*, *A[3]* を保持する．また，`shadow` 指示文の指定により各ノードで 1 要素ずつ袖領域が確保される．2 種類のタスクが生成され，`loop` 指示文によって分散実行されるループが生成する配列 *A* の値を更新する *taskA* と `tasklet reflect` 指示文を実行する *taskB* である．図 3.7 のタスクフローのように 1 次元配列の `tasklet reflect` 指示文による袖領域交換を行う場合，ノード 1，2 がそれぞれ持つ領域を送信するタスクと，袖領域に値を受信するタスクの 4 種類が生成される．従って，*taskA* の中でも隣接ノードと接する領域を演算するタスク（ノード 1 では *A[1]*，ノード 2 は *A[2]* を更新するタスク）の実行が終了次第，隣接ノードとの通信を開始することが可能である．

ローカルビュー

図 3.8 は `tasklet` 指示文の `put`，`put_ready` 節によるプログラミング例とタスクフローである．2 ノードで実行され，ノード 2 の *taskB* からノード 1 の coarray 宣言された変数 *A* に対する coarray/Put を実行する．ノード 1 では，変数 *A* を読み込む *taskA* の実行後に Put が実行され，再度変数 *A* を読み込む

```
#pragma xmp nodes P(2)
   int A:[*], B, C, tag;
#pragma xmp tasklets
{
#pragma xmp tasklet in(A) out(B) on P(1)
   B = A;        /* taskA */
#pragma xmp tasklet out(A) put(tag) on P(2)
   A:[1] = 1;    /* taskB */
#pragma xmp tasklet in(A) out(C) ¥
                 put_ready(A, P(2), tag) on P(1)
   C = A;        /* taskC */
}
```

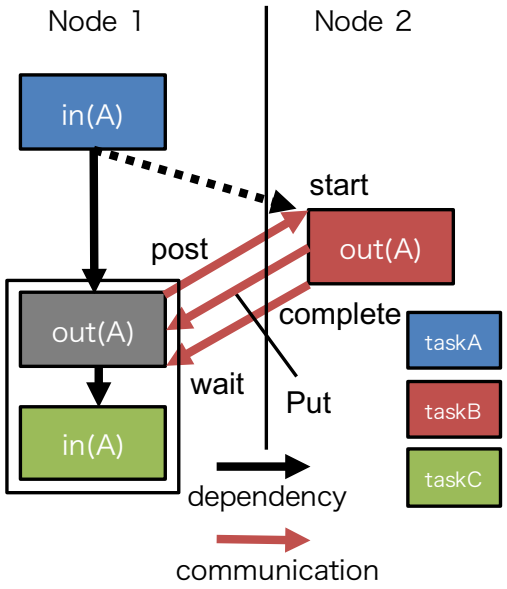


図 3.8 `tasklet` 指示文の `put`，`put_ready` 節のプログラミング例とタスクフロー．

```
#pragma xmp nodes P(2)
   int A:[*], B, tag;
#pragma xmp tasklets
{
#pragma xmp tasklet out(A) ¥
                 get_ready(A, P(2), tag) on P(1)
   A = 0;         /* taskA */
#pragma xmp tasklet in(A) out(B) get(tag) on P(2)
   B = A:[1];    /* taskB */
#pragma xmp tasklet out(A) on P(1)
   A = 1;        /* taskC */
}
```

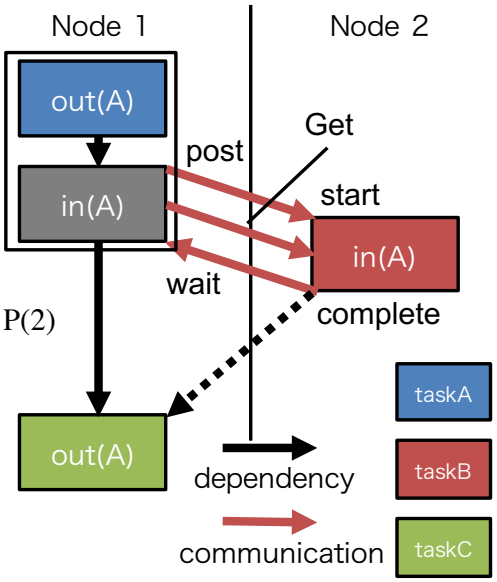


図 3.9 `tasklet` 指示文の `get`，`get_ready` 節のプログラミング例とタスクフロー．

taskC を実行する例となる．ノード 1 の *taskC* において `put_ready` 節で変数 *A* が指定されているため，依存関係として *out* のタスクが生成され，先行する *taskA* との反依存を生成することでタスクの実行順序を制御する．`put_ready` 節に記述されたノード集合とタグより，ノード集合に指定されたノード上のタグと一致する `put` 節が記述されたタスクに対して Put の実行可能通知を送り，Put の完了通知を受け取るまで待機する．完了通知の受信後は *taskC* のブロック内に記述された演算を実行する．図 3.8 の場合，*taskC* がノード 2 の *taskB* に対して実行可能通知を送り，そのタスクより完了通知を受け取る．ノード 2 の `put` 節が記述された *taskB* は，タスク実行開始後に Put の実行可能通知を受信するまで待機す

る．put 節に記述されたタグに一致するタスクより Put の実行可能通知を受信後にタスク内に記述された coarray/Put を実行する．Put の通信完了を保証した後は，実行可能通知を受信したノードに対して Put の完了通知を送りタスクの実行を終了する．

図 3.9 に tasklet 指示文の get，get_ready 節によるプログラミング例を示す．図 3.8 と同様に 2 ノードで実行され，ノード 2 の *taskB* からノード 1 の coarray 宣言された変数 *A* に対する coarray/Get を実行する．ノード 1 では，変数 *A* を更新する *taskA* の実行後に Get が実行され，再度変数 *A* の更新を行う *taskC* を実行する．ノード 1 の *taskA* において get_ready 節で変数 *A* が指定されているため，依存関係として *in* のタスクが生成され，後続の *taskC* との反依存を生成することでタスクの実行順序を制御する．*taskA* のブロック内の処理の実行後に，get_ready 節に記述されたノード集合とタグより，ノード集合に指定されたノード上のタグと一致する get 節が記述されたタスクに対して Get の実行可能通知を送り，Get の完了通知を受け取るまで待機する．完了通知の受信後はタスクの実行を終了する．図 3.9 の場合，*taskA* がノード 2 の *taskB* に対して実行可能通知を送り，そのタスクより完了通知を受け取る．ノード 2 の get 節が記述された *taskB* は，タスク実行開始後に Get の実行可能通知を受信するまで待機する．get 節に記述されたタグに一致するタスクより Get の実行可能通知を受け取り後にタスク内に記述された coarray/Get を実行する．Get の通信完了を保証した後は，実行可能通知を受信したノードに対して Get の完了通知を送りタスクの実行を終了する．

図 3.8，3.9 の例より put_ready，get_ready 節のどちらにおいても，ブロック内に記述された演算の実行と通信通知を依存関係の動的生成により 2 種類のタスクに分離することが可能な仕様とした．図 3.8 のタスクフロー中の点線のように，Put の開始通知を出すタスク（図中のグレーのタスク）を前もって実行することで，Put の実行開始を早めることが可能であり，他の演算タスクと通信のオーバラップが可能となる．同様に図 3.9 のタスクフロー中の点線では，Get の完了通知は他の反依存を持つタスクの実行の直前までに済ませておけば良いため，完了通知のための通信を遅らせることが可能である．今後の課題としては，タスクスケジューリングを改善し，put_ready，get_ready 節により生成される片側通信の開始通知を出すタスクのスケジューリングを行い，通信と計算のオーバラップ率の向上による性能向上を行うことが挙げられる．

3.3 MPI+OpenMP による実装

tasklet 指示文の実装を示す．tasklet 指示文を実装する XMP コンパイラを Omni XMP Compiler とする．Omni XMP Compiler は，XMP 指示文が記述されたコードを MPI+OpenMP で実装されたランタイム呼び出しへと変換する source-to-source なトランスレータである．従って，本研究では Omni XMP Compiler を改良することで，tasklet 指示文を対応する MPI の通信 API や OpenMP 指示文へと変換する実装を行う．しかし，Omni XMP Compiler によるランタイム呼び出しへの変換規則を示すだけでは，ランタイム内の挙動が見えない．そこで，本節では tasklet 指示文が Omni XMP Compiler によるコード変換でどのような挙動をする MPI+OpenMP コードへと変換されるかを擬似コードで示す．

提案する tasklet 指示文の変換後の MPI の通信 API や OpenMP 指示文を示す．tasklets 指示文は OpenMP parallel，single 指示文とし，taskletwait 指示文は OpenMP taskwait 指示文へと変換する．また，tasklet 指示文と in，out 及び inout 節は OpenMP task 指示文と depend 節

ソースコード 3.1 タスク並列における通信と同期.

```
1  int comp;
2  /* Communication */
3  MPI_Isend( ... );
4
5  /* Synchronization */
6  MPI_Test( &comp, ... );
7  while (! comp) {
8  #pragma omp taskyield
9    MPI_Test( &comp, ... );
10 }
```

の *in*，*out* 及び *inout* とする．ユーザが記述する片側通信の coarray や `gmove in/out` 指示文を除き，`tasklet` 指示文により生成される通信は，ソースコード 3.1 のように変換する．ソースコード 3.1 は，*MPI_Isend()* による送信側のみの例だが，受信側も同様に *MPI_Irecv()* と同期処理へと変換する．プロセスレベルの通信記述では，*MPI_Send/Recv()* によるブロッキング通信や，*MPI_Isend/Irecv()* によりノンブロッキング通信を開始し *MPI_Wait()* や *MPI_Waitall()* で同期をとる記述が一般的である．しかし，タスク内で通信を実行する場合，通信以外に演算を行うタスクが大量に実行されることを考慮に入れる必要がある．例えば，タスク内でブロッキング通信や *MPI_Wait()* などによる同期処理を行った場合，通信が完了するまでそのタスクはスレッドを専有する．つまり，他の依存関係が解消された実行可能なタスクが多数あるにも関わらず，通信完了までタスクの実行が滞り性能低下を引き起こすことが考えられる．そこで，ソースコード 3.1 のように *MPI_Test()* と OpenMP `taskyield` 指示文を用いる同期方法とする．*MPI_Test()* は通信完了を非同期的に確認可能な MPI の API である．`taskyield` 指示文は，他に実行可能なタスクがある場合に `taskyield` 指示文を実行したタスクを一時停止させ，他のタスクの実行を優先する指示文であり，ロックや通信などデッドロックが発生しうる箇所で用いられる．この 2 種類の構文を用いて，*MPI_Test()* により通信が完了していなければ `taskyield` 指示文により別の実行可能タスクへとスイッチすることで，デッドロックや通信タスクがスレッドを専有するのを防ぎ，スレッド内においても通信と計算のオーバラップを実現する．今後のグローバルビュー，ローカルビューにおける `tasklet` 指示文の変換後のコードにある P2P 通信は，全てソースコード 3.1 のようにコード変換されるが，コード表示の簡略化のため全て *MPI_Send/Recv()* と記述する．

3.3.1 グローバルビュー

グローバルビューが提供する `tasklet gmove`，`tasklet reflect` 指示文のコード変換例を示す．ソースコード 3.2 は，図 3.6 の `tasklet gmove` 指示文によるコードを MPI+OpenMP コードへと変換した例である．図 3.6 の *taskA* と *taskC* は，依存関係と `on` 節が記述された `tasklet` 指示文により生成されるため，4 から 7，18 から 21 行目のようにタスク内の記述を維持しつつ OpenMP `task` 指示文と

ソースコード 3.2 図 3.6 の tasklet gmove 指示文のコード変換例.

```
1  #pragma omp parallel
2  #pragma omp single
3  {
4      if (my node num is 1) {
5  #pragma omp task depend(out:A[0])
6          /* ... */
7      } /* taskA */
8      if (my node num is 1 or 2) {
9          if (I have the variable of rhs) {
10 #pragma omp task depend(in:A[0]) depend(out:A[1])
11             MPI_Send( ... ); /* Send A[0] to the node of lhs */
12         }
13         if (I have the variable of lhs) {
14 #pragma omp task depend(in:A[0]) depend(out:A[1])
15             MPI_Recv ( ... ); /* Receive A[0] from the node of rhs in A[1] */
16         }
17     } /* taskB */
18     if (my node num is 2) {
19 #pragma omp task depend(in:A[1]) depend(out:B)
20         /* ... */
21     } /* taskC */
22 }
```

depend 節，on 節を持つノードかどうかの if 文へと変換される．*taskB* を生成する tasklet gmove 指示文は，配列代入文に記述された分散配列やノードローカルな値を持つノードが通信を行う．配列代入文の左辺と右辺の要素と on 節により tasklet gmove 指示文を実行するノードは，通信の実行対象の全てのノードが判断可能なため MPI の P2P 通信へと変換できる．コード変換では，8 行目からの if 文で on 節で指定された実行ノードのみとし，9 から 12 行目の if 文で右辺，13 から 16 行目の if 文で左辺の実行ノードを決定し，それぞれタスクで通信が実行される．

ソースコード 3.3 は，図 3.7 の tasklet reflect 指示文によるコードを MPI+OpenMP コードへと変換した例である．図 3.7 では，loop 指示文で並列実行されるループ内で tasklet 指示文による依存関係を持つタスクである *taskA* を生成する．コード変換により *taskA* は，5 行目の分散されたループ内でタスク内の記述を維持しつつ OpenMP task 指示文と depend 節，on 節を持つノードかどうかの if 文へと変換される．*taskB* を生成する tasklet reflect 指示文の場合は袖領域通信をタスクで実行するため，送信する領域と隣接ノードから受信する領域が依存関係となる．図 3.7 の場合，1 次元の分散配列

ソースコード 3.3 図 3.7 の `tasklet reflect` 指示文のコード変換例.

```
1  #pragma omp parallel
2  #pragma omp single
3  {
4      /* loop length is distributed by loop directive */
5      for (int i = local_begin; i < local_end; i++) {
6          if (I have template T[i]) {
7  #pragma omp task depend(in:A[i])
8              /* ... */
9          } /* taskA */
10     }
11     /* It is assumed that this example of code translation is executed by two nodes */
12     if (my node num is 1) {
13 #pragma omp task depend(in:A[upper])
14         MPI_Send( ... ); /* Send A[upper] for updating lower halo region of node 2 */
15 #pragma omp task depend(out:A[upper+1])
16         MPI_Recv( ... ); /* Receive A[upper+1] from node 2 */
17     }
18     if (my node num is 2) {
19 #pragma omp task depend(in:A[lower])
20         MPI_Send( ... ); /* Send A[lower] for updating upper halo region of node 1 */
21 #pragma omp task depend(out:A[lower−1])
22         MPI_Recv( ... ); /* Receive A[lower−1] from node 1 */
23     } /* task B */
24 }
```

の上端下端に 1 要素ずつ袖領域を持つため，12 と 18 行目からの if 文によりノード 1，2 における境界領域の送信と袖領域の受信をそれぞれタスクで実行する．境界領域や袖領域の依存関係はグローバルビューのデータ分散に基づき，ランタイムが境界インデックスを計算し自動で `depend` 節を付与する．

3.3.2 ローカルビュー

ローカルビューが提供する `tasklet` 指示文の `put`，`put_ready`，`get` 及び `get_ready` 節のコード変換例を示す．本研究において，タスク内での片側通信を実現するための開始通知と完了通知は，メッセージ長 0 の P2P 通信による実装とする．従って，1 回の片側通信を実行するためには，実行主体は *MPI_Recv()* で開始通知を受け取った後，片側通信を実行，片側通信の完了保証後に *MPI_Send()* で完了通知を送る 3 種類の通信が必要となる．また，Omni XMP Compiler の coarray や `gmove in/out` 指示文

ソースコード 3.4 図 3.8 の put，put_ready 節のコード変換例.

```
1  #pragma omp parallel
2  #pragma omp single
3  {
4      if (my node num is 1) {
5  #pragma omp task depend(in:A) depend(out:B)
6          /* ... */
7      } /* taskA */
8      if (my node num is 2) {
9  #pragma omp task depend(out:A)
10 {
11         MPI_Recv( ... ); /* Receive zero element msg from any source with tag */
12         MPI_Put( ... ); /* Put the data to the node */
13         MPI_Win_flush( ... ); /* Completion of one−sided communication */
14         MPI_Send( ... ); /* Send zero element msg to the node with tag */
15 }
16     } /* taskB */
17     if (my node num is 1) {
18 #pragma omp task depend(out:A)
19 {
20         MPI_Send( ... ); /* Send zero element msg to P(2) with tag */
21         MPI_Recv( ... ); /* Receive zero element msg from P(2) with tag */
22 }
23 #pragma omp task depend(in:A) depend(out:C)
24         /* ... */
25     } /* taskC */
26 }
```

のランタイム実装は，MPI の片側通信のパッシブモデルによる同期を用いており，プログラムの開始と終了時に *MPI_Win_lock_all()*，*MPI_Win_unlock_all()* を呼び，全てのノードに対する片側通信の実行を可能とし，同期は *MPI_Win_flush()* など通信主体側のみ完了保証をする実装である．そのため，本研究においても同様のモデルによる実装とする．

ソースコード 3.4 は，図 3.8 の put，put_ready 節によるコードを MPI+OpenMP コードへと変換した例である．図 3.8 の *taskA* は，依存関係と on 節が記述された tasklet 指示文により生成されるため，4 から 7 行目のようにタスク内の記述を維持しつつ OpenMP task 指示文と depend 節，on 節を持つノードかどうかを判断する if 文へと変換される．*taskB* は put 節が記述されているため，*tag* と

ソースコード 3.5 図 3.9 の `get`，`get_ready` 節のコード変換例.

```
1  #pragma omp parallel
2  #pragma omp single
3  {
4      if (my node num is 1) {
5  #pragma omp task depend(out:A)
6          /* ... */
7  #pragma omp task depend(in:A)
8  {
9          MPI_Send( ... ); /* Send zero element msg to P(2) with tag */
10         MPI_Recv( ... ); /* Receive zero element msg from P(2) with tag */
11 }
12     } /* taskA */
13     if (my node num is 2) {
14 #pragma omp task depend(in:A) depend(out:B)
15 {
16         MPI_Recv( ... ); /* Receive zero element msg from any source with tag */
17         MPI_Get( ... ); /* Get the data from the node */
18         MPI_Win_flush( ... ); /* Completion of one−sided communication */
19         MPI_Send( ... ); /* Send zero element msg to the node with tag */
20 }
21     } /* taskB */
22     if (my node num is 1) {
23 #pragma omp task depend(out:A)
24         /* ... */
25     } /* taskC */
26 }
```

一致する `put_ready` 節を持つタスクより開始通知を待ち，通知の受信後は coarray/Put を実行し通信完了を保証する．その後，開始通知を受信したノードに対して完了通知を送る．11 行目が開始通知を受信するための *MPI_Recv()*，12 行目が `tasklet` 指示文内のブロックに記述された coarray/Put を変換した *MPI_Put()*，13 行目が片側通信の完了保証をする *MPI_Win_flush()*，14 行目が完了通知を送信する *MPI_Send()* をそれぞれ表す．*taskC* は `put_ready` 節が記述されているため，このタスクが実行可能となった時点で `put_ready` 節に記述された対象ノードの *tag* と一致するタスクに対して，片側通信の開始通知を送る．開始通知を送信後は完了通知を受信する *MPI_Recv()* の同期待ちをし，完了通知を受信後に `tasklet` 指示文に記述された処理の実行を開始する．18 行目からの `task` 指示文が片側通信の開始，

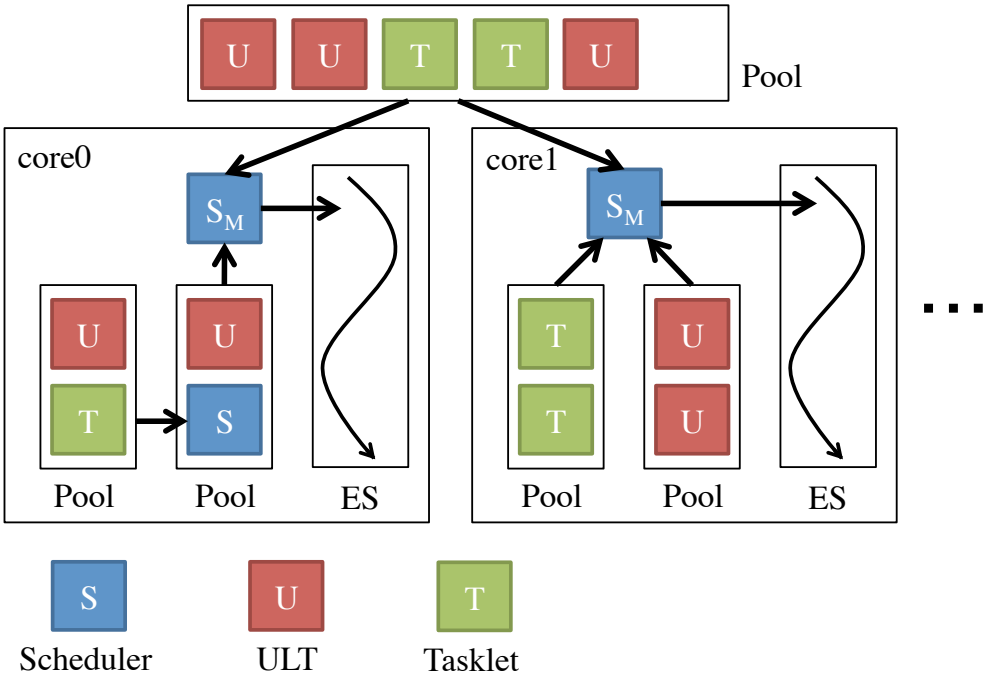


図 3.10　Argobots の実行モデル．

完了通知の送受信をするタスクであり，20 行目が開始通知を送信する *MPI_Send()*，21 行目が完了通知を受信する *MPI_Recv()* をそれぞれ表す．このタスクは `put_ready` 節に記述された要素を基に自動生成される．23 行目以降は，*taskA* と同様に `tasklet` 指示文より変換される．

ソースコード 3.5 は，図 3.9 の `get`，`get_ready` 節によるコードを MPI+OpenMP コードへと変換した例である．図 3.9 の *taskA* は，`get_ready` 節が記述されているため，`tasklet` 指示文内に記述された処理を実行後に，片側通信の Get が `get_ready` 節に記述されたノードより実行される．対象ノードの *tag* と一致するタスクに片側通信の開始通知を送信し，完了通知を受信する *MPI_Recv()* の同期待ちをする．9 行目の *MPI_Send()* が開始通知の送信，10 行目の *MPI_Recv()* が完了通知の受信を表す．5 行目の `task` 指示文は `tasklet` 指示文と依存関係により変換され，7 行目のタスクは `get_ready` 節により自動生成される．*taskB* は `get` 節が記述されているため，`get_ready` 節が記述されている *tag* と一致するタスクより片側通信の開始通知を受信し，coarray/Get の実行後に通信完了を保証する．その後，開始通知を受信したノードに対して完了通知を送る．`get` 節の変換は，`put` 節とほぼ同等の変換が行われ，16 行目が開始通知を受信する *MPI_Recv()*，17 行目が `tasklet` 指示文内のブロックに記述された coarray/Get を変換した *MPI_Get()*，18 行目が片側通信の完了保証をする *MPI_Win_flush()*，19 行目が完了通知を送信する *MPI_Send()* をそれぞれ表す．*taskC* は，依存関係と `on` を持つ `tasklet` 指示文であるため，22 から 25 行目のように `task` 指示文と `depend` 節及び `on` 節の条件を満たすノードかどうかを判断する if 文へと変換される．

3.4 Argobots による実装

タスク生成の高速化や，一定動作をするタスクスイッチング機構を実現するための軽量スレッドライブラリ Argobots によるタスク並列実行の実装を示す．本節では，Argobots の概要を示した後，Argobots によるタスク並列実行の実装を示す．Argobots は Argonne National Laboratory（ANL）のエクサスケールコンピューティングに向けた研究プロジェクト Argo[53] の一部として研究開発が進められている，ユーザレベルのスレッドライブラリである．大量の細粒度タスクによる並列処理を実現するためのフレームワークとして，スレッド制御機構やタスクのスケジューリングのための API を提供している．図 3.10 に Argobots の実行モデルを示す．Argobots の実行単位である *Working Unit*（*WU*）は，*User Level Thread*（*ULT*）と *Tasklet* の 2 種類がある．*ULT* は既存のスレッドライブラリと同等の機能をユーザレベルで提供しており，*mutex* や条件変数などの排他制御や同期機構に加え，高速なタスク生成・コンテキストスイッチを特徴としている．*Tasklet* は関数を抽象化した軽量実行ユニットであり，高速に実行可能だが *ULT* が持つ排他制御や同期機構を持たず，コンテキストスイッチもできないなど制約も強い．

各 *WU* は直接コアに割り当てられて実行されるのではなく，*WU* を入れるためのキューであるプールに入れられ，プール内の *WU* を *Execution Stream*（*ES*）が実行する．図 3.10 では，CPU の各物理コアに対して一つの *ES* が割り当てられている．*ES* に紐付けられたマススケジューラによりプールから *WU* が取り出され *ES* 上で逐次的に実行されるが，それぞれの *ES* は並列に動作するため並列実行が可能である．プールには *WU* の他にスケジューラを入れることも可能である．図 3.10 のコア 0 のマススケジューラに割り当てられたプール内にあるスケジューラが実行可能になった場合は，そのスケジューラが持つプール内の *WU* が実行される．また，コア 1 のプールのようにマススケジューラに複数のプールを割り当てることや，コア 0，1 で共有のプールを設定することも可能であり，*ES* 間でのタスクのスティーリングやマイグレーションもユーザが実装可能である．

Argobots の各機能とタスク，スレッドとの対応付けとタスク並列実行の実装の説明を示す．スレッドと Argobots の *ES* は一対一の対応とし，ユーザが指定する実行スレッド数によって *ES* の数を指定可能とする．各物理コアに対して *ES* とプールを一つずつ割り当てるが，全ての *ES* が全てのプールへとアクセス可能とする．基本的には，各 *ES* が持つマススケジューラによって自分に割り当てられたプールからタスクを取り出し *ES* 上で実行するが，プールにタスクがない場合は他の *ES* が持つプールよりタスクのスティーリングを実行する実装とする．タスクの生成はマススレッドによって逐次的行われ，`tasklet` 指示文によって生成されるタスクを一つの *ULT* として扱う．`in`，`out` 及び `inout` 節を持たない場合や既に依存関係が満たされている場合は，タスクは直ちに生成されプールへと入れられる．マススレッドは，`taskletwait` 指示文か `tasklets` 指示文によるブロックの出口まで到達したときなど，ユーザによって指定された同期ポイントまでの全てのタスクの生成を終えたのちに，タスクの実行を開始する．

依存関係を持つタスクの実行条件を示す．`in` 節が指定されたタスクの場合，依存関係はフロー依存のみであるため，直前の `out` 節を持つタスクの終了により実行が開始される．`out` 節を持つタスクの場合，出力依存と反依存の 2 種類の依存関係があるため，直前の `out` 節を持つタスクとそのタスク以降に生成された `in` 節を持つ全てのタスクの終了により実行が開始される．以上の実行条件を表すため，本研究で

表 3.1　実験環境（Oakforest-PACS）.

CPU	Intel Xeon Phi 7250 1.4 GHz 68 cores
Memory	16GB (MCDRAM) + 96GB (DDR4)
Interconnect	Intel Omni-Path Architecture
Compiler	Intel Compiler 17.0.1
	Intel MPI Library 2017 Update 1

はカウンタによる実装を行った．`in`，`out` 及び `inout` 節に記述された変数や配列セクションは，それらのアドレスを基にハッシュ値が生成されハッシュテーブルのバケット上で管理される．バケットはリストによる実装とした．タスク生成時に自タスクと依存関係を持つタスクが既にあり，そのタスクの実行が開始されていない場合，カウンタの値を増加させハッシュテーブルのバケットに追加する．タスクの実行終了時に，自タスクが持つ依存と同一の依存をハッシュ値よりハッシュテーブルから探索し，格納されているタスクのカウンタの値を減らす．この時，カウンタの値が 0 となった場合，タスクの依存関係が解消したとしバケットから取り出した後，*ULT* を生成しプールへ挿入する．

OpenMP の `taskyield` 指示文で実行されるタスクのスイッチングは，Argobots が提供するタスクスイッチング関数を用いる．*ULT* 内でこの関数が呼ばれた場合，現在実行中の *ULT* はプールへと退避させられ，マスタースケジューラが起動される．プールへ挿入されているタスクは依存がない，もしくは既に解消されているタスクのみであるため，マスタースケジューラはプールからタスクを取り出し，直ちに実行を開始する．本研究における実装では，デッドロックが起きうる通信同期において，通信が完了していなければタスクスイッチングは必ず実行される実装とした．

3.5　通信性能の評価

1.3.3 節において，プロセスレベルでの通信と比較して MPI_THREAD_MULTIPLE のマルチスレッド環境における通信性能は低いという既知の問題について示した．本節では，Intel Omni-Path や InfiniBand をインターコネクトとして持つ環境にて，実際の MPI_THREAD_MULTIPLE の性能を示した後，通信性能改善のための実装について述べる．

3.5.1　実験環境

Intel Omni-Path をインターコネクトとして持つ環境として筑波大学計算科学研究センターと東京大学情報基盤センター共同の Joint Center for Advanced HPC : 最先端共同 HPC 基盤施設（JCAHPC）が運用する Oakforest-PACS（OFP）[54] を利用する．また，InfiniBand をインターコネクトとして持つ環境としては，筑波大学計算科学研究センターが運用する COMA[55] を利用する．各システムの 1 ノードの計算機環境やソフトウェアのバージョンをそれぞれ表 3.1，3.2 に示す．OFP は Intel Xeon Phi 7250（68 コア）をホストプロセッサとし，インターコネクトは理論ピークバンド幅 100Gb/s を持つ Intel Omni-Path Architecture である．COMA は，1 ノードに Intel Xeon E5-2670v2 の 2 ソケット（10 コア × 2 ソケット）

表 3.2　実験環境（COMA）.

CPU	Intel Xeon E5-2670v2 × 2
	10 cores/CPU × 2 = 20 cores
Memory	64GB (DDR3)
Interconnect	Mellanox Connect-X3 Single-port FDR
Compiler	Intel Compiler 16.0.4
	Intel MPI Library 5.1.3

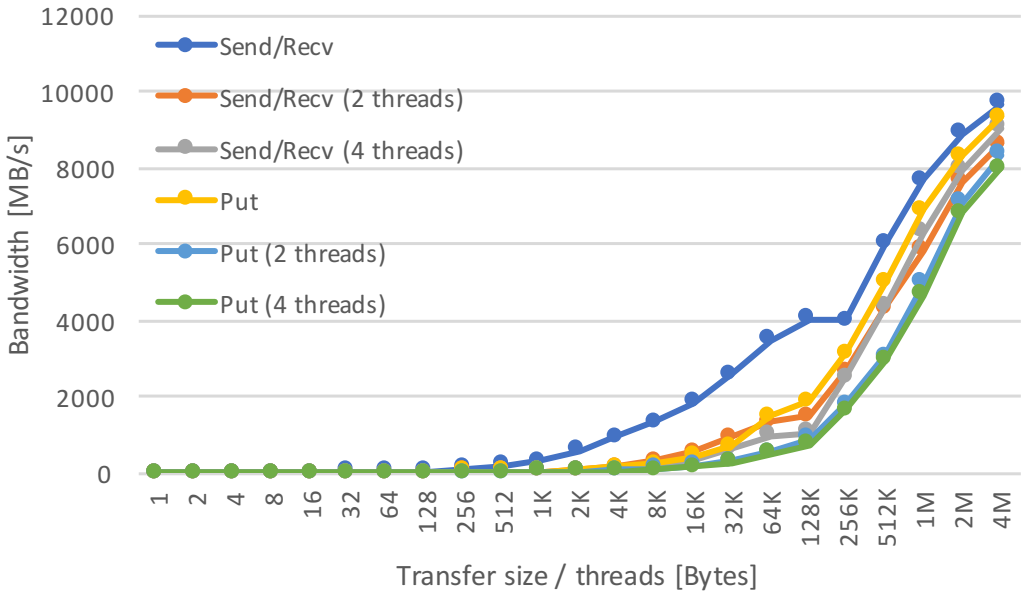


図 3.11　通信性能の評価（Oakforest-PACS）.

と Intel Xeon Phi 7110P コプロセッサを 2 枚（61 コア × 2）持つが，本研究ではホストプロセッサのみを対象とする．インターコネクトとして Mellanox Connect-X3 Single-port FDR を採用しており，理論ピークバンド幅は 56Gb/s である．MPI やスレッド通信を実装するための OpenMP のコンパイラとして Intel Compiler を使用する．

3.5.2　マルチスレッド環境での通信性能の予備評価

3.5.1 節で示した環境を用い，予備評価として通信性能の調査を行う．タスク並列ではスレッド上で実行されるタスクから通信が実行されるため，複数のスレッドが同時に通信を行うことが可能な MPI_THREAD_MULTIPLE が必要となる．そこで，通信性能の評価には，通常のプロセス対プロセスの Ping-Pong プログラムに加えて，スレッド対スレッドで通信を実行可能な Ping-Pong プログラムを用いる．また，グローバルビューでは P2P 通信，ローカルビューは片側通信であるため，*MPI_Send/Recv()*,

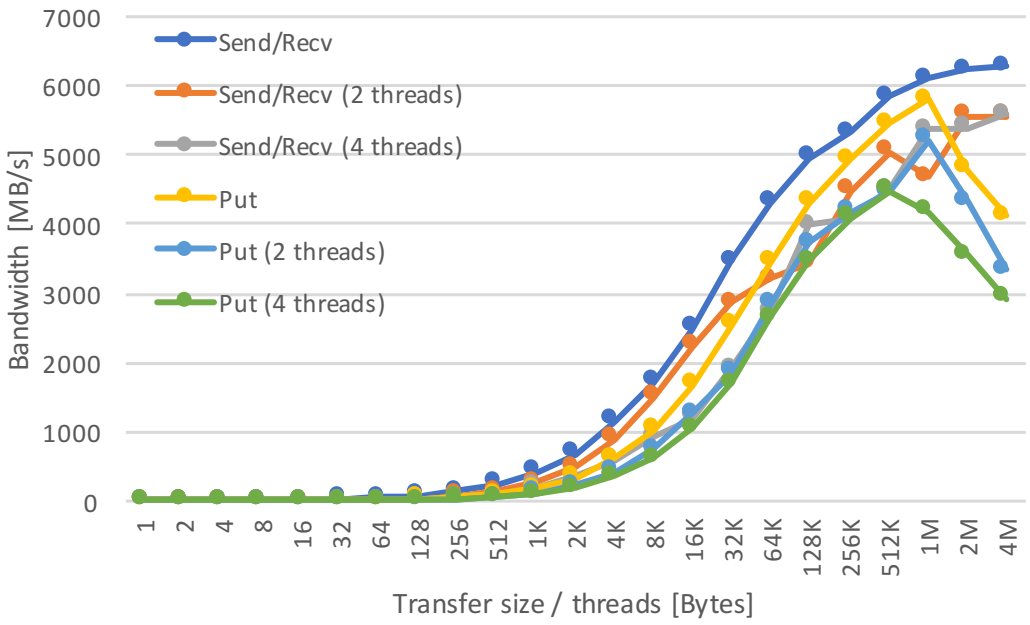


図 3.12　通信性能の評価（COMA）.

MPI_Put() のそれぞれを用いて性能評価を行う．片側通信の場合は通信対象に依らず通信を実行可能であるため，*MPI_Put()* を実行後に *MPI_Win_flush()* で完了を保証し，*MPI_Send/Recv()* による 0Byte メッセージ通信を用いて完了通知を送る実装とした．さらに，同時に通信するスレッド数を 1，2，4 と増加させた場合の評価も行う．Ping-Pong プログラムにはオハイオ州立大学で研究開発が進められている OSU Micro-Benchmarks を用い，スレッドレベルでの通信の場合は，そのプログラムをベースとした Ping-Pong プログラムを実装し，評価に用いた．

図 3.11，3.12 に OFP と COMA での通信性能の評価を示す．横軸は 1 スレッドが通信するサイズを表し，縦軸はスレッドあたりのバンド幅を表す．例えば軸の 4KB は，1 スレッドならば 4KB，2 スレッドは 8KB，4 スレッドは 16KB を合計で送信する．まず，1 スレッド実行での *MPI_Send/Recv()* と *MPI_Put()* を比較する．どちらのシステムにおいても *MPI_Put()* より *MPI_Send/Recv()* の性能が良いことがわかる．OFP におけるその差は大きく，最大で 2GB/s 以上性能が異なる．また，COMA における *MPI_Put()* は 1MB 以上の通信において性能が急激に低下することがわかった．次に 1 スレッドとマルチスレッドでの通信を比較する．どちらのシステムにおいても *MPI_Send/Recv()*，*MPI_Put()* ともにスレッド数増加とともに性能が低下する．従って，InfiniBand，Intel Omni-Path のどちらのインターコネクトにおいても片側通信より P2P 通信の性能が良く，マルチスレッドで通信を行う MPI_THREAD_MULTIPLE よりも 1 スレッドが通信を行う従来のプロセス単位での通信が高速であることが示された．

3.5.3　通信委譲による通信最適化

タスク並列実行におけるマルチスレッドでの通信性能を改善するために，通信最適化を行う．3.5.2 節より，Intel Omni-Path，InfiniBand ともに，1 プロセスもしくは 1 スレッドが通信した場合の性能は良い

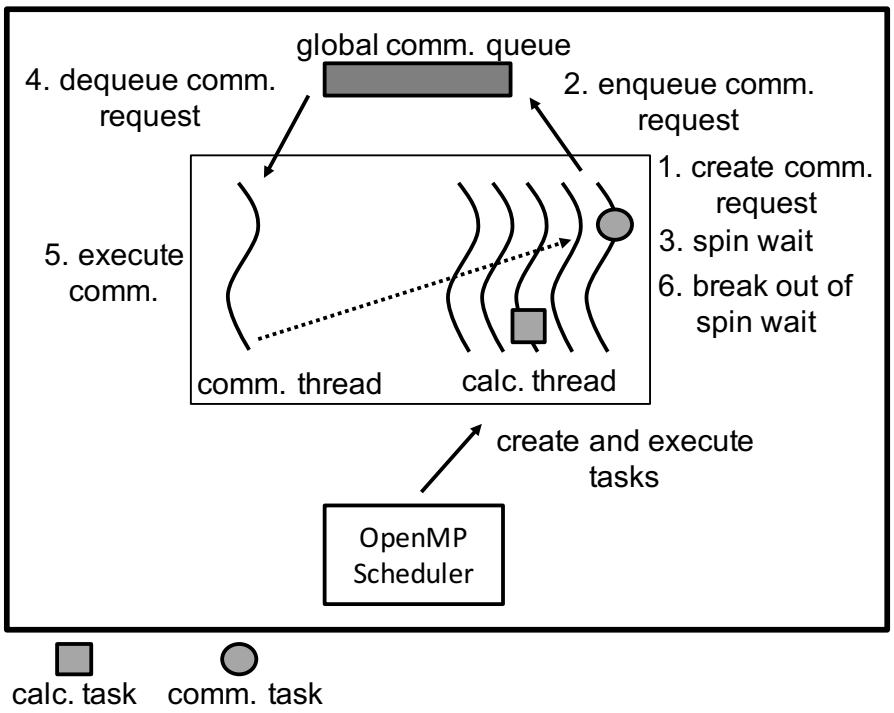


図 3.13 通信最適化の実装.

ため，本研究では tasklet 指示文によるタスク並列実装において，ユーザからはマルチスレッドに通信を実行するようなプログラムで，ランタイムにより 1 スレッドに通信を委譲する実装とする．汎用的な実装とするため通信最適化の実装は全て OpenMP で行う．また，比較として Argobots による実装も行い性能を示す.

図 3.13 に 1 スレッドに通信委譲をする通信最適化の実装の概要を示す．実行スレッドの 1 つを通信スレッドとし，その他のスレッドを計算スレッドとする．tasklets 指示文の変換により OpenMP parallel，single 指示文へと変換されるが，その 2 つの指示文の間に master 指示文を挿入し，その指示文で生成されるスレッドを通信スレッドとする．従って，single 指示文内で実行される task 指示文が master 指示文のスレッドで実行されることはない．通信を実行するタスクを含む全てのタスクは計算スレッドで一度実行される．通信タスクの場合は通信をスレッドで直接実行するのではなく，通信リクエストを生成し全てのスレッドが参照可能なキューへと入れる．キューへ挿入後は通信完了まで taskyield 指示文を呼びながら，通信リクエストに対してスピンウェイトを実行する．通信リクエストの値の書き換えを通信完了とし，スピンウェイトを抜けタスクを終了する．通信スレッドは，キューにリクエストが挿入されるのを待つ．キューにリクエストがある場合は，そのリクエストを基に通信を生成し実行する．通信の完了は *MPI_Test()* により非同期的に確認し，通信が完了していれば通信リクエストの値を書き換える．single 指示文のブロックの出口に到達し，全てのタスクが終了したのちに通信スレッドは終了する．Argobots による実装の場合，ES0 を通信タスクを実行するスレッドとする．ES0 上には常に 1 つのタスクが実行されており，このタスクのみが通信を実行する．他の ES 上で実行されるタスクの挙動は OpenMP 実装と同様にキューによる通信リクエストのやり取りであるため詳細は省略する.

3.6　評価

提案する XMP のタスク並列モデルの性能評価を示す．`tasklet` 指示文のグローバルビュー，ローカルビューをそれぞれ用いたベンチマークを作成し，Intel Xeon Phi メニーコアプロセッサを搭載するクラスタ上で性能評価を行う．また，Intel Xeon プロセッサを搭載するクラスタ上での実行も行い，マルチコア環境での性能評価も合わせて示す．対象のベンチマークはブロックコレスキー分解とラプラスソルバとする．XMP 実装との比較対象は以下に示した実装であり，全て MPI+OpenMP による実装である．

- ループ並列によるワークシェアリングとデータ依存モデルによるタスク並列による実装.
- *MPI_Send/Recv()* による P2P 通信と *MPI_Put()* による片側通信による実装.
- OpenMP と Argobots の比較.
- 通信最適化の有無.

以上の実装との比較により，従来のループ並列によるワークシェアリングとタスク並列による実装を比較することで，メニーコアシステムにおけるタスク並列に優位性を示す．また，XMP と MPI+OpenMP のタスク依存による実装の性能比較により，タスク並列による XMP プログラムの性能が MPI+OpenMP と同等であり，生産性が高い実装であることを示す．最後に Argobots による並列タスク実行の実装や通信最適化の適用によりタスク並列による実装のさらなる高速化を目指す．Omni XMP Compiler への `tasklet` 指示文の実装は，ランタイムのみ実装が完了しているためコード変換は手動で行った．ただし，`tasklet gmove` 指示文のみ一部コード変換の実装が完了しており，ブロックコレスキー分解のみ XMP グローバルビューの性能を評価する．また，Argobots の適用はブロックコレスキー分解のみであり，ラプラスソルバへの適用は今後の課題としたい．

3.6.1　環境変数，実行時パラメータ

評価環境には，3.5.1 節で用いた OFP と COMA の 2 種類のシステムを用いる．Intel Xeon Phi クラスタとして OFP，Intel Xeon クラスタとして COMA を利用する．XMP のコンパイラには Omni Compiler 1.0.3 をベースに `tasklet` 指示文を実装したものを用い，バックエンドコンパイラや比較対象の MPI+OpenMP 実装には Intel Compiler を使用する．ブロックコレスキー分解で用いられる BLAS や LAPACK の実装には Intel MKL を使用し，OFP では 2017 Update1，COMA では 11.3.2 を用いた．

実行時に用いる実行パラメータや環境変数を示す．KNL のメモリモードを Flat モード，クラスタリングモードは OFP が指定する Quadrant モードとした．Flat モードで MCDRAM を用いる方法として numactl コマンドによる方法がある．numactl コマンドにより DDR4 メモリが NUMA ノード 0，MCDRAM が NUMA ノード 1 として見える．そこで，MCDRAM 上に全てのデータを配置する場合は，numactl -membind 1 と指定する．しかし，本研究で用いるベンチマークでは演算用のバッファ全体は MCDRAM 上に収まるが，初期化や結果検証のためのバッファを含めると MCDRAM の 16GB を超える．そこで，memkind ライブラリを用いて MEMKIND_HBW_NODE=1，numactl -membind 0 とすることで DDR4 メモリのみを使用すると指定し，演算バッファを *hbw_posix_memalign()* で確保する．それにより，

演算バッファのみを MCDRAM に，その他を DDR4 メモリ上に確保することが可能である．

OFP では Intel Omni-Path の通信スタックとして Tag Matching Interface（TMI），OpenFabrics Alliance（OFA）及び OpenFabrics Interface（OFI）が使用可能であり，COMA では Direct Access Programming Library（DAPL）と OFA が選択できる．I_MPI_FABRICS により設定が可能であり，本研究では OFP では TMI，COMA では OFA とした．

文献 [56] において，OFP ではタイマ割り込みを受けるコアが 0 と設定してあると述べられており，コア 0 の使用が推奨されていない．また，KNL は同一タイルにおいて 2 コアが L2 キャッシュを共有しているため，L2 キャッシュ汚染を考えると 2 コアを演算に用いない設定が良いとされている．そこで，KMP_HW_SUBSET=64c@2,1t とすることで，“64c,1t” で 1 コア 1 スレッドの実行を表し，“@2” で最初の 2 コアを飛ばす設定となる．また，I_MPI_PIN_PROCESSOR_EXCLUDE_LIST=“0, 1, 68, 69, 136, 137, 204, 205” とすることで，指定したスレッド上に MPI プロセスがバインドされない設定となる．KNL は，512bit の simd 長を用いることが可能な AVX512 をサポートしているため，コンパイル時のフラグに-axMIC-AVX512 を追加した．

3.6.2 ブロックコレスキー分解の性能評価

コレスキー分解は，正定値対称行列を下三角行列とその転置の積に分解するコードであり，本研究ではブロック化した行列を解くブロックコレスキー分解を対象とする．ブロックコレスキー分解のコードは OmpSs のパッケージ内で提供されているコードを用いる．OmpSs のコードは共有メモリ向けの実装であるため，その実装を基に OpenMP 実装を行い，MPI と組み合わせることで分散メモリ環境に対応する実装を行った．ブロックコレスキー分解は以下の 4 種類の処理に分けられる．括弧内は BLAS や LAPACK のルーチン名を表す．

1. コレスキー分解（potrf）
2. 三角行列を係数行列とする行列方程式を解く（trsm）
3. 対称行列のランクを更新（syrk）
4. 行列積（gemm）

上記の処理は全てブロック単位で実行される．各演算は OpenMP の `task` 指示文と `depend` 節で記述されたタスク内で実行され，データ依存で実行順序が制御される．図 3.14 にブロック数が 4×4 の行列を対象としたブロックコレスキー分解を，1 ノードで実行したときのタスクフローを示す．ブロックコレスキー分解のそれぞれの処理には依存関係が存在し，例えば初回のイテレーションの場合，*potrf(A[0][0])* は，*trsm(A[0][0], A[0][1])*，*trsm(A[0][0], A[0][2])* 及び *trsm(A[0][0], A[0][3])* と *A[0][0]* に関するフロー依存がある．しかし，*trsm* 間では依存関係がないため並列実行が可能であり，待機中のスレッドがあれば順次タスクがスケジューリングされ実行を開始する．

`tasklet` 指示文による実装

`tasklet` 指示文によるブロックコレスキー分解の実装例をソースコード 3.6，3.7 に示す．ソースコード 3.6 が `tasklet gmove` 指示文，ソースコード 3.7 が `tasklet` 指示文の `get`，`get_ready` 節によ

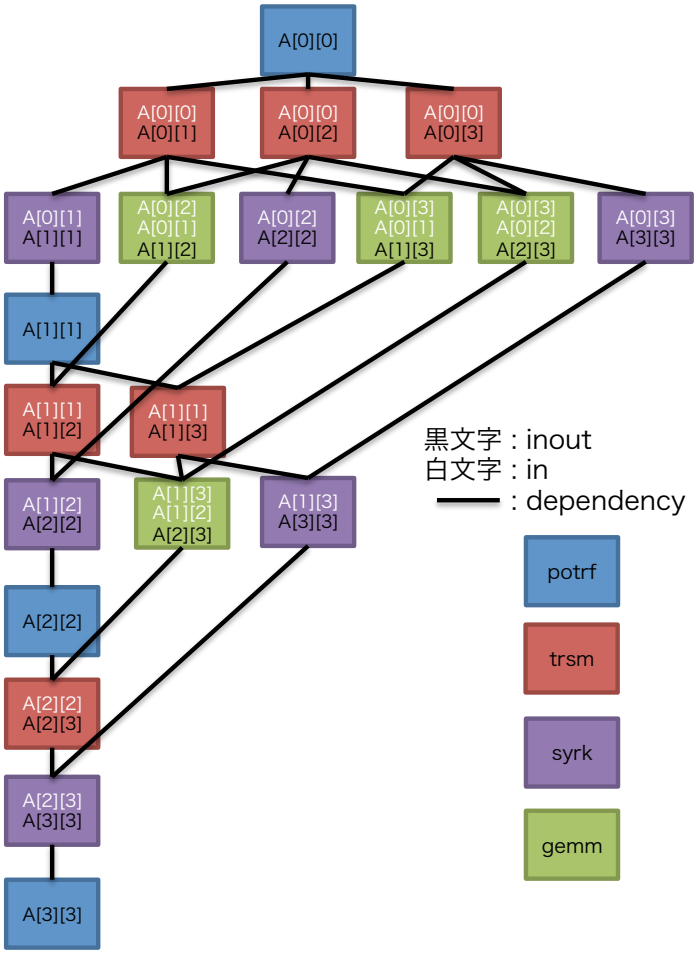


図 3.14　4×4 ブロックを持つブロックコレスキー分解のタスクフロー．

る実装である．どちらのコードも配列 *A* を分散配列とし，配列 *B* と *C* を通信バッファとして扱う．配列 *A* はブロックコレスキー分解を実行する配列であり，1，2 次元目がブロックインデックスで 3 次元目がブロックの要素を表す．全ての演算はブロック単位で行われるため，データ依存記述は 1，2 次元のブロックインデックスで全て記述される．どちらのコードにおいても 3 から 5 行目の `template`，`distribute` 及び `align` 指示文により，配列 *A* の 2 次元目に対してサイクリック分割が指定されている．

`tasklet` 指示文や `on` 節は，1 回の BLAS や LAPACK のルーチン呼び出しに対して記述される．BLAS や LAPACK のルーチンの演算内容によりデータ依存を決定し，実行ノードはそれらの演算による更新が発生する領域を持つノードのみがタスクを生成，実行する実装とした．例えば，*syrk* の場合，左辺を用いて右辺のブロックが更新されるため，`in` 節に *A[k][i]*，`out` 節に *A[i][i]* を記述し，*A[i][i]* を持つノードを表すテンプレート `T(i)` を `on` 節に記述する．

potrf から *trsm*，*trsm* から *gemm* と *syrk* においてフロー依存が存在するため，データ分散によって各演算の間で通信が必要となる．ここでは，*potrf* と *trsm* 間の通信実装のみを説明する．ソースコード 3.6 では，12 行目の `tasklet gmove` 指示文により *potrf* で演算された *A[k][k]* を各ノードが持つローカル配列 *B* へと通信する．従って，依存記述は `in` 節に *A[k][k]*，`out` 節に *B* となる．*trsm* を演算するノードは

ソースコード 3.6 `tasklet gmove` 指示文によるブロックコレスキー分解の実装例.

```
1  double A[nt][nt][ts*ts], B[ts*ts], C[nt][ts*ts];
2  #pragma xmp nodes P(*)
3  #pragma xmp template T(0:nt-1)
4  #pragma xmp distribute T(cyclic) onto P
5  #pragma xmp align A[*][i][*] with T(i)
6
7  #pragma xmp tasklets
8  for (int k = 0; k < nt; k++) {
9  #pragma xmp tasklet out(A[k][k]) on T(k)
10     potrf(A[k][k]);
11
12 #pragma xmp tasklet gmove in(A[k][k]) out(B) on T(k:)
13     B[:] = A[k][k][:];
14
15     for (int i = k + 1; i < nt; i++) {
16 #pragma xmp tasklet in(B) out(A[k][i]) on T(i)
17         trsm(B, A[k][i]);
18
19 #pragma xmp tasklet gmove in(A[k][i]) out(C[i]) on T(i:)
20         C[i][:] = A[k][i][:];
21     }
22     for (int i = k + 1; i < nt; i++) {
23         for (int j = k + 1; j < i; j++) {
24 #pragma xmp tasklet in(A[k][i], C[j]) out(A[j][i]) on T(j)
25             gemm(A[k][i], C[j], A[j][i]);
26         }
27 #pragma xmp tasklet in(A[k][i]) out(A[i][i]) on T(i)
28         syrk(A[k][i], A[i][i]);
29     }
30 }
```

ソースコード 3.7　get，get_ready 節によるブロックコレスキー分解の実装例.

```
1  double A[nt][nt][ts*ts], B[ts*ts], C[nt][ts*ts];
2  #pragma xmp nodes P(*)
3  #pragma xmp template T(0:nt-1)
4  #pragma xmp distribute T(cyclic) onto P
5  #pragma xmp align A[*][i][*] with T(i)
6
7  #pragma xmp tasklets
8  for (int k = 0; k < nt; k++) {
9  #pragma xmp tasklet out(A[k][k]) get_ready(A[k][k], T(k:), k*nt+k) on T(k)
10     potrf(A[k][k]);
11
12 #pragma xmp tasklet in(A[k][k]) out(B) get(k*nt+k) on T(k:)
13 #pragma xmp gmove in
14     B[:] = A[k][k][:];
15
16     for (int i = k + 1; i < nt; i++) {
17 #pragma xmp tasklet in(B) out(A[k][i]) get_ready(A[k][i], T(i:), k*nt+i) on T(i)
18         trsm(B, A[k][i]);
19     }
20
21     for (int i = k + 1; i < nt; i++) {
22 #pragma xmp tasklet in(A[k][i]) out(C[i]) get(k*nt+i) on T(i:)
23 #pragma xmp gmove in
24         C[i][:] = A[k][i][:];
25
26         for (int j = k + 1; j < i; j++) {
27 #pragma xmp tasklet in(A[k][i], C[j]) out(A[j][i]) on T(j)
28             gemm(A[k][i], C[j], A[j][i]);
29         }
30 #pragma xmp tasklet in(A[k][i]) out(A[i][i]) on T(i)
31         syrk(A[k][i], A[i][i]);
32     }
33 }
```

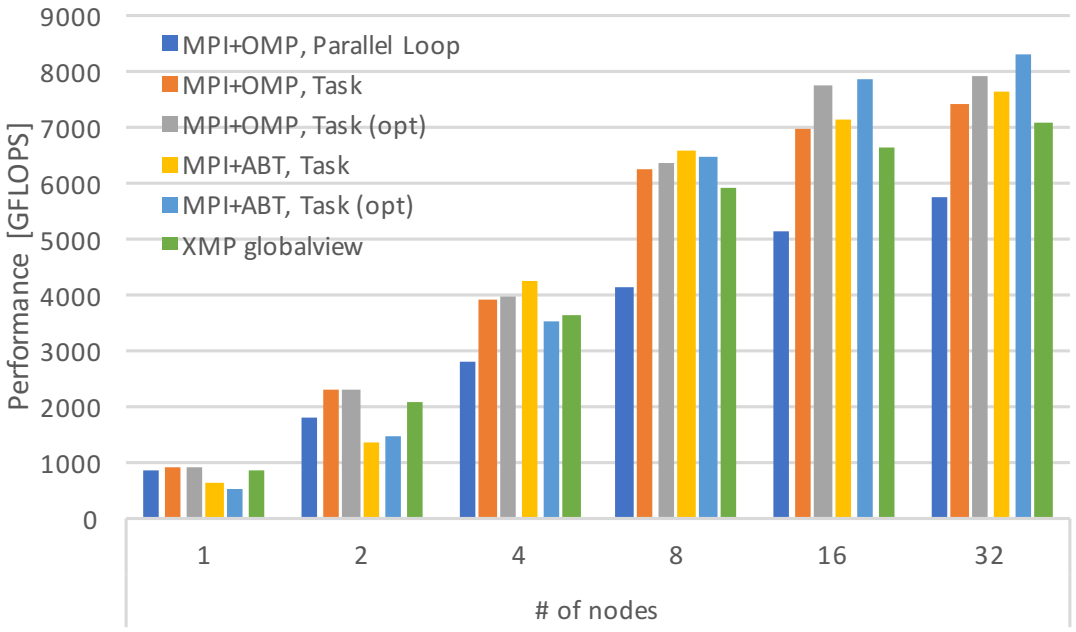
for 文の演算範囲（*k+1* より *nt* まで）よりテンプレート `T(k+1:)` で示されるが，同一ノード内においても *trsm* で配列 *B* を用いているため，自ノードを含む `T(k:)` を `on` 節で記述する．ソースコード 3.7 では，9，12 行目の `tasklet` 指示文の `get_ready`，`get` 節及び 13 行目の `gmove in` 指示文で片側通信とノード間におけるタスクの同期を記述する．通信する範囲はソースコード 3.6 と同一だが，片側通信のため *potrf* と通信タスクの間で `get`，`get_ready` 節による同期が必要となる．*potrf* 演算後に片側通信が実行可能となるため，*potrf* のタスクに `get_ready` 節が記述される．片側通信を実行するのは *trsm* を実行するノードであるため，実行ノードはテンプレート `T(k:)` と表せる．片側通信を実行するタスクに対しては `get` 節が記述され，`get_ready` 節とマッチさせるタグ（例では *k*nt+k*）を指定する．

性能評価

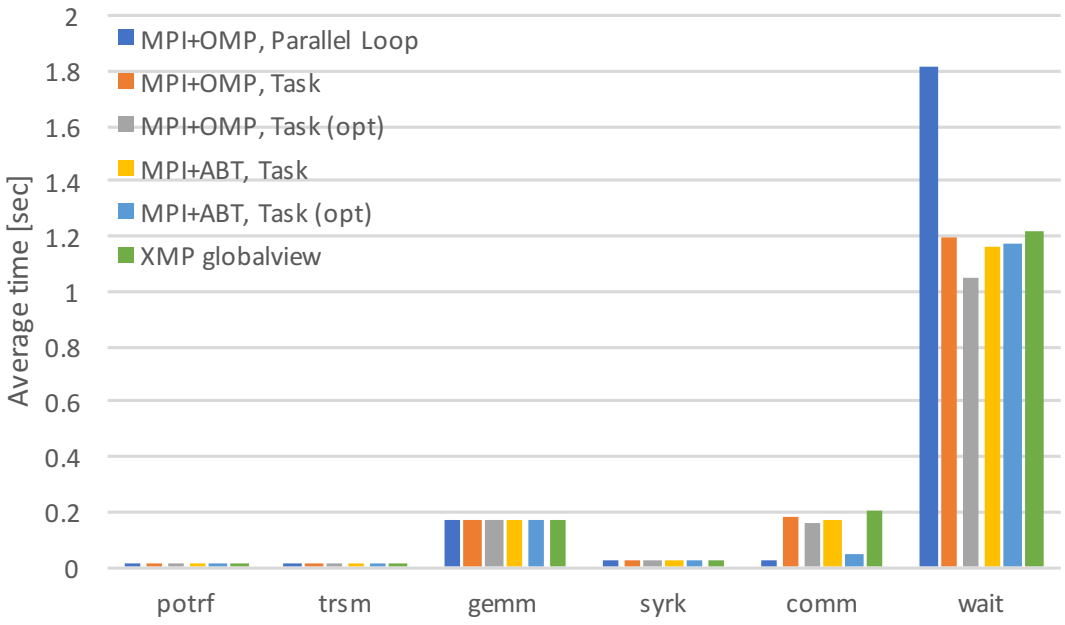
ブロックコレスキー分解の評価に用いる配列は，倍精度浮動小数点数型で 32768×32768，1 ブロックサイズは 512×512 とし，2 次元のサイクリック分割による評価を示す．Omni XMP Compiler のコード変換部は一部未実装箇所があるため，XMP の評価はグローバルビューを除き手動変換のコードによる性能評価となる．OFP では，1 ノード 1 プロセス，1 プロセスあたりのスレッド数を 64 とし，最大 32 ノードを用いる．COMA システムでは，1 ノード 1 プロセス，1 プロセスあたりのスレッド数を 16 とし，最大 16 ノードを用いて評価を行う．性能評価として FLOPS 値と最大ノード数を用いた場合の実行時間の内訳を示す．ブロックコレスキー分解の場合は，プロセス 0 における演算タスク（*potrf*，*trsm*，*gemm* 及び *syrk*），通信時間（*comm*）とスレッドのアイドル状態の時間（*wait*）を全スレッドでの処理毎の平均値を示す．タスク実行の場合，プログラムの実行毎にスレッドで実行される演算が変わるため，マスタースレッドのみの実行時間の内訳では比較とならない．そこで，本研究では全スレッドでの処理毎の平均時間による性能比較とした．

図 3.15，3.16 に OFP，COMA での Send/Recv 通信によるブロックコレスキー分解の性能と，最大ノード実行時の実行時間の内訳を示す．まず，図 3.15(a)，3.16(a) の MPI+OpenMP で記述したループ並列（MPI+OMP, Parallel Loop）とタスク並列（MPI+OMP, Task）による実装の性能を比較する．タスク並列実装はループ並列実装と比較して，OFP では 30%，COMA は 32% 性能が向上している．図 3.15(b)，3.16(b) を見るとスレッドのアイドル時間を表す *wait* が大幅に減少しているのがわかる．これは，ループ並列実装で記述された全体同期をタスク並列によりデータ依存による一対一同期としたことで，同期待ち状態のスレッドが減少し演算タスクの実行が促されて性能が向上したと言える．また，OFP，COMA の両システムにおいても同様に性能が向上しており，マルチ，メニーコアシステムのどちらにおいてもタスク並列により性能向上が見込めることがわかった．

タスク並列モデルの XMP（XMP globalview）と MPI+OpenMP（MPI+OMP, Task）による実装の性能を比較する．図 3.15(a)，3.16(a) より，XMP は MPI+OpenMP 実装と比較して，OFP では 5%，COMA は 30% 性能が低下している．XMP と MPI+OpenMP 実装の異なる点として通信がある．*potrf* と *trsm* 間の通信を例に挙げると，MPI+OpenMP 実装では自ノードが *potrf* の演算結果を保持している場合は，*trsm* ではその値を用いて演算するが，XMP 実装の場合は `tasklet gmove` 指示文にて自ノードを含む *trsm* を演算する全てのノードと通信を実行する．そのため，演算結果を保持しているノードの場合であっても *MPI_Send/Recv()* によりノード内でメモリコピーが発生する．図 3.15(b)，3.16(b) の XMP と MPI+OpenMP 実装を見ると，XMP 実装の通信時間が増加しており，`tasklet gmove` 指示文による不

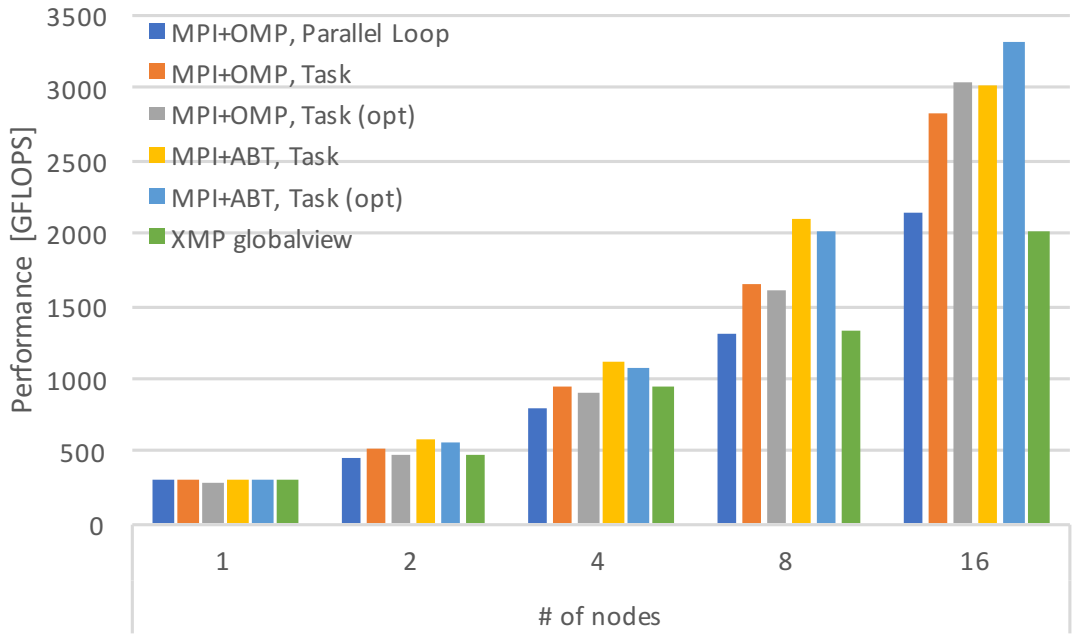


(a) 性能（GFLOPS）.

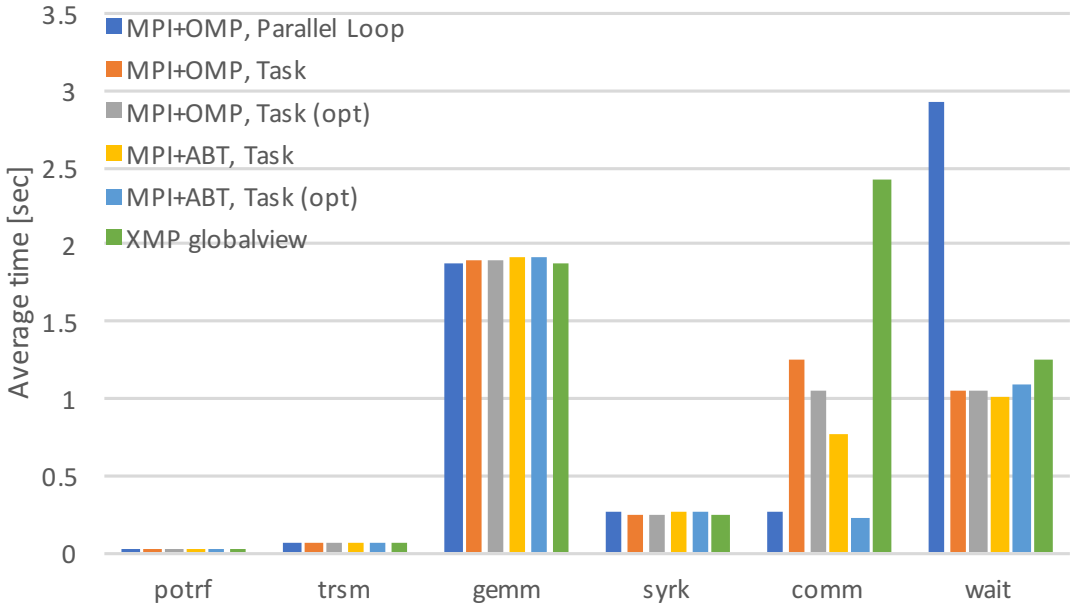


(b) 32 ノード実行時の実行時間の内訳.

図 3.15　Send/Recv 通信によるブロックコレスキー分解の性能評価（Oakforest-PACS）.

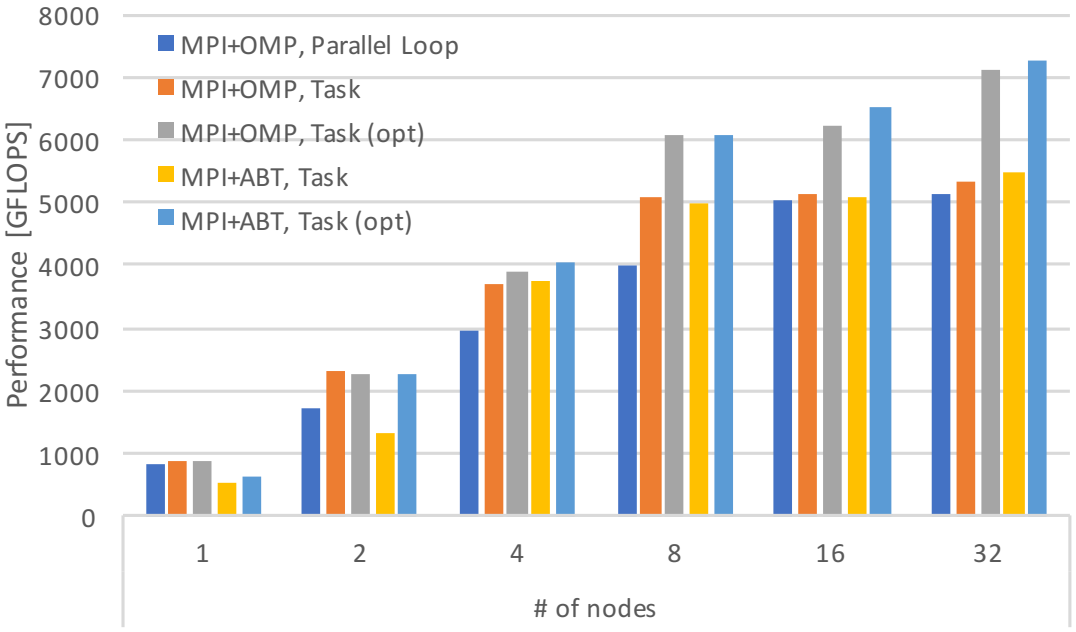


(a) 性能（GFLOPS）.

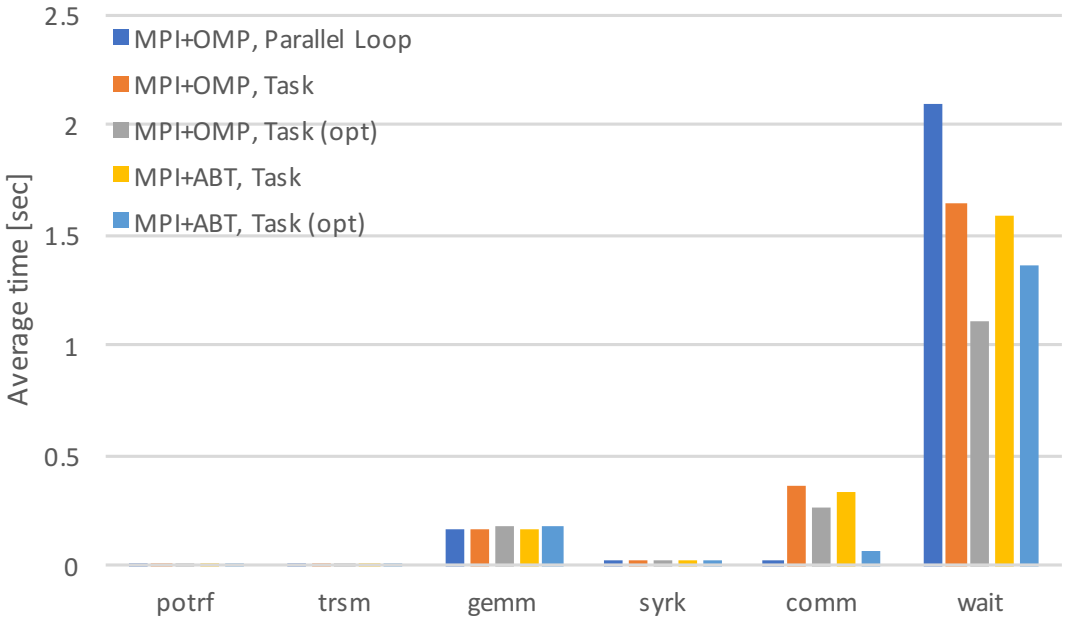


(b) 16 ノード実行時の実行時間の内訳.

図 3.16 Send/Recv 通信によるブロックコレスキー分解の性能評価（COMA）.

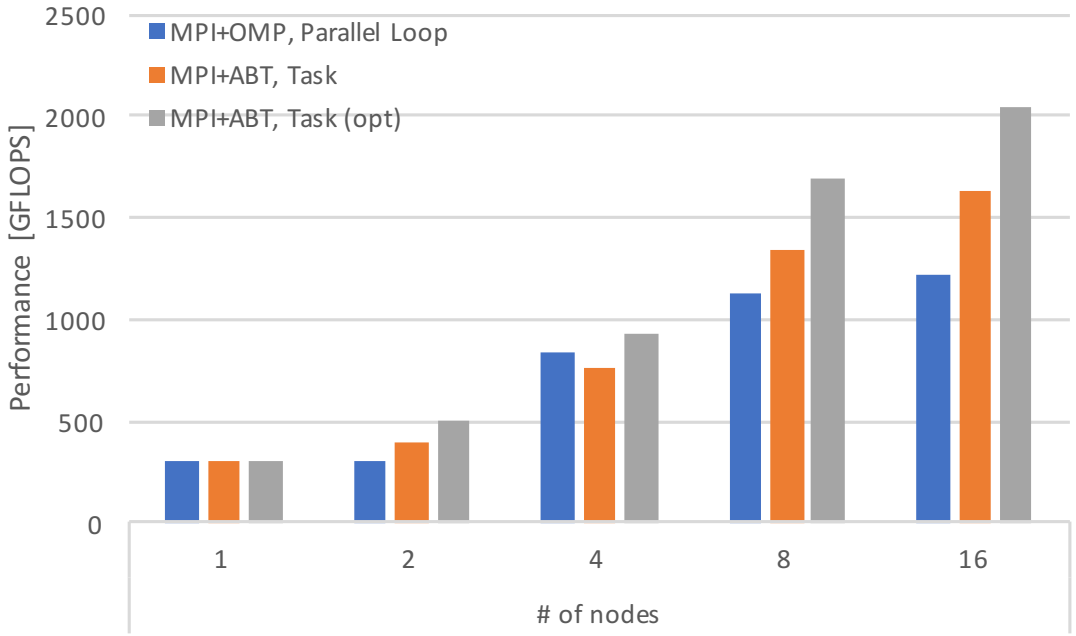


(a) 性能（GFLOPS）.

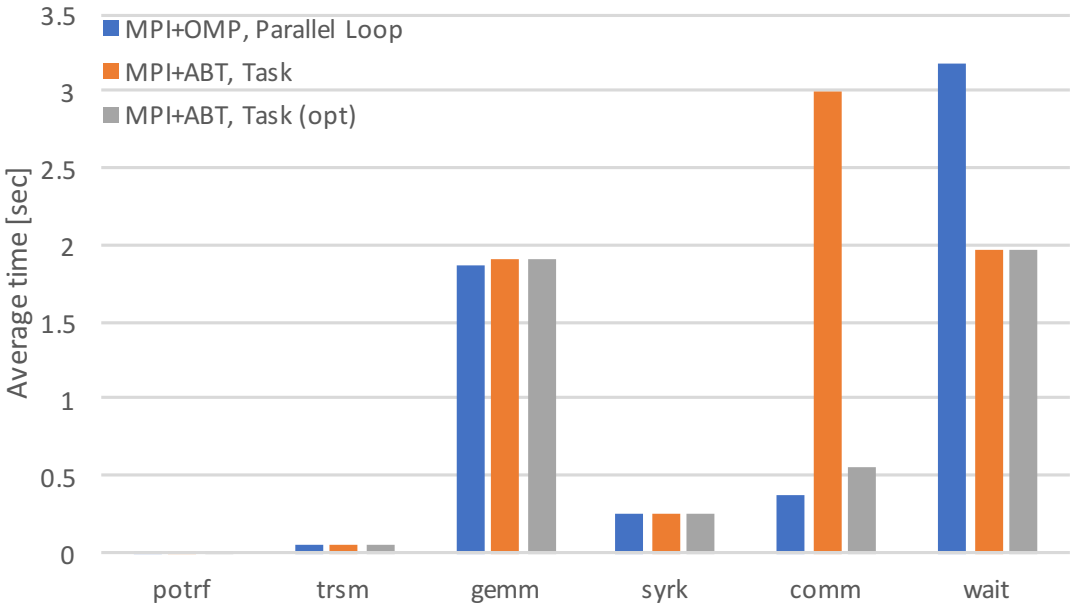


(b) 32 ノード実行時の実行時間の内訳.

図 3.17　Put 通信によるブロックコレスキー分解の性能評価（Oakforest-PACS）.



(a) 性能（GFLOPS）.



(b) 16 ノード実行時の実行時間の内訳.

図 3.18　Put 通信によるブロックコレスキー分解の性能評価（COMA）.

必要な領域の通信によって性能が低下していると言える．また，OFP と比較して COMA での XMP 実装の通信時間は大きく低下している．XMP 実装ではノード内のメモリコピーを含めて *MPI_Send/Recv()* による通信として実行しているため，通信回数自体が MPI+OpenMP 実装と比較して増加している．通信同期の実装で用いた `taskyield` 指示文の挙動は OpenMP では不定であるため，通信同期が必要なタスク数が増加すると通信が終了するまでスレッドを専有するタスクが増加する．また，OFP と比較して COMA は実行スレッド数が少ないため，多くのスレッドが通信タスクによって専有されたことで，演算タスクの実行が遅れ性能が低下したと考えられる．

タスク並列モデルの OpenMP（MPI+OMP, Task）と Argobots（MPI+ABT, Task）による実装の性能を比較する．図 3.15(a)，3.16(a) より，Argobots は OpenMP と比較して，OFP では 3%，COMA は 7% 性能が向上している．ベースラインのループ並列（MPI+OMP, Parallel Loop）による実装との比較では，Argobots は OFP では 34%，COMA は 41% の性能向上が見られた．図 3.15(b)，3.16(b) の実行時間の内訳より，両システムにおける Argobots 実装の通信時間が減少していることがわかる．Argobots を用いたことで，OpenMP では不定だった `taskyield` 指示文によるタスクのスイッチングが必ず実行され，スレッド内においても通信と計算のオーバラップが可能となり性能が向上したと考えられる．

OpenMP，Argobots による実装に対して，通信委譲により 1 スレッドが通信を実行する通信最適化を施した実装（MPI+OMP, Task (opt) と MPI+ABT, Task (opt)）の性能を比較する．通信最適化を行うことで，OFP では OpenMP で 7%，Argobots で 9% 性能が向上し，同様に COMA でも OpenMP で 7%，Argobots で 10% 性能が向上している．ベースラインのループ並列（MPI+OMP, Parallel Loop）による実装と比較をすると，OFP では OpenMP で 38%，Argobots で 45% 向上し，COMA では OpenMP で 41%，Argobots で 55% の性能向上を達成した．1 スレッドに通信を委譲し MPI_THREAD_MULTIPLE から擬似的に MPI_THREAD_FUNNELED とすることで，図 3.15(b)，3.16(b) に示された通信時間 *comm* を削減し性能が向上した．

Put 通信による OFP，COMA でのブロックコレスキー分解の性能と，最大ノード実行時の実行時間の内訳を図 3.17，3.18 に示す．図 3.17(a) の MPI+OpenMP で記述したループ並列（MPI+OMP, Parallel Loop）とタスク並列（MPI+OMP, Task）による実装の性能を比較すると，タスク並列実装はループ並列実装と比較して，OFP では 5% 性能が向上している．しかし，COMA ではタスク実行が終了せず，性能評価を行うことができなかった．この理由として，タスク内における通信同期の実装に用いた OpenMP `taskyield` 指示文の動作が不定であり，依存関係がない通信タスクが多くのスレッドで同時に起動したことによる通信のデッドロックが発生したことが挙げられる．OFP と比較して COMA は実行スレッド数が少ないため，多くのスレッドが通信タスクに専有されてしまい，COMA のみデッドロックが発生したと考えられる．

タスク並列モデルの OpenMP（MPI+OMP, Task）と Argobots（MPI+ABT, Task）による実装の性能を比較する．図 3.17(a)，3.18(a) より，Argobots は OpenMP と比較して，OFP では 3% 性能が向上し，COMA では少ない実行スレッド数におけるスレッドレベルでの通信が多発するプログラムにおいても安全に実行可能であることを示した．ベースラインのループ並列（MPI+OMP, Parallel Loop）による実装との比較では，Argobots は OFP では 8%，COMA は 34% の性能向上が見られた．Argobots を用いたことで，OpenMP では不定だった `taskyield` 指示文によるタスクスイッチが必ず実行され，Send/Recv 通信による実装と同様にスレッド内においても通信と計算のオーバラップが可能となり性能が向上したと考えら

れる．

OpenMP，Argobots による実装に対して，通信委譲により 1 スレッドが通信を実行する通信最適化を施した実装（MPI+OMP, Task (opt) と MPI+ABT, Task (opt)）の性能を比較する．COMA の場合は，OpenMP による実行はデッドロックを起こしたため，Argobots のみの比較を示す．通信最適化を行うことで，OFP では OpenMP で 33%，Argobots で 32% 性能が向上し，同様に COMA でも Argobots は 25% 性能が向上している．ベースラインのループ並列（MPI+OMP, Parallel Loop）と比較をすると，OFP では OpenMP で 39%，Argobots で 49% 向上し，COMA の Argobots では 68% の性能向上を達成した．Send/Recv と同様に Put 通信の場合でも，片側通信実行のための通知による通信と Put を 1 スレッド上で実行することで性能が向上することを示した．

3.6.3 ラプラスソルバの性能評価

ラプラスソルバは，2 次元ラプラス方程式をヤコビ法を用いて解くプログラムである．典型的なステンシル問題であり，2 次元格子状における近傍 4 点の平均による値の更新が主な演算である．分散メモリ環境で実行する場合は，分散境界の値の更新に隣接プロセスが持つ領域が必要となる．この領域は袖領域として実装され，イテレーション毎に更新されるステンシル演算の結果を隣接プロセス間で通信することで，次イテレーションで演算に用いる値を保持する．ヤコビ法では空間全体の一斉更新があるため，イテレーション毎に値を一時的に保持する必要がある．従ってラプラスソルバでは，一時的なデータ保持のためのデータコピー（*copy*），袖領域を用いた隣接プロセス間での通信（*comm*）及び 4 点ステンシル演算（*calc*）の 3 種類で構成される．また，本研究で用いるラプラスソルバは MPI+OpenMP のループ並列による実装がされており，比較のためタスク並列による実装を新たに行った．両実装において，演算領域を仮想的にブロック化し，ブロック単位で演算を行うことでメモリアクセスの最適化を行うキャッシュブロッキングを適用した．

`tasklet` 指示文による実装

ソースコード 3.8，3.9 に `tasklet` 指示文によるラプラスソルバの実装例を示す．ソースコード 3.8 は，グローバルビューによるデータ分散と `tasklet`，`tasklet reflect` 指示文によるタスク並列実装である．3 から 5 行目の `template`，`distribute` 及び `align` 指示文により，サイズ *XSIZE* × *YSIZE* の 2 次元配列 *u*，*uu* の 2 次元ブロック分散が行われる．また，1 イテレーション前のステンシル演算の結果を保持する配列 *uu* には，`shadow` 指示文により袖領域が指定されており，20 行目のように `tasklet reflect` 指示文による袖領域通信が実行可能である．タスク並列を実行する演算範囲は 9 行目から始まる for ループであるため，そのループに対して `tasklets` 指示文を指定する．1 タスクが実行する演算の粒度は，キャッシュブロッキングされたブロックサイズとする．よって，10，22 行目に表される `loop` 指示文により各ノードで for ループの分散実行が開始され，ノード内では 13，25 行目の `tasklet` 指示文により 1 ブロックの演算がタスクとして実行される．`tasklet` 指示文の実行ノードは対象ブロックを保持するノードのみであるため，`on` 節では対象ブロックの開始インデックスを表すテンプレートを指定する．ブロック単位での演算では隣接ノードが実態を持つ袖領域と同様に，ブロック境界の値の更新に隣接するブロックが持つ値が必要となる．そのため，30 から 32 行目のステンシル演算を実

ソースコード 3.8 `tasklet reflect` 指示文によるラプラスソルバの実装例.

```
1  double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
2  #pragma xmp nodes P(*, *)
3  #pragma xmp template T(0:YSIZE−1, 0:XSIZE−1)
4  #pragma xmp distribute T(block, block) onto P
5  #pragma xmp align [i][j] with T(j, i) :: u, uu
6  #pragma xmp shadow uu[1][1]
7
8  #pragma xmp tasklets
9  for (int k = 0; k < n; k++) {
10 #pragma xmp loop (by, bx) on T(by, bx)
11     for (int bx = 0; bx < XSIZE; bx+=bs)
12         for (int by = 0; by < YSIZE; by+=bs)
13 #pragma xmp tasklet in(u[bx:bs][by:bs]) \
14                         out(uu[bx:bs][by:bs]) on T(by, bx)
15             /* Calculate begin and end indices of the block from bx and by */
16             for (int x = x_begin; x < x_end; x++)
17                 for (int y = y_begin; y < y_end; y++)
18                     uu[x][y] = u[x][y];
19
20 #pragma xmp tasklet reflect(uu) chunksize(bs, bs)
21
22 #pragma xmp loop (by, bx) on T(by, bx)
23     for (int bx = 0; bx < XSIZE; bx+=bs)
24         for (int by = 0; by < YSIZE; by+=bs)
25 #pragma xmp tasklet in(uu[bx:bs][by:bs], \
26                         uu[bx−bs:bs][by:bs], uu[bx+bs:bs][by:bs], \
27                         uu[bx:bs][by−bs:bs], uu[bx:bs][by+bs:bs]) \
28                         out(u[bx:bs][by:bs]) on T(by, bx)
29             /* Calculate begin and end indices of the block from bx and by */
30             for (int x = x_begin; x < x_end; x++)
31                 for (int y = y_begin; y < y_end; y++)
32                     u[x][y] = (uu[x−1][y] + uu[x+1][y] + uu[x][y−1] + uu[x][y+1])/4.0;
33 }
```

ソースコード 3.9 `put`，`put_ready` 節によるラプラスソルバの実装例.

```
1  double u[XSIZE][YSIZE], uu[XSIZE][YSIZE]:[*];
2  #pragma xmp nodes P(*)
3
4  #pragma xmp tasklets
5  for (int k = 0; k < n; k++) {
6      for (int bx = 0; bx < XSIZE; bx+=bs)
7          for (int by = 0; by < YSIZE; by+=bs)
8  #pragma xmp tasklet in(u[bx:bs][by:bs]) out(uu[bx:bs][by:bs])
9              /* Calculate begin and end indices of the block from bx and by */
10             for (int x = x_begin; x < x_end; x++)
11                 for (int y = y_begin; y < y_end; y++)
12                     uu[x][y] = u[x][y];
13
14     for (int by = 0; by < YSIZE; by+=bs) {
15 #pragma xmp tasklet in(uu[0:bs][by:bs]) put(by)
16         uu[xmax-1][by*bs:bs]:[upper] = uu[1][by*bs:bs];
17 #pragma xmp tasklet out(uu[XSIZE/bs-bs:bs][by:bs]) \
18                         put_ready(uu[XSIZE/bs-bs:bs][by:bs], P(lower), by)
19
20 #pragma xmp tasklet in(uu[XSIZE/bs-bs:bs][by:bs]) put(by + offset)
21         uu[0][by*bs:bs]:[lower] = uu[xmax-2][by*bs:bs];
22 #pragma xmp tasklet out(uu[0:bs][by:bs]) \
23                         put_ready(uu[0:bs][by:bs], P(upper), by + offset)
24     }
25
26     for (int bx = 0; bx < XSIZE; bx+=bs)
27         for (int by = 0; by < YSIZE; by+=bs)
28 #pragma xmp tasklet in(uu[bx:bs][by:bs], \
29                            uu[bx-bs:bs][by:bs], uu[bx+bs:bs][by:bs], \
30                            uu[bx:bs][by-bs:bs], uu[bx:bs][by+bs:bs]) \
31                        out(u[bx:bs][by:bs])
32             /* Calculate begin and end indices of the block from bx and by */
33             for (int x = x_begin; x < x_end; x++)
34                 for (int y = y_begin; y < y_end; y++)
35                     u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])/4.0;
36 }
```

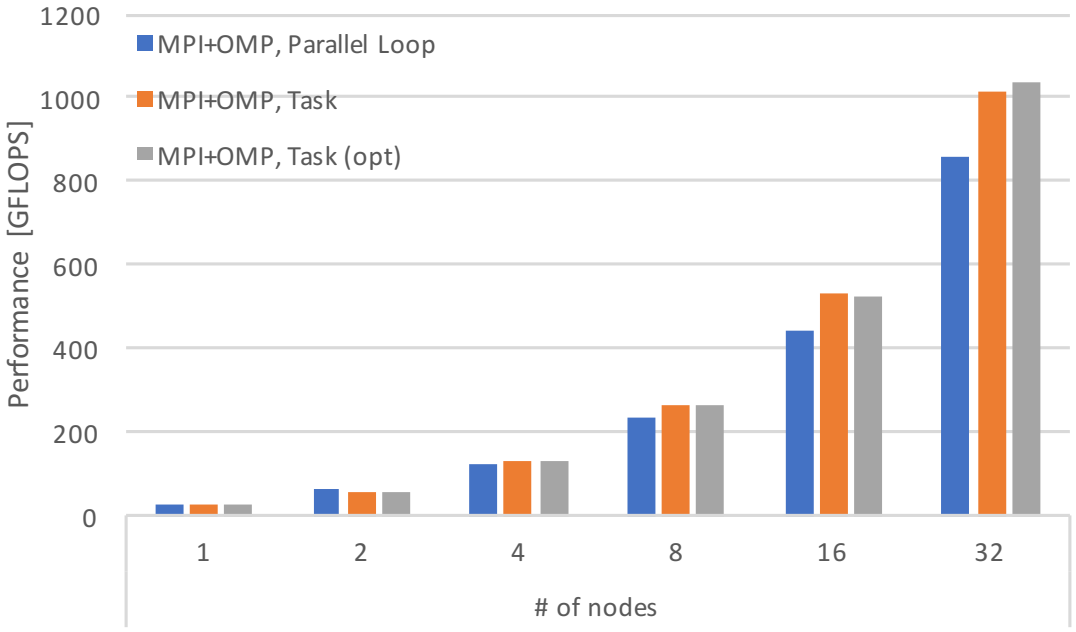
行するタスクの依存関係は，25 から 28 行目のように演算するブロックの他に隣接 4 ブロックの演算範囲を依存関係として記述する．20 行目の `tasklet reflect` 指示文により，隣接ノードが持つ境界値を自ノードが持つ袖領域へと更新を行う袖領域通信をタスクとして実行する．ここで，依存関係は全てブロックサイズ毎に記述されているが，袖領域通信は最外ブロックが持つ要素の一部のみであるため，ユーザ記述と XMP ランタイムが生成する依存関係がマッチしない．そこで，`tasklet reflect` 指示文において演算の依存関係がどのように記述されているかを XMP ランタイムへと知らせる `chunksize` 節を用いて依存記述がブロック単位であることを指定することで，最外ブロックの演算と袖領域通信の間の依存関係を示す．

ソースコード 3.9 は，ローカルビューの coarray/Put と `tasklet` 指示文，`put`，`put_ready` 節による実装例である．ソースコード簡略化のため 1 次元ブロック分割の例を示す．また予め，配列 *u*，*uu* のサイズ *XSIZE* が分散された値とする．データコピーやステンシル演算部の `tasklet` 指示文によるタスク並列実装は，ループ領域の全体が自ノードの演算範囲となるため `on` 節は必要ない．それ以外はソースコード 3.8 と同等であるため説明は省略する．通信は 16，21 行目の coarray/Put で記述され，隣接ノード *upper*，*lower* 同士でブロック単位の境界要素の交換を行う．18 行目の `put_ready` 節によりノード *lower* に対して *uu[0:bs][by:bs]* への Put が可能であることを知らせ，Put 完了までタスクは保持される．ノード *lower* では，15 行目の `put` 節でノード *upper* からの通信可能通知を受け取り，16 行目の Put を実行し，最終的にノード *upper* への完了通知を出すことでタスクの実行は終了する．

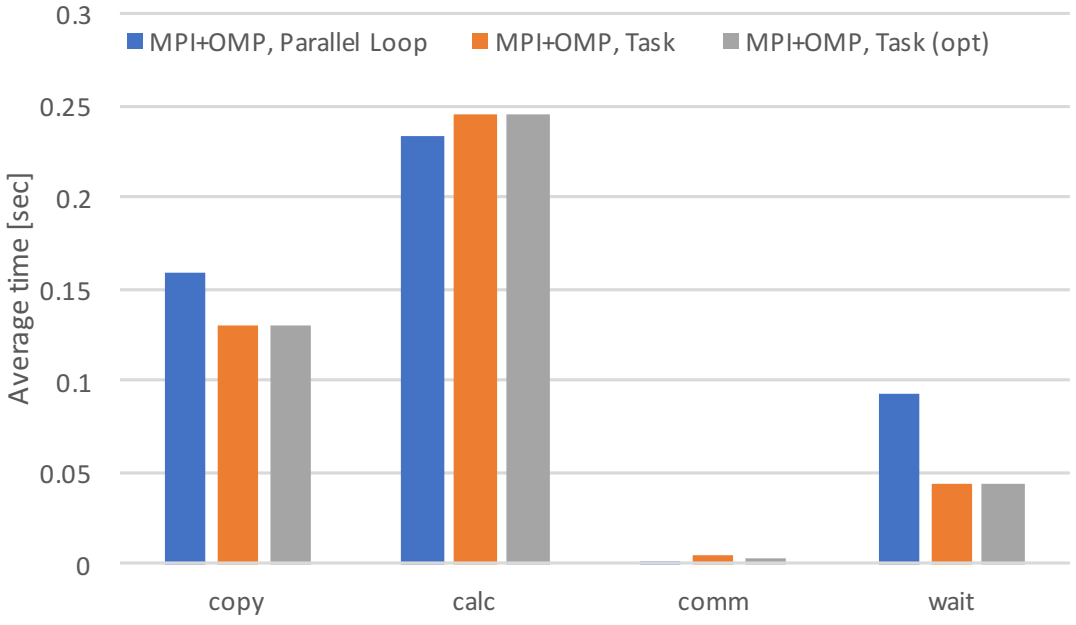
性能評価

ラプラスソルバの評価に用いる配列は，倍精度浮動小数点数型で 32768×32768，1 ブロックサイズは 512×512 とし，2 次元のブロック分割による評価を示す．XMP 実装，Argobots の適用による評価は完了しておらず，評価は手動変換による MPI+OpenMP コードの性能評価のみとなる．ブロックコレスキー分解での評価と同様に，OFP システムでは 1 ノード 1 プロセス，1 プロセスあたりのスレッド数を 64 とし最大 32 ノードを用いる．COMA システムでは 1 ノード 1 プロセス，1 プロセスあたりのスレッド数を 16 とし最大 16 ノードを用いて評価を行う．性能評価として FLOPS 値，最大ノード数を用いた場合の実行時間の内訳を示す．ラプラスソルバの場合は，プロセス 0 における演算（*copy* と *calc*），通信時間（*comm*）とスレッドのアイドル状態の時間（*wait*）の全スレッドでの処理毎の平均値を示す．

図 3.19，3.20 に OFP，COMA での Send/Recv 通信によるラプラスソルバの性能と，最大ノード実行時の実行時間の内訳を示す．図 3.19(a)，3.20(a) の MPI+OpenMP によるループ並列（MPI+OMP, Parallel Loop）とタスク並列（MPI+OMP, Task）による実装の性能を比較すると，OFP ではタスク並列により 18% 性能が向上したが，COMA では 13% 性能が低下した．図 3.19(b) に示される OFP の実行時間の内訳では，スレッドのアイドル時間を表す *wait* が大幅に減少していることがわかる．ブロックコレスキー分解の評価と同様に，ループ並列で記述された全体同期をデータ依存によるタスク並列の一対一同期としたことで，同期待ち状態のスレッドを減少させ性能が向上したと言える．しかし，図 3.20(b) の COMA での実行時間の内訳では，タスク並列により *wait* の時間は減少しているが *wait* 自体の実行時間は短く，そもそもの実行においてロードインバランスが発生していないことがわかる．そのため，タスク実行によるオーバヘッドやマルチスレッド通信による通信時間の増加により性能が低下した．次に，MPI+OpenMP のタスク並列（MPI+OMP, Task）と，その実装に対して通信最適化を行った実装（MPI+OMP, Task (opt)）

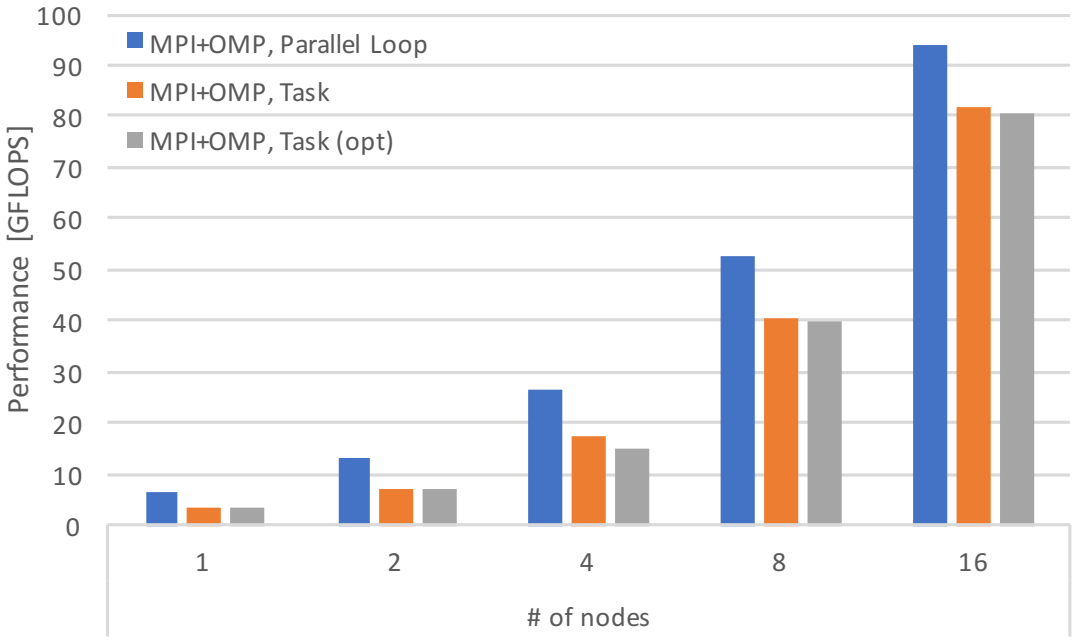


(a) 性能（GFLOPS）.

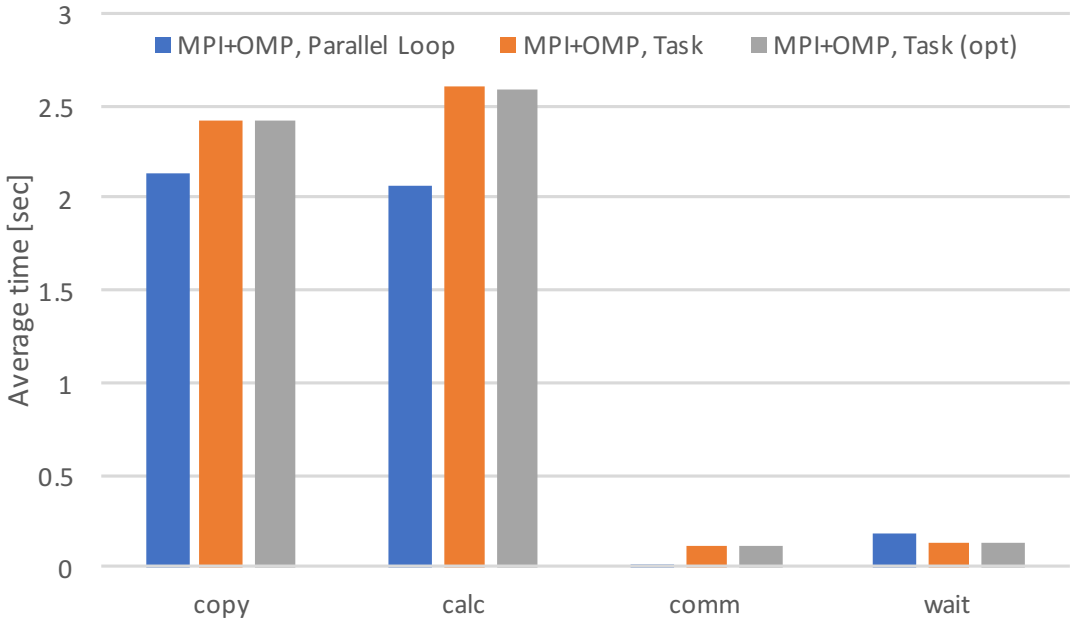


(b) 32 ノード実行時の実行時間の内訳.

図 3.19 Send/Recv 通信によるラプラスソルバの性能評価（Oakforest-PACS）.

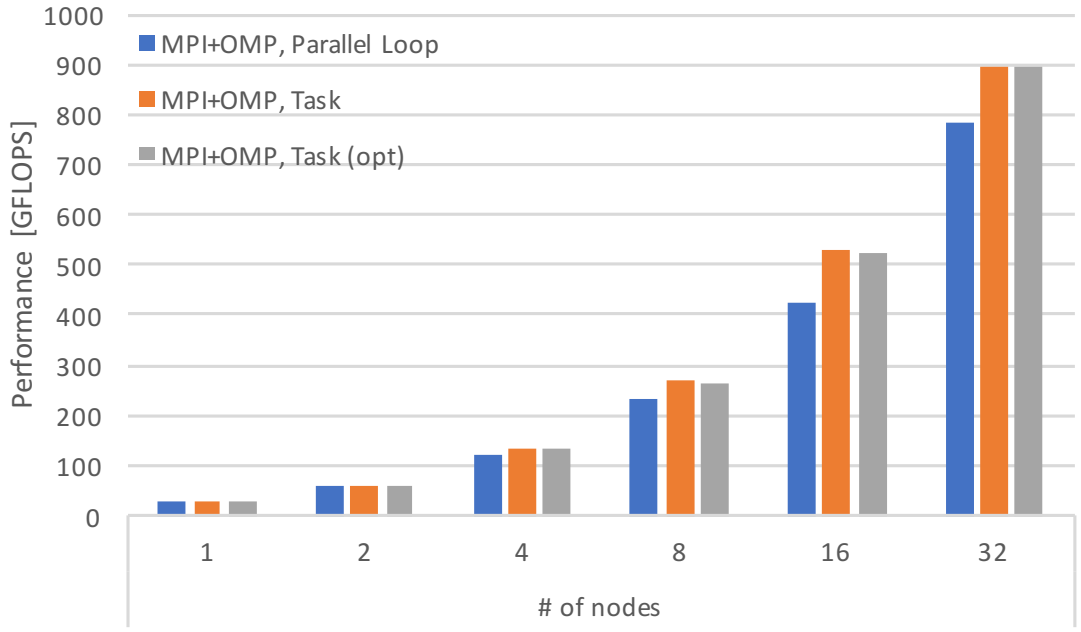


(a) 性能（GFLOPS）.

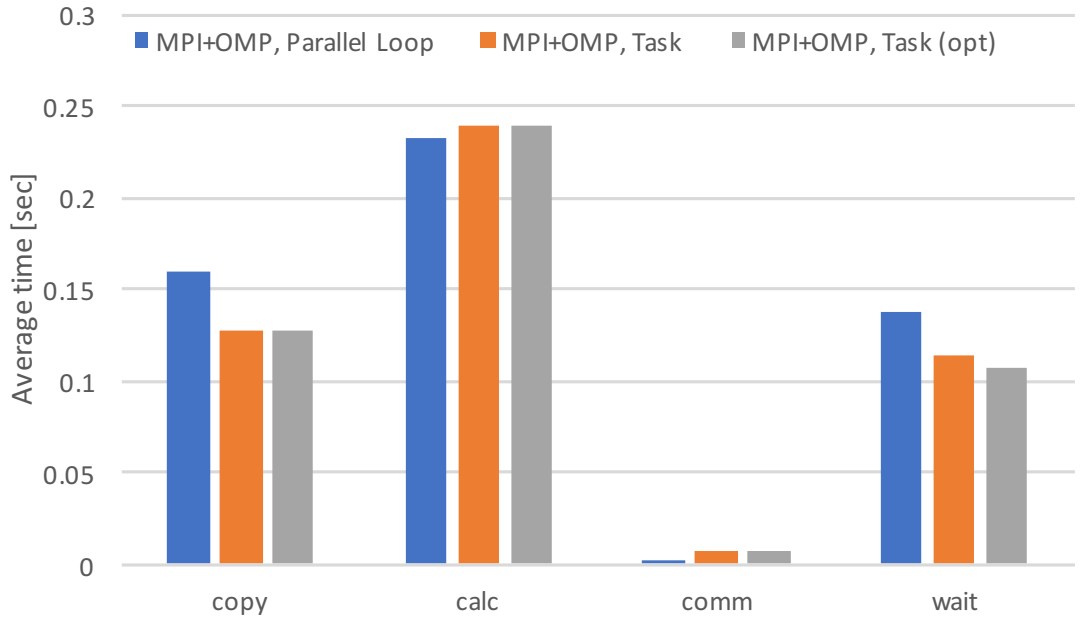


(b) 16 ノード実行時の実行時間の内訳.

図 3.20　Send/Recv 通信によるラプラスソルバの性能評価（COMA）.

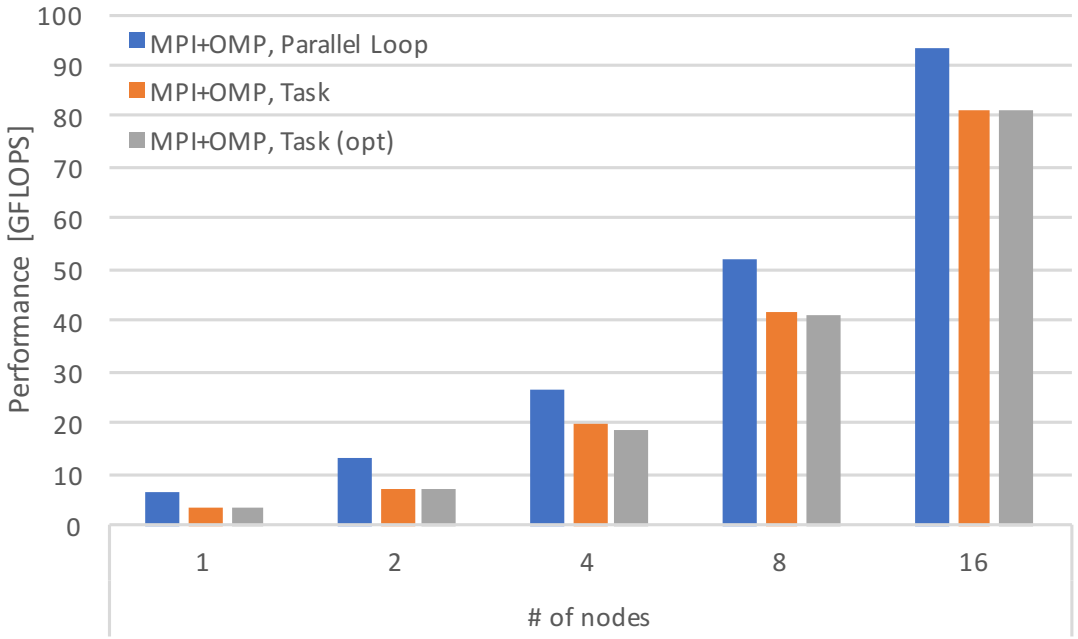


(a) 性能（GFLOPS）.

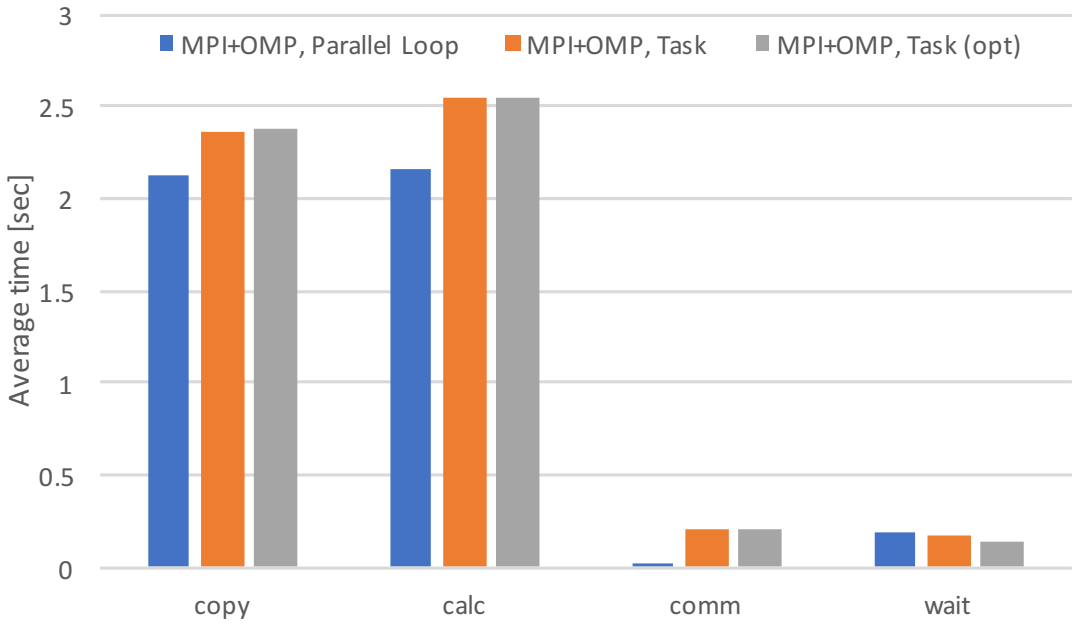


(b) 32 ノード実行時の実行時間の内訳.

図 3.21 Put 通信によるラプラスソルバの性能評価（Oakforest-PACS）.



(a) 性能（GFLOPS）.



(b) 16 ノード実行時の実行時間の内訳.

図 3.22　Put 通信によるラプラスソルバの性能評価（COMA）.

表 3.3 ブロックコレスキー分解における各実装の Delta-SLOC.

	Serial	OMP	MPI+OMP	XMP
SLOC	318	330	582	338
modified	-	0	13	2
added	-	12	264	20
deleted	-	0	0	0
Total Delta-SLOC	-	12	277	22

との比較を示す．OFP，COMA の両システムにおいて通信最適化の前後で性能差はみられなかった．ラプラスソルバの 1 イテレーションあたりの通信はブロックコレスキー分解と比較して，2 次元空間における上下左右の隣接 4 ノードとの袖領域通信のみと少ない．図 3.19(b)，3.20(b) の通信時間 *comm* を見ても，全ての実装において通信時間に大きな差はなく，マルチスレッド実行において通信性能が低下しなかったことが理由として挙げられる．

図 3.21，3.22 に OFP，COMA での Put 通信によるラプラスソルバの性能と，最大ノード実行時の実行時間の内訳を示す．図 3.21(a)，3.22(a) の MPI+OpenMP によるループ並列（MPI+OMP, Parallel Loop）とタスク並列（MPI+OMP, Task）による実装の性能を比較すると，OFP ではタスク並列により 14% 性能が向上したが，COMA では 13% 性能が低下した．Send/Recv 通信による評価と同様で，OFP では図 3.21(b) より，スレッドのアイドル時間 *wait* が減少しており，ループ並列の全体同期からタスク並列による一対一同期としたことで性能が向上した．しかし，図 3.22(b) の COMA では，タスク並列により *wait* の時間は多少減少しているが *wait* 自体の実行時間は短く，そもそもの実行においてロードインバランスが発生していないことがわかる．次に，MPI+OpenMP のタスク並列（MPI+OMP, Task）と，その実装に対して通信最適化を行った実装（MPI+OMP, Task(opt)）を比較する．結果として，OFP，COMA の両システムにおいて通信最適化の前後で性能差はみられなかった．この理由についても Send/Recv 通信による実装と同様に，1 イテレーションあたりの通信が各隣接ノードに対する 1 回の通信であり，ブロックコレスキー分解と比較して少なかったために，1 スレッドに通信を委譲する最適化によって性能が向上しなかったと考えられる．

3.6.4 生産性

メニーコアシステムを含む分散メモリ環境でタスク並列プログラミングをする場合，ノード内は OpenMP，ノード間は MPI と異なるメモリ空間向けのプログラミングモデルを組み合わせて記述する必要がある．MPI による煩雑なデータ分散や並列実行の記述が必要となり，さらにノード間でのタスク依存を実現するためにノード内のタスク依存記述と複雑な通信記述を組み合わせた通信タスクの記述が必要となる．また，タスク依存の記述によっては，前後で記述されたタスクの実行順序が逆になるなど，プログラムのフローに依らない記述も可能であるため，プログラムのバグを減らすためにも通信とタスク依存を組み合わせた直感的で簡易な記述が求められる．

本研究で提案する XMP におけるタスク並列モデルでは，グローバルビューとローカルビューのそれぞ

表 3.4　ラプラスソルバにおける各実装の Delta-SLOC.

	Serial	OMP	MPI+OMP	XMP
SLOC	119	138	245	146
modified	-	0	16	2
added	-	19	126	27
deleted	-	0	0	0
Total Delta-SLOC	-	19	142	29

れの通信に基づく通信タスクの記述方法を提供している．グローバルビューによる `tasklet gmove`，`tasklet reflect` 指示文の場合は，逐次プログラムのデータコピーのような配列代入文形式での通信や，分散配列を指定するだけの袖領域通信をタスク上で実現している．`tasklet gmove` 指示文では，通常のタスク依存記述と同様に配列代入文で指定した配列を依存として記述するだけであり，`tasklet reflect` 指示文の場合は XMP ランタイムが自動的に依存関係を指定する．従って，OpenMP のタスク並列モデルのように，タスク内で記述された代入文を依存関係として記述するのと同様の記述で，分散メモリ環境におけるタスク依存が記述可能となった．ローカルビューによる `put`，`put_ready`，`get` 及び `get_ready` 節の場合は，タスク内での片側通信を実行するための同期構文を提供している．片側通信の Put，Get をタスク内で実行する場合の 3 種類の通信をユーザが全て記述するのは困難であり，記述も煩雑である．そこで，ユーザ記述のタグマッチングにより 2 ノード間での通信の整合性を自動でとる仕様としたことで，ユーザが全ての通信を明示的に記述する負担を軽減している．

`tasklet` 指示文による実装の生産性を定量的に評価するため，各プログラミングモデルによる実装のコード行数を比較する．比較手法として 2.5.4 節の GTC-P の生産性の評価と同様に Delta-SLOC 方式を用いる．表 3.3 にブロックコレスキー分解，表 3.4 にラプラスソルバのそれぞれの実装の行数と逐次プログラムからの差分を示す．表 3.3 における XMP は `tasklet gmove` 指示文による実装，表 3.4 では `tasklet reflect` 指示文による実装を表す．全体の行数として，両ベンチマークの MPI+OpenMP 実装は逐次プログラムと比較して数百行以上増加しているのに対して，XMP は OpenMP 実装とほぼ同等の行数増加で実装を可能としている．この理由として，データ分散や並列実行がグローバルビューの指示文のみで実行が可能になったことや，通信対象の決定やソースコード 3.1 のような通信と同期を指示文変換で XMP ランタイムが吸収している点が挙げられる．また，ラプラスソルバの XMP 実装では `tasklet reflect` 指示文により，不連続領域の通信におけるデータのパッキング/アンパッキングを含む 2 次元空間の 4 方向の袖領域通信を 1 行で表すことができる点がコード行数削減の多くを占めている．Delta-SLOC の評価を見ても，XMP は OpenMP 実装と同等のコード変更量でブロックコレスキー分解，ラプラスソルバともに実装できていることがわかる．XMP のローカルビューによる実装は 2.5.4 節のローカルビューの生産性の評価と同様に，coarray に用いる配列宣言時においてプログラムの書き換えが起こる．また `tasklet` 実装においては，ノードローカルの通信バッファを演算に用いる実装としたため，その 2 点のみが逐次実装からの変更点である．しかし，片側通信を実行するための 2 回の同期を指示文の節としたことでの行数の削減や，提案指示文内で片側通信の同期が自動的に取られるなど，MPI と比

較して簡易な実装とすることができた．

3.7 関連研究

分散メモリ環境におけるタスク並列を記述可能なプログラミング言語やランタイムライブラリは様々開発が進められている．StarPU[57] は，Institut National de Recherche en Informatique et en Automatique（INRIA）で開発が進められている CPU や演算加速機構を対象にタスク並列や負荷分散を記述可能なランタイムライブラリである．記述方法として *codelet* と呼ばれる構造体を使用し，実行関数，実行リソースや依存関係などを予め記述し，タスク生成時に用いる方法をとる．実行モデルは OpenMP 同様にデータ依存によるタスク並列モデルを採用しており，逐次実行に基づくデータの read/write（*STARPU_R*，*STARPU_W* 及び *STARPU_RW* を指定）で依存関係を構築する．分散メモリ環境で実行する場合には，StarPU と MPI を組み合わせて明示的に通信を記述する．また StarPU に対して，実行時に全てのノードが全てのタスクに到達するという制約を付けてデータ分散を予め指定しておくことで，全ノードが全く同じタスクグラフを生成することが可能となり，そのグラフを基に StarPU がノード間でのデータ依存を判断し，通信を自動生成することも可能である．Queuing and Runtime for Kernels on Distributed（QUARK-D）[58] はテネシー大学により開発が進められている分散メモリ環境を対象としたタスク並列処理を記述可能なランタイムライブラリである．先行研究である QUARK[59] は共有メモリ環境のみを対応としていたが，QUARK のタスク記述に対してタスクの *ID* やタスク内の演算で用いる分散データを持つノード番号を明示することで，分散メモリ環境に対応している．実行モデルは StarPU，OpenMP と同様にデータ依存に基づくタスク並列モデルを採用しており，*INPUT*，*OUTPUT* 及び *INOUT* でデータ依存を記述する．StarPU[*1] や QUARK-D はどちらもランタイムライブラリであり，タスクとして実行する関数，データ依存，実行リソースなど多くの制御構文を関数形式で全て明示的に記述する必要がある．本研究で提案するタスク並列モデルは，PGAS モデルを基にした指示文ベースの記述を提供する．従って，逐次の実装を維持しつつ分散メモリ環境におけるプログラミングが可能であり，タスク依存も指示文による簡易な記述となっている．

UPC++[4] は，UPC++ Specification Working Group によって提案されている C++ ベースの PGAS 言語であり，その機能の一つとしてタスク並列実行を提供している．UPC++ では分散メモリ環境でのタスク並列記述のために *event-driven* と *finish-async* の 2 種類のプログラミングモデルを採用している．*event-driven* モデルは Phalanx[60]，*finish-async* モデルは X10[7] のプログラミングモデルを踏襲している．*event-driven* モデルは *event* によりタスク依存を記述する．UPC++ が提供する *async()*，*async_after()* に，関数で記述されたタスクと *event* を記述することで．*async()* に記述されたタスクの実行後に *async_after()* のタスクの実行が開始される．一方で，*finish-async* モデルは *finish* 構文内に複数のタスクが記述され，構文内に記述されたタスクは並列に実行される．*finish* 構文は完了同期も含んでおり，全てのタスクの終了後に *finish* 構文を抜ける．*event-driven*，*finish-async* モデルと本研究で提案するタスク並列モデルの比較を示す．*finish-async* モデルはタスクの依存関係を記述可能なモデルではない．また，*event-driven* モデルは依存関係は記述可能だが，依存記述はデータ依存ではなく別途 *event* を用意しタスクフローを記述する

[*1] StarPU は C 向けにデータ依存を記述可能な指示文も提供しているが，分散メモリ環境には対応していない．

必要がある．タスク内で実行される演算とは関係のない *event* の指定が必要となるため，データ依存で記述する提案モデルと比較すると依存記述を直感的に記述することは難しいと考えられる．

OmpSs[61] は，Barcelona Supercomputing Center（BSC）が研究開発を進めている共有メモリやデバイス向けのデータ依存に基づくタスク並列プログラミングモデルである．OmpSs ではデータ依存を `depend` 節ではなく `in`，`out` 及び `inout` 節で記述する以外は，3.1.1 節の OpenMP によるデータ依存と同等の記述である．OpenMP と異なる点として実行モデルが挙げられる．OpenMP は `parallel` 指示文による *fork-join* モデルであるが，OmpSs は *thread-pool* モデルを採用している．*fork-join* モデルでは `parallel` 指示文を記述することでスレッドが生成され，そのブロック内でスレッドが並列に実行される．一方で *thread-pool* モデルでは，プログラム実行開始時に全スレッドが生成され，マスタースレッドがプログラムを実行する．その他のスレッドは，マスタースレッドがワークシェアリングが指定されたループやタスクに到達し，並列実行が必要となるまで待機する．従って，OpenMP のように明示的に `parallel` 指示文を記述する必要がない．分散メモリ環境で実行する場合には OmpSs のタスク内で MPI の P2P 通信を記述する方法が取られる [52]．MPI の通信をノード間の依存関係とし，通信の完了をノード間の依存関係が解消したとしている．この場合，実行モデルは本研究での提案モデルと同等である．しかし，記述は OmpSs+MPI が必要となり，OmpSs のデータ依存記述に加えて，ノード間のデータ依存を MPI 通信で全て明示的に管理する必要がある．提案モデルではノード内/間の依存関係を XMP で統一的に記述可能とする．

OmpSs[62] ではさらに，*master-slave* モデルによる分散メモリ環境に対応したタスク並列モデルも提案している．このモデルでは，OmpSs の `task` 指示文が記述されたタスクに `target` 指示文を追加するのみで分散メモリ環境で実行可能となる．実行ノードは，マスターノードとスレーブノードに分類され，マスターノードが全てのタスクを展開しスレーブノードに対して，バッファ確保，データコヒーレンシをとるための通信及びタスク実行のためのシグナルを送る．従って，プログラム全体で使用可能なメモリ量はマスターノードのメモリ量と同等となり，データ分散や通信は全て OmpSs ランタイムが決定し暗黙に実行される．並列プログラミングの経験が少ないユーザにとって，暗黙のデータ分散や通信は簡易に並列実行が実現可能であり有用である．しかし，暗黙に実行されるプログラムの挙動を追うことは難しく，性能解析やチューニングを困難にする．本研究で提案する PGAS 言語 XMP に基づくタスク並列モデルでは，データ分散や並列実行，通信・同期は簡易な記述の指示文で全て明示的に示す必要がある．それは指示文を記述しなければ意図しない動作が起こらないことを示す．従って，プログラムの挙動が追いやすく，デバッグや性能チューニングがしやすいという特徴がある．

タスク並列を記述可能な共有メモリ向けのプログラミングモデルは，OpenMP，Thread Building Blocks（TBB）[63]，Cilk Plus[64] や Chapel[5] など様々ある．本研究では，タスク依存モデルによる PGAS 言語への対応を目的としているため，多くのプログラミングモデルの中でも OpenMP を採用した．

3.8　まとめ

メニーコアプロセッサを持つ大規模な並列システムにおける並列プログラムの性能，生産性の向上を目的として，PGAS モデルによるデータ依存に基づくタスク並列モデルの提案を行った．データ依存によるタスク並列実装をすることで，従来の OpenMP のループ並列によるワークシェアリングが内包する全体同

期を排除しタスク間の一対一同期による高速化や，ノードを跨るタスク間の依存関係を通信で表し，PGAS モデルによる簡易な通信記述でノード内/間のタスク依存を統一的に記述可能とすることによる生産性の向上を目的とした．本研究では，PGAS 言語 XMP を対象としタスク並列を記述可能な `tasklet` 指示文に加え，タスク内での分散配列に対する通信を記述可能な `tasklet gmove`，`tasklet reflect` 指示文と片側通信を実行可能とする `put`，`put_ready`，`get` 及び `get_ready` 節を提案した．また，タスク生成自体の性能向上や一定動作をする `taskyield` 指示文を実現するために軽量スレッドライブラリ Argobots による並列タスク実行の実装や，MPI_THREAD_MULTIPLE によるマルチスレッドでの通信性能低下を改善するための 1 スレッドへの通信委譲による通信最適化を行った．提案指示文によりブロックコレスキー分解とラプラスソルバを実装し，従来のループ並列とタスク並列，OpenMP と Argobots，通信最適化の有無の比較を行った．ブロックコレスキー分解，ラプラスソルバともにループ並列と比較してタスク並列による性能向上を確認した．また，Argobots による実装とすることで，OpenMP では不定だった `taskyield` 指示文によるタスクスイッチングが一定動作をするようになり，通信を実行するタスクが通信完了までスレッドを専有せず，スレッド内においても通信と演算のオーバラップを実行可能とした．通信最適化では，ラプラスソルバは通信回数が少ないためマルチスレッド通信による大きな性能低下が起きず，通信委譲による最適化の効果は得られなかったが，ブロックコレスキー分解においてはタスク並列実装以上の性能向上を確認した．また，生産性の観点からもグローバルビューでは，逐次プログラムのデータコピーのような記述での通信や，分散配列を指定するだけの袖領域通信をタスク上で簡易に記述可能とした．ローカルビューでは，片側通信を実行するための複数回の同期を `tasklet` 指示文の節として記述可能とし，ユーザが全ての通信を明示的に記述する負担を軽減している．結果として，XMP によるタスク並列モデルにより，従来のループ並列と比較して高い性能が得られ，MPI+OpenMP 記述と比較してより簡易な実装を可能とした．

今後の課題として，Omni XMP Compiler の `tasklet` 指示文のトランスレータ部の実装を完了させ，ブロックコレスキー分解やラプラスソルバの XMP 実装の性能評価を行うことや，ハイブリッドビュー同様に実アプリケーションなどの複雑な実装への適用を行うことが挙げられる．また，片側通信によるタスク並列実行の性能向上のために，片側通信実行のための通信回数を削減可能な通信方法の実現が挙げられる．文献 [65] では，片側通信自体に *notify* を付け，片側通信と完了通知を 1 回の通信とすることで通信性能を向上をさせている．この手法をタスク並列における片側通信にも導入することで，通信回数削減による性能向上が期待される．

第 4 章

結論と今後の課題

4.1　結論

本研究では，大規模な分散メモリシステムにおける並列プログラムの性能と生産性の向上を目的とした．まず，生産性の向上を目的として，PGAS モデルを基にしたハイブリッドビューの提案を行った．ハイブリッドビューは，グローバルビューの簡易なデータ分散，並列実行及び通信・同期を記述可能としつつ，ローカルな名前空間でのプログラムを求められるような複雑な通信に対してはローカルビューの簡易な記述による片側通信を記述可能とする．提案モデルにより，従来ではグローバルビューの適用が困難なプログラムに対しても部分的に適用可能となり，プログラム全体としての生産性を向上させることが可能となる．本研究では，PGAS 言語 XMP を対象とし，XMP のグローバルビューとローカルビューを組み合わせたハイブリッドビューを用いて核融合シミュレーションコード GTC-P の実装を行い，オリジナルの MPI 実装と比較をすることで性能と生産性の評価を行った．ハイブリッドビューによる実装では，オリジナルの実装を含むロードインバランスが発生する一部の評価を除き，MPI 実装に近い性能を達成した．生産性の観点からは，グローバルビューによる領域分割により，逐次実装に指示文を追加するのみでの並列化や，隣接格子点間の通信を `reflect` 指示文 1 行で記述することが可能であるため簡易な実装と言える．また，ローカルビューの coarray は，配列代入文形式で通信を記述可能なことから，MPI と比較してより直感的なため可読性が高く，XMP が自動で通信の整合性をとるため通信記述も容易である．以上のことから，PIC 法に含まれる各プロセスの演算量が動的に変化する粒子軌道演算のような，グローバルビューのみで実装することが困難な複雑なアルゴリズムに対しても，XMP のプログラミングモデルを組み合わせることで実装が可能になり，さらに一定の性能を保ちつつ，簡便かつスケーラブルに記述できる事が示された．

ただし，ハイブリッドビューはノード間での通信，データ分割や並列実行を簡易に記述可能としているモデルであり，近年登場しているメニーコアプロセッサを搭載するシステムにおいてはノード間並列に加えて，ノード内並列の性能，生産性も考慮に入れる必要がある．そこで，メニーコアプロセッサを持つ大規模な並列システムにおける並列プログラムの性能，生産性の向上を目的として，PGAS モデルによるデータ依存に基づくタスク並列モデルの提案を行った．データ依存によるタスク並列実装をすることで，従来の OpenMP のループ並列によるワークシェアリングが内包する全体同期を排除し，タスク間の一対一同期による高速化を実現する．また，ノードを跨るタスク間の依存関係を通信で表し，PGAS モデル

による簡易な通信記述でノード内/間のタスク依存を統一的に記述可能とする．本研究では，PGAS 言語 XMP を対象とし `tasklet` 指示文を提案し，提案指示文によるブロックコレスキー分解とラプラスソルバの実装を行い，従来のループ並列による実装との性能，生産性の比較を行った．結果として，両ベンチマークにおいてタスク並列で記述することで，ループ並列による実装よりも高い性能を達成した．また，生産性の観点からも，グローバルビューでは，分散配列に対して配列代入文形式での逐次プログラムのデータコピーのような記述での通信や，分散配列を指定するだけの袖領域通信をタスク上で簡易に記述可能とした．ローカルビューでは，片側通信を実行するための複数回の同期を `tasklet` 指示文の節として記述可能とし，ユーザが全ての通信を明示的に記述する負担を軽減している．プログラム全体として，OpenMP とほぼ同等の記述で分散メモリ環境上でのタスク並列を可能としたことから生産性の高さも示した．性能最適化に関しては，タスク生成の軽量化や一定動作をする `taskyield` 指示文を実現するために軽量スレッドライブラリ Argobots による並列タスク実行の実装や，マルチスレッド環境での通信性能低下を改善するための通信最適化を行った．結果として，通信回数が少ないラプラスソルバにおいては通信最適化の効果は得られなかったが，ブロックコレスキー分解に対しては効果があり性能向上を確認した．

4.2 今後の課題

今後の課題として，ハイブリッドビューモデルによる核融合シミュレーションコード GTC-P の実装では，XMP/C の coarray のランタイムライブラリを別の片側通信ライブラリにより実装し，性能評価を行うことが挙げられる．Omni XMP Compiler では，2018 年 1 月現在 coarray のランタイムライブラリの実装に MPI を用いた実装が追加されており，それを用いた評価や，GASPI や ComEx などの他の PGAS 向けの通信ライブラリを用いて実装することが挙げられる．また，本研究では，分散配列は全て静的に確保された配列を対象としたため，`template_fix` 指示文や *xmp_malloc()* を用いた動的な分散配列を用いた実装を行うことが挙げられる．性能評価で用いたサイズ A は問題スケールとして小規模であったため，ITER のような大規模な核融合装置に対応した問題サイズを解いた場合の評価や，XACC を用いた GPU クラスタ向けの実装を行うことが考えられる．

PGAS モデルによるメニーコアシステム向けタスク並列プログラミングモデルの提案では，Omni XMP Compiler の `tasklet` 指示文のトランスレータ部の実装を完了させ，ブロックコレスキー分解やラプラスソルバの XMP 実装の性能評価を行うことや，ハイブリッドビュー同様に実アプリケーションなどの複雑な実装への適用を行うことが挙げられる．また，片側通信によるタスク並列実行の性能向上のために，片側通信実行のための通信回数を削減可能な通信方法の実現が挙げられる．文献 [65] では，片側通信自体に *notify* を付け，片側通信と完了通知を 1 回の通信とすることで通信性能を向上をさせている．この手法をタスク並列における片側通信にも導入することで，通信回数削減による性能向上が期待される．

また，Top500 を見ると GPU を代表とする演算加速機構を搭載するシステムもメニーコアプロセッサ以上に多く登場している．演算加速機構に対するプログラミングは CUDA や OpenCL，指示文ベースなら OpenACC や OpenMP が代表的な言語やモデルであり，ホストプロセッサから演算をオフロードする形でプログラムを記述する．OpenMP の仕様 4.0 や OmpSs[66] では，`target` 指示文で演算をオフロードすることが可能であり，そのブロック内でループ並列やタスク並列が記述可能である．本研究でも用い

た OpenMP のデータ依存に基づくタスク並列は演算加速機構上で実行可能であり，メニーコアプロセッサと同様に全体同期ではなくタスク間の細粒度な同期による高速化が見込める．演算加速機構を搭載するクラスタは，メニーコアシステム同様の MPI+OpneMP 記述に加えて，`target` 指示文による演算加速機構へのデータ転送まで考慮したプログラミングが必要となり，記述はさらに複雑となる．そこで，本研究で提案した `tasklet` 指示文を演算加速機構向けに拡張し，GPU などの演算加速機構を搭載するクラスタにおいても，データ依存に基づくタスク並列を用いて高速化を行い，演算加速機構を含めてタスク依存で統一的に記述可能とし生産性を向上させることが，今後の課題として挙げられる．

近年では，レイテンシ重視の CPU と，スループット重視の演算加速機構（GPU など）の間を埋める中間のデバイスとして Field Programmable Gate Array（FPGA）が注目を集めている．FPGA による並列クラスタの開発も進められており，筑波大学では Accelerator in Switch（AiS）[67] という，通信機構を含む FPGA 上にアプリケーション特化の演算機構を組み込むコンセプトで次世代並列クラスタの研究開発が進められている．FPGA は Verilog HDL によるプログラミングが一般的だが，より上位の言語より FPGA の回路を生成する高位合成の技術の普及により，近年では OpenCL でのプログラミングが可能となった．しかし，OpenCL での記述も簡易とは言えず，さらに OpenMP や OpenACC の指示文ベースでの記述を可能とする研究も行われている [68]．また，OpenMP のタスクモデルを FPGA に適用する研究 [69] も進められており，FPGA を搭載するような次世代並列クラスタにおける性能や生産性の向上のために，本研究で提案した分散メモリ環境で動作可能な `tasklet` 指示文を適用し，評価を行うことが今後の課題として挙げられる．

謝辞

本論文を執筆するにあたり，5 年間という長い期間，丁寧で熱心なご指導をして頂きました筑波大学システム情報工学研究科教授（連携大学院）・佐藤三久先生に心から感謝を申し上げます．また，お忙しい中，本論文の副査を引き受けて頂きました筑波大学計算科学研究センター教授・朴泰祐先生，筑波大学システム情報工学研究科教授・亀山幸義先生，東北大学サイバーサイエンスセンター教授・滝沢寛之先生，筑波大学システム情報工学研究科准教授・前田敦司先生に深く感謝致します．日頃よりお世話になりました筑波大学計算科学研究センター教授・高橋大介先生，同教授・建部修見先生，同准教授・川島英之先生，同助教・多田野寛人先生，同助教・小林諒平先生，筑波大学システム情報工学研究科准教授・山口佳樹先生，理化学研究所計算科学研究機構・児玉祐悦博士に感謝の意を表します．

本研究の遂行にあたり PGAS 言語 XcalableMP に関して数々の助言を頂きました同機構・村井均博士，同機構・中尾昌広博士，同機構・李珍泌博士，株式会社富士通・岩下英俊氏に御礼申し上げます．核融合シミュレーションコード GTC-P の XcalableMP による実装では，コード提供や実装に関する数々のご助力を頂きました Princeton 大学教授・William Tang 先生，同大学・Bei Wang 博士，原子力研究開発機構・奴賀秀男博士に感謝を申し上げます．PGAS モデルにおけるタスク並列プログラミングに関する研究では，Houston 大学へのインターンシップを受け入れてくださり，私の拙い英語にも関わらず辛抱強く議論して頂きました，Stoney Brook 大学教授・Barbara Chapman 先生，Intel Corp. Dounia Khaldi 博士，Cray, Inc. Deepak Eachempati 博士に深く感謝をするとともに，軽量スレッドライブラリ Argobots に関する多くの助言を頂きました理化学研究所計算科学研究機構・李珍泌博士，株式会社ソニー・インタラクティブエンタテインメント・杉山大輔氏に感謝致します．本研究の一部は 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」と理化学研究所計算科学研究機構と筑波大学計算科学研究センターの共同研究「ポスト京の並列プログラミング環境およびネットワークに関する研究」によるものです．

海外出張や研究生活における様々な面において大変お世話になりました先輩である筑波大学計算科学研究センター・藤田典久博士，理化学研究所計算科学研究機構・小田嶋哲哉博士，株式会社富士通研究所・大辻弘貴博士，ヤフー株式会社・鷹津冬将博士に御礼申し上げます．日頃の研究や生活面で大変お世話になりました PA チーム・田渕晶大氏を始め，後輩である株式会社日立製作所・宇川斉志氏，日本電気株式会社・大川千聡氏，HPCS 研究室の皆様にこの場を借りて感謝を述べさせていただきます．最後に大学生活 9 年間を支えてくださいました両親に深く御礼申し上げます．

参考文献

[1] “Top500 Supercomputer Sites”, https://www.top500.org/

[2] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, W. De Meuter, “Partitioned Global Address Space Languages”, ACM Computing Surveys (CSUR), Vol. 47 No. 4, pp. 62:1–62:27 (2015).

[3] UPC Consortium, “UPC Language Specifications Version 1.3”, https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf (2013).

[4] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, K. Yelick, “UPC++: A PGAS Extension for C++”, 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 1105–1114, Phoenix, AZ, USA (2014).

[5] B. L. Chamberlain, D. Callahan, H. P. Zima, “Parallel Programmability and the Chapel Language”, International Journal of High Performance Computing Applications, Vol. 21, No. 3, pp. 291–312 (2007).

[6] J. Nieplocha, R. J. Harrison, R. J. Littlefield, “Global Arrays: a portable “shared-memory” programming model for distributed memory computers”, Proceedings of Supercomputing ’94, pp. 340–349, Washington, DC, USA (1994).

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, “X10: An Object-oriented Approach to Non-uniform Cluster Computing”, Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA ’05), pp. 519–538, San Diego, CA, USA (2005).

[8] C. H. Koelbel, M. E. Zosel. “The High Performance FORTRAN Handbook”, MIT Press, Cambridge, MA, USA (1993).

[9] XcalableMP Specification Working Group, “XcalableMP Website”, http://www.xcalablemp.org/

[10] J. Lee, M. Sato, “Implementation and Performance Evaluation of XcalableMP: a Parallel Programming Language for Distributed Memory Systems”, 2010 39th International Conference on Parallel Processing Workshops, pp. 413–420, San Diego, CA, USA (2010).

[11] M. Nakao, J. Lee, T. Boku, M. Sato, “Productivity and Performance of Global-view Programming with XcalableMP PGAS Language”, 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2012), pp. 402–409, Ottawa, ON, Canada (2012).

[12] J. Reid, “Coarrays in the next Fortran Standard”, ISO/IEC JTC1/SC22/WG5 N1824 (2010).

[13] B. Chapman, T. Curtis, S. Pophale, C. Koelbel, J. Kuehn, S. Poole, L. Smith, “Introducing OpenSHMEM, SHMEM for the PGAS Community”, Proceedings of the Fourth Conference on Partitioned Global

Address Space Programming Model (PGAS '10), pp. 2:1–2:3, New York, USA (2010).

[14] S. Sumimoto, Y. Ajima, K. Saga, T. Nose, N. Shida, T. Nanri, "The Design of Advanced Communication to Reduce Memory Usage for Exa-scale Systems", VECPAR 2016: 12th International Conference, pp. 149–161, Porto, Portugal (2016).

[15] 土井淳, "XcalableMP による格子 QCD の並列化と Blue Gene/Q における性能評価", 情報処理学会研究報告, Vol. 2014-HPC-148, No. 28, pp. 1–8 (2014).

[16] M. Nakao, H. Murai, H. Iwashita, A. Tabuchi, T. Boku, M. Sato, "Implementing Lattice QCD Application with XcalableACC Language on Accelerated Cluster", 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 429–438, Honolulu, HI, USA (2017).

[17] M. Frumkin, H. Jin, J. Yan, "Implementation of NAS Parallel Benchmarks in High Performance Fortran", NAS Technical Report NAS-98-009, pp. 1–25 (1998).

[18] H. Shan, S. Williams, Y. Zheng, A. Kamil, K. Yelick, "Implementing High-Performance Geometric Multigrid Solver with Naturally Grained Messages", 2015 9th International Conference on Partitioned Global Address Space Programming Models, pp. 38-46, Washington, DC, USA (2015).

[19] H. Shan, S. Williams, Y. Zheng, W. Zhang, B. Wang, S. Ethier, Z. Zhao, "Experiences of Applying One-Sided Communication to Nearest-Neighbor Communication", 2016 PGAS Applications Workshop (PAW), pp. 17–24, Salt Lake City, UT, USA (2016).

[20] A. Tabuchi, M. Nakao, H. Murai, T. Boku, M. Sato."Implementation and Evaluation of One-sided PGAS Communication in XcalableACC for Accelerated Clusters", 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), pp. 625–634, Madrid, Spain (2017).

[21] M. Pritchard, H. P. Jr, B. Zoran, S. Vivek, "Graph 500 on OpenSHMEM: Using a Practical Survey of Past Work to Motivate Novel Algorithmic Developments", Los Alamos National Laboratory Technical Report, LA-UR–16-29614, pp. 1–16 (2016).

[22] D. Eachempati, A. Richardson, T. Liao, H. Calandra, B. Chapman, "A Coarray Fortran Implementation to Support Data-Intensive Application Development", 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 771–776, Salt Lake City, Utah, USA (2012).

[23] 塙 敏博, 星野 哲也, 中島 研吾, 大島 聡史, 伊田 明弘, "Xeon Phi+OmniPath 環境における OpenMP, MPI 性能最適化", 情報処理学会研究報告, Vol. 2017-HPC-158, No. 21, pp. 1–8 (2017).

[24] A Sodani, "Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor", IEEE Hot Chips 27 Symposium, https://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.70-Processors-Epub/HC27.25.710-Knights-Landing-Sodani-Intel.pdf (2015).

[25] EPCC, "EPCC OpenMP micro-benchmark suite", https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite

[26] M. Si, P. Balaji, "Process-based Asynchronous Progress Model for MPI Point-To-Point Communication", 2017 19th IEEE International Conference on High Performance Computing and Communications, pp. 1–7, Bangkok, Thailand (2017).

[27] HV. Dang, S. Seo, A. Amer, P. Balaji, "Advanced Thread Synchronization for Multithreaded MPI Implementations", 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing,

pp. 314–324, Madrid, Spain (2017).

[28] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, B. Joó, “Improving Concurrency and Asynchrony in Multithreaded MPI Applications Using Software Offloading”, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC’15), pp. 30:1–30:12, Austin, Texas, USA (2015)

[29] PC Cluster Consortium, “PC Cluster Consortium”, https://www.pccluster.org/en/

[30] RIKEN AICS, University of Tsukuba, “Omni Compiler Project”, http://omni-compiler.org/

[31] S. Ethier, M. Adams, J. Carter, L. Oliker, “Petascale Parallelization of the Gyrokinetic Toroidal Code”, Proceedings of 9th International Conference of High Performance Computing for Computational Science, Berkeley, CA, USA, pp. 1–9 (2010).

[32] B. Wang, S. Ethier, W. Tang, K. Z. Ibrahim, K. Madduri, S. Williams, L. Oliker, “Modern Gyrokinetic Particle-In-Cell Simulation of Fusion Plasmas on Top Supercomputers”, ArXiv e-prints, pp. 1–20 (2015).

[33] 内藤 裕志, 佐竹 真介, “5. 粒子シミュレーションのコーディング技法 (核融合プラズマシミュレーションの技法 -大規模並列計算環境の活用-)”, プラズマ・核融合学会誌, Vol. 89, No. 4, pp. 245–260 (2013).

[34] H. Nuga, “Kinetic Modeling of the Heating Processes in Tokamak Plasmas”, PhD Thesis, Kyoto University, pp. 1–109 (2011).

[35] S. Ethier, W. M. Tang, Z. Lin, “Gyrokinetic Particle-in-cell Simulations of Plasma Microturbulence on Advanced Computing Platforms”, Journal of Physics:Conference Series, Vol. 16, No. 1, pp. 1–15 (2005).

[36] DoE SCiDAC, UC Irvine, etc., “GTC: Gyrokinetic Toroidal Code”, http://phoenix.ps.uci.edu—/GTC/

[37] Y. Shimomura, R. Aymar, V. Chuyanov, M. Huguet, R. Parker, ITER Joint Central Team, “ITER Overview”, Nuclear Fusion, Vol. 39, No. 9Y, p. 1295 (1999).

[38] The Ohio State University, “OSU Micro-Benchmarks”, http://mvapich.cse.ohio-state.edu/benchmarks/

[39] Center for Computational Sciences, University of Tsukuba, “HA-PACS/TCA”, https://www.ccs.tsukuba.ac.jp/eng/supercomputers/#HA-PACS

[40] LBNL FTG, U.C. Berkeley, “GASNet Specification Version 1.8.1”, http://gasnet.lbl.gov/dist/docs/gasnet.pdf

[41] W. Huang , G. Santhanaraman , H. Jin , Q. Gao , D. K. Panda, “Design and Implementation of High Performance MVAPICH2: MPI2 over InfiniBand”, 2016 Sixth IEEE International Symposium on Cluster Computing and the Grid, pp. 43–48, Singapore (2006).

[42] A. Stone, J. Dennis, M. Strout, “Evaluating Coarray Fortran with the CGPOP Miniapp”, Proceedings of 5th Conference on Partitioned Global Address Space Programming Models, pp. 1–10, Galveston Island, TX, USA (2011).

[43] R. Preissl, N. Wichmann, B. Long, J. Shalf, S. Ethier, A. Koniges, “Multithreaded Global Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms”, 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, pp. 78:1–78:11, Seatle, WA, USA (2011).

[44] H. Sakagami, T. Mizuno, “Compatibility comparison and performance evaluation for Japanese HPF compilers using scientific applications”, Concurrency Computation Practice and Experience. Vol. 14, pp.

679–689 (2002).

[45] 下坂 健則, 佐藤 三久, 朴 泰祐, William Tang, “京速コンピュータ「京」における核融合シミュレーションコード GTC-P の評価”, 情報処理学会研究報告, Vol. 2013-HPC-139, No. 2, pp. 1–6 (2013).

[46] X. Liao, L. Xiao, C. Yang, Y. Lu, “MilkyWay-2 supercomputer: system and application”, Frontiers of Computer Science. Vol. 8, No. 3, pp. 345–356 (2014).

[47] K. Madduri, K. Z. Ibrahim, S. Williams, E. J. Im, S. Ethier, J. Shalf, L. Oliker, “Gyrokinetic Toroidal Simulations on Leading Multi- and Manycore HPC Systems”, 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 23:1–23:12, Seatle, WA, USA (2011).

[48] M. Nakao, H. Murai, T. Shimosaka, A. Tabuchi, T. Hanawa, Y. Kodama, T. Boku, M. Sato, “XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters”, 2014 First Workshop on Accelerator Programming using Directives, pp. 27–36, New Orleans, LA, USA (2014).

[49] Fraunhofer ITWM, “Gaspi: Global Address Space Programming Interface, Specification of a PGAS API for communication”, http://www.gaspi.de/

[50] J. Daily, A. Vishnu, B. Palmer, H. van Dam, D. Kerbyson“On the suitability of MPI as a PGAS runtime”, 2014 21st International Conference on High Performance Computing, pp. 1–10, Dona Paula, India (2014).

[51] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, A. Castell, D. Genet, T. Herault, P. Jindal, L. V. Kal, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, P. Beckman, “Argobots: A Lightweight, Low-Level Threading and Tasking Frame- work”, ANL/MCS-P5515-0116, pp. 1–12 (2016).

[52] A. Fernandez, V. Beltran, X. Martorell, R. M. Badia, E. Ayguade, J. Labarta, “Task-Based Programming with OmpSs and Its Application”, Euro-Par 2014: Parallel Processing Workshops, pp. 25–26, Porto, Portugal (2014).

[53] Argonne National Laboratory, “Argo: An exascale operating system”, http://www.mcs.anl.gov/project/argo-exascale-operating-system

[54] JCAHPC（Joint Center for Advanced HPC：最先端共同 HPC 基盤施設）, http://jcahpc.jp/

[55] Center for Computational Sciences, University of Tsukuba, “COMA (PACS-IX)”, https://www.ccs.tsukuba.ac.jp/eng/supercomputers/#COMA

[56] T. Boku, “Interconnection and I/O System on Oakforest-PACS: World Largest KNL+OPA Cluster”, Third International Workshop on Communication Architectures for HPC, Big Data, Deep Learning and Clouds at Extreme Scale, Frankfurt, Germany, http://nowlab.cse.ohio-state.edu/static/media/workshops/presentations/exacomm17/exacomm17-invited-talk-taisuke-boku.pdf (2017)

[57] A. Cedric, T. Samuel, N. Raymond, W. Pierre-Andre, “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures”, Concurrency and Computation: Practice and Experience, Vol. 23, No. 2, pp. 187–198 (2011).

[58] A. YarKhan, “Dynamic Task Execution on Shared and Distributed Memory Architectures”, PhD Dissertation, Major Advisor: J. Dongarra, University of Tennessee, pp. 1–120 (2012).

[59] A. YarKhan, J. Kurzak, J. Dongarra, “QUARK Users’ Guide: QUeueing And Runtime for Kernels”,

University of Tennessee Innovative Computing Laboratory Technical Report, ICL-UT-11-02, pp. 1–18 (2011).

[60] M. Garland, M. Kudlur, Y. Zheng, “Designing a Unified Programming Model for Heterogeneous Machines”, International Conference on High Performance Computing, Networking, Storage and Analysis, SC ’ 12, pp. 67:1–67:11, Salt Lake City, Utah, USA (2012).

[61] D. Alejandro, A. Eduard, B. Rosa M, L. Jesus, M. Luis, M. Xavier, P. Judit, “OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures”, Parallel Processing Letters, Vol. 21, pp. 173–193 (2011).

[62] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, J. Labarta, ”Productive Cluster Programming with OmpSs”, Euro-Par 2011 Parallel Processing, Bordeaux, France, pp. 555–566 (2011).

[63] “Intel Threading Building Blocks”, https://www.threadingbuildingblocks.org/

[64] “Intel CilkPlus”, https://www.cilkplus.org/

[65] R. Belli, T. Hoefler, “Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization”, 2015 IEEE International Parallel and Distributed Processing Symposium, pp. 871–881, Hyderabad, India (2015).

[66] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, J. Labarta, “Productive Programming of GPU Clusters with OmpSs”, 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pp. 557–568, Shanghai, China (2012).

[67] T. Boku, “Next Generation Interconnection For Accelerated Computing”, http://nowlab.cse.ohio-state.edu/static/media/workshops/presentations/ExaComm16-Invited-Talk-7-Taisuke-Boku.pdf (2016).

[68] S. Lee, J. Kim, J. S. Vetter, “OpenACC to FPGA: A Framework for Directive-Based High-Performance Reconfigurable Computing”, 2016 IEEE International Parallel and Distributed Processing Symposium, pp. 544–554, Chicago, IL, USA (2016)

[69] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, D. Jimenez-Gonzalez, “OpenMP extensions for FPGA accelerators”, 2009 International Symposium on Systems, Architectures, Modeling, and Simulation, pp. 17–24, Samos, Greece (2009)

付録 A

公表論文リスト

論文誌

1. Keisuke Tsugane, Taisuke Boku, Hitoshi Murai, Mitsuhisa Sato, William Tang, Bei Wang, ”Hybrid-view Programming of Nuclear Fusion Simulation Code in the PGAS Parallel Programming Language XcalableMP”, Parallel Computing, Vol. 57, pp. 37–51, Sep. 2016.

査読付き国際会議論文

1. Keisuke Tsugane, Hideo Nuga, Taisuke Boku, Hitoshi Murai, Mitsuhisa Sato, William Tang, Bei Wang, ”Hybrid-view Programming of Nuclear Fusion Simulation Code in the PGAS Parallel Programming Language XcalableMP”, The 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2014), pp. 640–647, Hsinchu, Taiwan, Dec. 2014.
2. Keisuke Tsugane, Jinpil Lee, Hitoshi Murai, and Mitsuhisa Sato. “Multi- tasking Execution in PGAS Language XcalableMP and Communication Optimization on Many-core Clusters. In HPC Asia 2018: International Conference on High Performance Computing in Asia-Pacific Region, pp. 75–85, Tokyo, Japan, Jan. 2018.

査読付き国内会議論文

1. 津金佳祐, 朴泰祐, 村井均, 佐藤三久, William Tang, Bei Wang, ”PGAS 言語 XcalableMP のハイブリッドビューによる核融合シミュレーションコードの実装と評価”, 2015 年ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集, pp. 54–63, 東京, 2015 年 5 月

研究会発表

1. 津金佳祐, 奴賀秀男, 朴泰祐, 村井均, 佐藤三久, William Tang, ”並列言語 XcalableMP による核融合シミュレーションコードの実装と評価”, 情報処理学会研究報告, Vol. 2014-HPC-145, No. 37, pp. 1–7, 新潟, 2014 年 7 月
2. 津金佳祐, 中尾昌広, 李珍泌, 村井均, 佐藤三久, ”PGAS 言語 XcalableMP における動的タスク機能の提案”, 情報処理学会研究報告, Vol. 2015-HPC-151, No. 5, pp. 1–7, 沖縄, 2015 年 10 月
3. 津金佳祐, 中尾昌広, 李珍泌, 村井均, 佐藤三久, ”軽量スレッドライブラリ Argobots による PGAS 言語 XcalableMP の動的タスク並列機能の設計”, 情報処理学会研究報告, Vol. 2016-HPC-155, No. 29, pp. 1–8, 長野, 2016 年 8 月
4. 津金佳祐, 田渕晶大, 李珍泌, 村井均, 朴泰祐, 佐藤三久, ”KNL メニーコア・プロセッサにおける PGAS 言語 XcalableMP アプリケーションの性能評価”, 情報処理学会研究報告, Vol. 2017-HPC-158, No. 17, pp. 1–9, 熱海, 静岡, 2017 年 3 月

ポスター発表

1. Keisuke Tsugane, Masahiro Nakao, Jinpil Lee, Hitoshi Murai, Mitsuhisa Sato, ”Proposal for Dynamic Task Parallelism in PGAS Language XcalableMP”, The 6th AICS International Symposium, p. 57, Kobe, Hyogo, Japan, Feb. 2016.