高並列言語による演算加速器及び
相互結合網の効率的利用に関する研究

２０１６年　３月

小田嶋　哲哉

高並列言語による演算加速器及び
相互結合網の効率的利用に関する研究

小田嶋　哲哉

システム情報工学研究科

筑波大学

２０１６年　３月

概要

GPU を代表とするアクセラレータ（演算加速器）は，その高い演算性能やメモリバンド幅，電力あたりの性能が CPU と比較して高く，これらを搭載したクラスタシステムが HPC の様々な分野で広く利用されている．一般的に，アクセラレータは，CPU のメモリとは異なるアドレス空間のメモリを持っているため，その間のデータ転送をユーザが明示的に行う必要がある．特に，アクセラレータを搭載したクラスタ環境においては，従来の分散メモリ環境向けのプログラミングに加えて，アクセラレータを制御するためのプログラムも記述する必要があり，プログラミングのコストが高い．このような環境において，アクセラレータと並行して汎用プロセッサを演算に利用することや，異なる性質のネットワークを用いてアクセラレータ間通信を実現には，より複雑なプログラミングが必要になり，アプリケーションの生産性が低下してしまうことが問題である．そこで，本論文では GPU クラスタを対象に，大規模分散メモリ環境における次世代のプログラミング言語として，PGAS 並列プログラミング言語 XcalableMP に着目し，そのアクセラレータ搭載の並列計算機向けの拡張言語 XcalableMP acceleration device extension による GPU と CPU によるワークシェアリングならびに，XcalableACC による TCA/PEACH2 と InfiniBand によるハイブリッド通信を実装し，これらを容易に実現するためのフレームワークを提供することを目的とする．アプリケーションの実行中に CPU と GPU に割り当てるデータサイズの最適化を適用的に行う機構を導入することで，GPU のみを演算に用いる場合と比較して高い演算性能を達成した．また，コモディティネットワークである InfiniBand と GPU 間直接通信機構である TCA/PEACH2 によるハイブリッド通信の最適化により，データ分割に伴う袖領域間通信や集団通信において，InfiniBand だけでは得られない性能を達成した．さらに，高並列言語中にこれらのシステムを適用することで，Hand-coding したプログラムに対して，高い生産性があることを示した．

謝辞

本論文を執筆するにあたり，時には優しく，時には厳しい指導を賜り，博士論文という形にすることができたのは筑波大学システム情報工学研究科 朴泰祐教授のおかげです．ここに深く感謝の意を表します．学群 4 年から現在に至る 6 年もの間，充実した研究環境を提供してくださり，多くの国際学会や研究会において発表することができました．これらの経験は，将来の糧になるものであります．ここに重ねて御礼申し上げます．

また，お忙しい中，本論文の副査を引き受けていただいたきました筑波大学システム情報工学研究科 和田耕一教授，筑波大学システム情報工学研究科 高橋大介教授，筑波大学システム情報工学研究科 建部修見教授，理化学研究所計算科学研究機構 村井均博士に深く感謝いたします．

本研究課程におきまして，XcalableMP，XcalableMP acceleration device extension および XcalableACC に関する助言を頂きました，筑波大学システム情報工学研究科 佐藤三久教授，理化学研究所計算科学研究機構 村井均博士，理化学研究所計算科学研究機構 中尾昌広博士，理化学研究所計算科学研究機構 李珍泌博士，筑波大学 HPCS 研究室 田渕晶大氏，津金佳祐氏に御礼申し上げます．

TCA アーキテクチャおよび PEACH2 のシステム開発におきまして，システムの構築や API ライブラリの設計・実装を通じて本研究にご協力いただきました，東京大学情報基盤センター 塙敏博准教授，筑波大学計算科学研究センター 松本和也博士，筑波大学 HPCS 研究室 藤田典久氏，藤井久史氏に御礼申し上げます．

また，インターンシップを通じて StarPU に関する助言や研究に協力いただきました，University of Bordeaux Raymond Namyst 教授，University of Bordeaux Samuel Thibault 准教授，INRIA Bordeaux Olivier Aumage 博士に感謝いたします．

最後に，本研究や日常の生活において多くのご指導，ご助力をいただきました HPCS 研究室の皆様に深く感謝いたします．

目次

図目次

表目次

Listings

第 1 章

序論

計算科学の分野において，スーパーコンピュータは，地震シミュレーションや気象予報などの科学技術計算を高速に演算するためには必要不可欠であり，複数の計算ノードを高速なネットワークで相互結合したクラスタシステムが主流となっている．搭載する CPU のアーキテクチャの改良やコア数の増加に加え，接続する計算ノードの数を増やすことで，より高い演算性能を実現することができる一方で，消費される電力もまた大幅に増加してしまうため，単純な大規模化は困難である．

そこで，近年，NVIDIA 社の GPU（Graphics Processing Unit） [1] や Intel 社の Xeon Phi [2] などのアクセラレータ（演算加速器）が持つ，高い演算性能やメモリバンド幅，電力あたりの性能に注目し，これらを搭載したクラスタシステムが HPC（High Performance Computing）の分野で広く利用されている．スーパーコンピュータの性能を競うランキングである TOP500 [3] の結果からも，数多くのアクセラレータを搭載したクラスタシステムが大規模スーパーコンピュータとして実用化されていることがわかる．これらのシステムは，単に演算性能が高いだけでなく，電力あたりの演算性能も非常に高いことが特徴として挙げられる．TOP500 と同様に HPL（High Performance Linpack）ベンチマークを用いて，電力あたりの性能を競うランキングである Green500 [4] において，2015 年 11 月の Top10 のシステム全てがアクセラレータを搭載している．このように，今後の大規模なシステムを構築するに当たって，アクセラレータは重要な要素技術であると言える．

GPU などのアクセラレータを搭載したクラスタシステムは，アクセラレータが異なるアドレス空間を持つヘテロジニアスなノード構成である．そのため，従来の，CPU のみで構成されるシステムにおける分散メモリ環境向けのプログラミングに加えて，アクセラレータを管理するためのプログラミングモデルも併用して記述しなければならず，プログラミングのコストが大きい．さらに，システム中の演算器や相互結合網を最大限利用するために，アクセラレータと並行して CPU を演算に利用したり，異なる性質を持つネットワークを用いてアクセラレータ間の通信を実行したりするためには，より複雑なプログラミングが求められ，アプリケーションの生産性が大きく低下することが問題となっている．しかしながら，今後大規模なクラスタシステムを構築するにはアクセラレータを搭載することが必要不可欠であり，アプリケーションの開発において，生産性を向上させることが大きな課題である．そこで本研究は，大規模分散メモリ環境における次世代のプログラミング言語として理化学研究所計算科学研究機構が中心となって開発を進めている，PGAS（Partitioned Global Address Space）並列プログラミング言語 XcalableMP に着目し，アプリケーションの生産性とシステムリソースの効率的利用を両立するためのプログラミングフ

レームワークを提供することを目的とする．

本論文では，特に，アクセラレータとして代表的な GPU を搭載した GPU クラスタのヘテロジニアス性に着目し，GPU と CPU によるハイブリッドワークシェアリングならびに GPU 間直接通信機構 TCA/PEACH2 とコモディティネットワークである InfiniBand によるハイブリッド通信を提案する．これらのシステムを高並列言語である XcalableMP の拡張言語に適用することで，高い生産性とシステムリソースの効率的な利用を両立するプログラミング環境を提供する．

計算の効率的な利用に関して，本論文では，アクセラレータ搭載のクラスタシステム向けの拡張言語である XcalableMP acceleration device extension（XMP-dev）に基づく，GPU と CPU ハイブリッドワークシェアリングを実現するシステムとして，XMP-dev/StarPU を提案する．XMP-dev は従来，アクセラレータのみをデータ分散や演算の対象としていたが，このバックエンドスケジューラとして INRIA Bordeaux で開発が進められている StarPU ランタイムシステムを適用することで，マルチノード環境における GPU と CPU によるワークシェアリングが可能になる．GPU/CPU ハイブリッドワークシェアリングにおいて，高い演算性能を発揮するためには，GPU と CPU へ割り当てるデータの割合が重要である．従来，この割合を決定するにはパラメータサーチが必要であったが，対象とする問題の性質やデータサイズによって値が異なり，静的な決定が困難であった．XMP-dev/StarPU では，この割合をプログラムの実行中に変更することが可能な適応型負荷分散機能を導入しているため，従来の静的な解析よりも低いコストで値を決定することができ，生産性が向上している．

GPU 間通信の効率的な利用に関して，本論文では，XcalableMP のアクセラレータ向けの拡張言語である XcalableACC に基づき，筑波大学計算科学研究センターが中心となって開発が進められている GPU 間直接通信機構 TCA/PEACH2 とコモディティネットワークである InfiniBand による TCA/InfiniBand ハイブリッド通信を提案する．クラスタシステム内のすべてのノードが，InfiniBand によってフラットに接続されていることは自然であり，このネットワーク中のローカルな通信を加速するネットワークとして TCA/PEACH2 を適用することでハイブリッド通信を実現する．これは，単純に 2 つの通信路を束ねて全体のバンド幅を向上させるだけではなく，それぞれのネットワークの特徴に応じた通信を XcalableACC のランタイムシステム内で選択することで，InfiniBand ネットワークのみでは得られない通信性能の向上を可能にする．さらに，XcalableACC のフレームワーク中にハイブリッド通信を適用することで，アプリケーションの生産性の向上と，異なる相互結合網の効率的利用を実現している．

本論文では，様々なベンチマークを通じて，これら提案したフレームワークが GPU クラスタにおける生産性と，演算や GPU 間通信の性能向上を実現しているか評価を行う．

本論文の構成を以下に示す．

第 2 章では，本論文の研究背景と目的を示す．本論文では特に，GPU クラスタが持つヘテロジニアス性に起因する，演算および GPU 間通信についての問題を明らかにし，本研究の目的を示す．

第 3 章では，本論文の先行研究である並列言語 XcalableMP ならびに，そのアクセラレータ搭載の並列計算機向けの拡張言語仕様である XcalableMP acceleration device extension および XcalableACC について，本論文を理解するために必要な最小限の説明を示す．

第 4 章では，GPU クラスタにおける，GPU と CPU によるハイブリッドワークシェアリングを実現するためのフレームワークとして，XMP-dev/StarPU を提案する．さらに，XMP-dev のフレームワーク上で，StarPU による GPU と CPU のワークシェアリングおよび負荷のバランスを調整する適応型負荷

分散機能を導入する．これによって，XMP-dev/StarPU のフレームワーク上で，GPU/CPU のハイブリッドワークシェアリングの生産性，GPU クラスタの計算リソースを最大限利用および，適応型負荷分散機能の有効性を検証する．

第 5 章では，筑波大学計算科学研究センターが中心となって開発している，PEACH2（PCI Express Adaptive Communication Hub version 2）に基づく TCA（Tightly Coupled Accelerators）アーキテクチャの概要とその基礎通信性能を示す．

第 6 章では，GPU クラスタにおける，コモディティネットワークである InfiniBand と GPU 間直接通信機構 TCA/PEACH2 によるハイブリッド通信を提案し，XACC の通信ランタイムに適用した．これによって，XACC のフレームワーク上で，低いプログラミングコストで複数の通信ネットワークを有効に利用できるか検証を行う．

最後に，第 7 章で本論文のまとめを述べる．

第 2 章

研究の背景と目的

本章では，本研究の研究背景を明らかにし，関連研究を交えて研究の位置づけと本研究の目的を示す．

2.1 研究の背景

2.1.1 GPU クラスタにおけるプログラミング環境

従来，GPU は画像処理に特化した専用のハードウェアを搭載していたが，より柔軟なプログラミングが可能なプログラマブルシェーダーへの移行が進んだことで，これを画像処理以外の汎用計算に用いる GPGPU（General-Purpose computing on GPU）が注目されるようになった．GPGPU が広く利用されるようになった要因として次の 2 つが挙げられる．

一つは，高い浮動小数点演算性能である．GPU は，数百から数千コアが同時に演算を行うことが可能であり，高い並列性やメモリバンド幅を持つ．最新の GPU（NVIDIA 社 Tesla K80）は，倍精度浮動小数点演算性能が 1.87TFLOPS に達する．これによって，従来のアプリケーションが数倍から数十倍に速度向上を達成することが可能になる．

もう一つは，プログラミング環境の改良である．GPGPU が普及する以前は，GPU のシステムに近いレイヤのプログラミングが必要であり，プログラミングのコストが非常に大きかった．NVIDIA 社の CUDA（Compute Unified Device Architecture）[5] や CHRONOS の OpenCL [6] などのアクセラレータ向けの統合開発環境が提供されたことで，C 言語のような従来のプログラミングに近い形での開発が可能になった．これによって，シングルノードにおけるプログラミングが容易となったため，GPGPU によるアプリケーションの開発が加速した．しかしながら，GPU クラスタなどのマルチノード環境におけるプログラミングコストは，依然として高いままである．この原因として，従来のクラスタシステムで用いられている MPI（Message Passing Interface）と OpenMP に加えて，CUDA や OpenCL による GPU を管理するためのプログラムを記述しなければならないことが挙げられる．図 2.1 に，従来のクラスタシステムと一般的な GPU クラスタにおけるプログラミングの流れを示す．図 2.1 より，青色の枠は従来のクラスタシステムで行われている操作であり，赤色の枠は GPU クラスタで必要とされている，GPU を管理するための操作である．図の上半分が従来のクラスタシステム，下半分が GPU クラスタのプログラミングの流れである．GPU に限らず，一般的にアクセラレータは，CPU 側のメインメモリ（ホストメモ

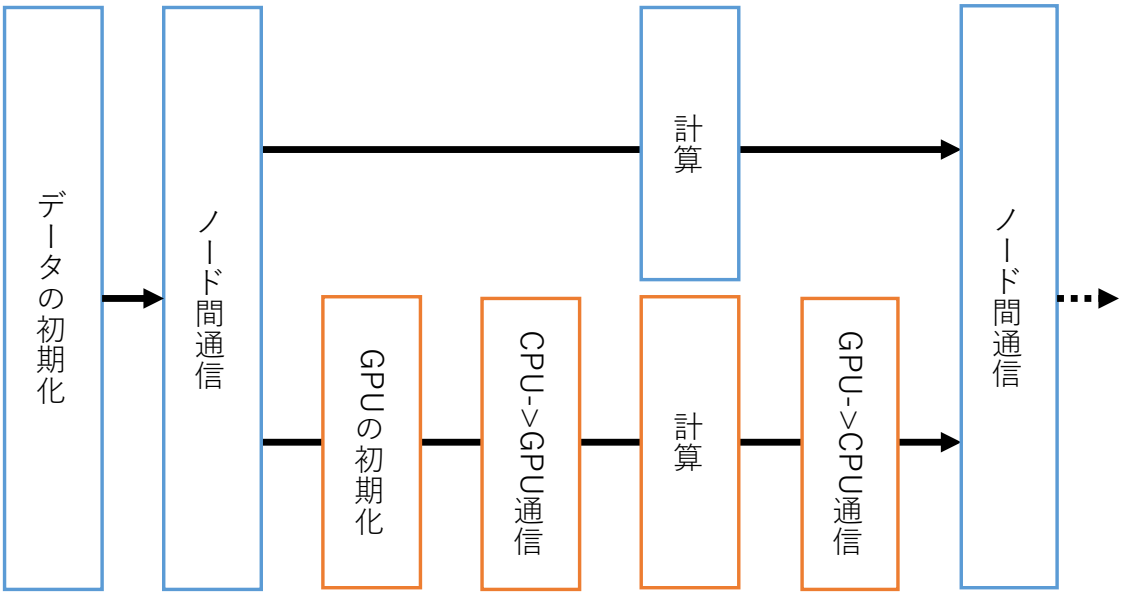


図 2.1　並列 GPU プログラミングの流れ

リ）と異なるアドレス空間である独自のメモリ（デバイスメモリ）を持っている．このように，GPU クラスタは，ノード間のホストメモリ間の分散メモリ環境の他に，ノード内のホストメモリとデバイスメモリという新たな分散メモリ環境が存在している．そのため，GPU で演算を行うには，計算に必要なデータをホスト〜デバイスメモリ間で明示的に通信を行う必要がある．

さらに，GPU クラスタには，演算性能と通信性能に関して GPU クラスタの持つヘテロジニアス性に起因した問題が存在している．

2.1.2　演算に関する問題

GPU クラスタでは，GPU を一種の非常に高速な計算加速器と見なして，CPU から計算するデータをオフロードし，計算が終わったらデータを受け取るという機能分散的なプログラミングが一般的に用いられる．しかし，これでは CPU は，GPU の管理のみを行うだけで GPU と並行して計算リソースとして利用することができず，システムの計算リソースを最大限に利用することができない．今日，CPU の演算性能の向上はめざましいものがある．例えば， Haswell アーキテクチャの Intel Xeon E5 v3 は最大で 18 コアを搭載しており，以前の数倍ものコアを 1 つのソケット内に実装している．さらに，Intel の AVX2 命令を代表とする，SIMD（Single Instruction Multiple Data）命令が強化されたことによって，単一コアの性能も向上している．このことからも，CPU の演算性能を無視することはできない．

そこで，GPU と CPU によるワークシェアリングを考える．大島らの研究 [7] では，GPU を用いる演算において，CPU の負荷が低いことに注目し，GPU と CPU を並行利用することで GPU のみを計算に利用した時と比較して HPL において 1.45 倍の性能向上を達成している．遠藤らの研究 [8, 9] では，CPU とアクセラレータである GPU および ClearSpeed を並行利用することで，HPL において CPU のみの演算に対して 2.28 倍の性能向上を達成している．一方で，GPU と CPU によるワークシェアリングにおいて，計算リソースの性能を最大限利用するためには，リソースに応じた最適なデータサイズを割り

当てる必要があるが，問題の性質や問題サイズ，実行環境によってこの割合は大きく異なるため，固定的な配分比では十分に最適化を行うことが非常に困難である．

2.1.3　通信に関する問題

GPU の持つ高いメモリバンド幅に対して，GPU を接続する PCI Express（PCIe）の性能は非常に低い．そのため，従来，ノードを跨ぐ GPU 間のデータ交換には，一度ホストメモリを経由して通信を行う必要があるため，通信レイテンシが非常に大きかった．近年，NVIDIA 社の Kepler アーキテクチャより最新の GPU では，GPUDirect Support for RDMA（以降「GDR」と略す） [10] を用いた，GPU 間でホストメモリを経由しない，直接データ通信が可能な技術が提供されている．コモディティネットワークである InfiniBand [11] と GDR を利用した MPI の実装に，オハイオ州立大学の MVAPICH2-GDR [12]（以降「MV2GDR」と略す）がある．MV2GDR は，ホストメモリを経由した通信よりも低いレイテンシを実現することができるが，依然として PCIe と InfiniBand 間のプロトコルの切り替えオーバヘッドが存在しているため，特にデータサイズが小さい場合において通信ボトルネックとなる．文献 [13] にもあるように，今後はアプリケーションの強スケーリング性能を向上することが重要となるため，アプリケーションの大きな性能ボトルネックとなりかねない．

この問題を解決するために，GPU 間を直接結合し，通信レイテンシの改善を図るための機構として，筑波大学の密結合並列演算加速機構 TCA（Tightly Coupled Accelerators）アーキテクチャ [14, 15, 16, 17] や NVIDIA 社の NVLink [18] などが開発されている．NVLink は，ノード内の複数の GPU を高速なバスで直接接続する．現在開発されている，NVLink 1.0 は PCIe Gen3 に基づいているが，将来の NVLink 2.0 では，専用の通信バスを提供することでさらに効率的な GPU 間直接通信が可能になると考えられる．TCA はそのプロトタイプ実装として PCIe Gen2 に基づく PEACH2（PCI Express Adaptive Communication Hub version 2）システムを用いて，ノードを跨ぐ複数の GPU 間での直接通信を実現し，その低レイテンシ通信によって GPU 間通信の性能が向上している（以降 PEACH2 に基づく TCA を「TCA/PEACH2」と定義する）．また，現状では，TCA/PEACH2 の適用範囲は PCIe などのハードウェア面の制約により，サブクラスタと呼ばれる最大で 16 ノードまでの直接結合にとどまる．このように，GPU 間直接通信は，ノード内の GPU 間から最大で十数ノードまでの直接までしか対応いないため，これだけで大規模なシステムを構築することは困難である．一方で，GPU クラスタでは，システム内の全てのノードがコモディティネットワークである InfiniBand などで相互接続されていることが一般的である．

そこで，GPU 間直接通信機構とコモディティネットワークによるハイブリッド通信を考える．ハイブリッド通信では，GPU 間直接通信機構をコモディティネットワークで接続されている GPU クラスタシステムにおけるローカルな通信を加速するネットワークとすることで，コモディティネットワークだけでは達成できない通信性能の向上を実現する．しかしながら，単純にネットワークを同時に使うだけでは，通信性能の特徴が異なるネットワーク [15] を有効に活用することができない．そこで，ユーザは通信の種類に応じて最適な通信パスを選択する必要がある．

2.1.4　GPU クラスタにおける生産性

すでに，GPU クラスタにおけるプログラミングは，MPI や OpenMP に加えて GPU を制御するための CUDA や OpenCL，OpenACC などのプログラミングモデルを用いる必要があり，プログラミングのコストが高く，アプリケーションの生産性が低い．さらに，2.1.2 節で述べたように，GPU と CPU によるワークシェアリングでは，MPI+OpenMP+CUDA などに加えて，各計算リソースに応じて最適なデータサイズを割り当てる必要がある．静的にこの割合を決定するためには，割合の値についてパラメータサーチを行う必要がある．しかし，同じ問題であっても，問題サイズや使用するノード・GPU の数が変わる場合，必ずしも同じ配分比が最適であるとは限らない．そのため，この割合はプログラムの実行中に動的に決定することが望ましいが，これを実現するにはプログラミングのコストがより高くなる．

同様に，2.1.3 節で述べたように，コモディティネットワークと GPU 間直接通信機構によるハイブリッド通信は，通信の種類とネットワークの特性から，最適なネットワークを選択する必要がある．しかしながら，最適なネットワークの選択には，ユーザがアプリケーションの通信特性と各ネットワークの特徴を熟知している必要があり，プログラム中に通信の種類ごとに条件分岐文などを挿入しなければならず，非常にプログラムの生産性が低い．

以上のように，GPU クラスタにおける計算や通信のリソースを最大限に利用するためには，通常の GPU クラスタにおけるプログラミングに加えて，非常に複雑な処理を記述する必要があるため，プログラミングのコストが高く，アプリケーションの生産性が著しく低下してしまうことが大きな問題となっている．

2.1.5　関連研究

GPU クラスタにおける，GPU と CPU によるハイブリッドワークシェアリングに関する研究がいくつかある．

本研究で用いている StarPU のように，ノード内の CPU と GPU にデータを割り当て，ワークシェアリングを行うランタイムシステムに Qilin[19] がある．Qilin は，C++ ベースの API を用いて CPU 用のマルチスレッドコードと GPU 用の CUDA コードを出力する．また，Qilin には単にデータを CPU と GPU に割り当てる機能だけでなく，GPU と CPU に割り当てるデータの割合を変更することができる．Qilin によるアプリケーションの実行は，事前にいくつかのデータセットを用いてデータをデータベースに蓄積し，割合を予測する training run と，蓄積したデータから得られた最適な割合を用いて，アプリケーションを実行する reference run から成る．StarPU では，CPU の割り当て単位はコアであったが，Qilin では複数のコアを持つソケット単位でデータの割り当てを行うため，コア単位の割り当てに比べると CPU のキャッシュをより有効に使える可能性がある．しかしながら，Qilin はノード内のみのワークシェアリングだけにしか対応していないことや，ノード内の 1 CPU と 1 GPU でしか実行ができないという制約がある．これでは，HA-PACS のような複数の CPU ソケットや GPU を搭載した GPU クラスタ環境では，計算リソースを有効に活用できない．一方，本実装に用いた StarPU は，マルチノード環境に対応していることやノード内に複数の CPU ソケットや GPU が存在する環境でも，計算リソースを全

て活用できる．さらに，2015 年現在も開発が続けられているため，最新のアーキテクチャへの対応もより迅速に行うことが可能である．

荒木らの研究 [20] は，HPF（High Performance Fortran）を拡張することで，プロセッサに割り当てるデータサイズを調節する GEN_BLOCK 分割を提案している．これによって，プロセッサの性能が異なるシステムを組み合わせた分散メモリ環境において，プロセッサごとに割り当てるデータサイズを指示文によって変更することができる．しかしながら，計算機ごとに性能が異なる場合や実行中にプロセッサの性能が変わる環境では，静的にデータの割り当てを決定することが難しい．そこで，時間発展ループ中における各プロセッサの実行時間の差に着目し，これらを最小にするように動的にデータサイズの調整を行う機構を導入している．動的負荷分散によって得られたデータサイズを用いることで，等分割に対して最大で 2 倍の高速化を達成した．本研究においても，同様な動的負荷分散機構を導入することで，等分割に対して性能向上を達成している．

大島らの研究 [7] や遠藤らの研究 [8, 9] では，多くのアプリケーションで用いられている行列積に対して，CPU とアクセラレータによるワークシェアリングを行っている．使用している CPU やアクセラレータの組み合わせは異なるが，両研究ともワークシェアリングによる行列積を HPL 内の GEMM（General Matrix Multiply）に適用することで，アクセラレータや CPU 単体による HPL の性能に対して高い演算性能を達成している．また，両研究に共通することとして，CPU とアクセラレータに割り当てる部分行列のサイズに着目し，最適な演算性能が得られるサイズをパラメータサーチで求めていることである．しかし，最適な部分行列サイズは，全体の問題サイズだけでなく，実行する環境によって値が大きく異る場合があり，これらの研究のように静的に値を求める手法には限界がある．本研究で提案しているハイブリッドワークシェアリングのフレームワークでは，HPL 内の GEMM ルーチンのようにプログラムの一部に対しても適用することができるため，HPL の実行中に最適な部分行列サイズを決定することができる．また，大島ら [7] や遠藤らの研究 [8] では，あからじめアクセラレータで高い演算性能が得られやすいデータサイズを基準にし，最適な部分行列サイズサイズを求めている（例えば，GPU は 72 の倍数，ClearSpeed は 288 の倍数）．このように，利用する環境に精通しているユーザは，本フレームワークを用いることでより高速に，高い演算性能が得られる部分行列サイズを決定することも可能である．演算性能に関して，大島らの研究 [7] では，CPU と GPU に 50% ずつデータを割り当てた時に，GPU のみの HPL に対して 1.45 倍の演算性能を達成している．遠藤らの研究 [8, 9] では，CPU に 35%，ClearSpeed に 32%，GPU に 33% のデータを割り当てた時に，CPU のみの HPL に対して 2.28 倍の演算性能を達成している．これらのデータの割合からわかるように，CPU の演算性能とアクセラレータの演算性能は非常に拮抗している．しかしながら，現在の環境，例えば，筑波大学の GPU クラスタである HA-PACS ベースクラスタにおける CPU のピーク性能は 332.8GLOPS/node，GPU のピーク性能は 2660GLOPS/node であり，ピーク性能比で約 8 倍の差がある．このため，ピーク性能で比較すると，ワークシェアリングによる効果が小さいように見える．一方で，アクセラレータはピーク性能を出すことが非常に難しいことが知られており，特にアクセラレータが処理するデータが十分に大きくない時には顕著である．そのため，ノードに割り当てられたデータサイズがある程度小さい時には十分にハイブリッドワークシェアリングの効果が得られると考えられる．

アクセラレータ向けのコンパイラとして PGI Accelerator Compilers[21] や HMPP Workbench[22] が挙げられる．これらは GPU を含めた様々なアクセラレータを対象とした指示文を提供する．PGI

Accelerator compilers は NVIDIA 社の CUDA が動作する GPU 向けのソースコードを生成できる．HMPP Workbench はバックエンドコンパイラとして CUDA や OpenCL を用いているため，逐次のソースコードに指示文を挿入することで，マルチコア CPU や GPU 向けのプログラムを生成する．しかし，これらの環境では CPU または GPU での実行を対象としているため，GPU クラスタにおける GPU/CPU ハイブリッドワークシェアリングをすることができない．XMP-dev/StarPU は元になっている XMP が PGAS モデルを提供しているため，GPU クラスタ上で容易な並列化と GPU/CPU ワークシェアリングが可能である．

XMP-dev/StarPU は，プログラムの記述はオリジナルの CUDA ベースの XMP-dev に準拠している．そのため，李らの研究 [23] にあるように Laplace 方程式のソルバーについても適用が可能であると考えられる．また，XMP-dev のベースとなっている XMP で記述されたコードも，一部を変更することで XMP-dev 対応することが原理的に可能である．中尾らの研究 [24] では XMP による共役勾配法の並列化が行われているように，XMP-dev/StarPU は様々なアプリケーションへの対応が可能であると考えられる．

一方，MAGMA[25] における NVIDIA の GPU 向けの BLAS（Basic Liner Algebra Subprograms）に StarPU を適用した研究 [26] がある．これはライブラリレベルで GPU と CPU のワークシェアリングを行っており，言語レベルで GPU と CPU によるワークシェアリングを行う XMP-dev/StarPU のフレームワークにも適用が可能だと考えられる．性能に関しても，StarPU を用いることでコレスキー分解を GPU1 台のみ使う実行時間に対して，Intel Nehalem X5550 6cores，NVIDIA FX5800 3 台という環境で最大 4 倍近い速度向上が得られている．XMP-dev/StarPU では基本的にループ分割によるワークシェアリングで記述できる問題については，より柔軟に両者のワークシェアリングを記述できる．また，BLAS のようなライブラリレベルではなく，ユーザのオリジナルコード全体が並列化・GPU/CPU ワークシェアリングの対象となるため，応用範囲が非常に広い．さらに，ライブラリではなくコンパイラレベルで並列化やハイブリッドワークシェアリングが可能である．

一方で，GPU 間直接通信を実現する研究もいくつかある．

TCA/PEACH2 のように，GPU 間でデータの直接通信を実現する研究がいくつかある．そのなかで，APEnet+[27, 28] は，現在の Kepler アーキテクチャの前の世代である Fermi アーキテクチャの GPU 間通信を実現した研究である．PEACH2 のように，FPGA（Field-Programmable Gate Array）を用いたネットワークインタフェースであり，3-D トーラスネットワークを構築する．しかしながら，APEnet+ は，独自のプロトコルを用いて GPU 間通信を実現し，GDR ではない GPUDirect P2P プロトコルを用いている点で，TCA/PEACH2 とは異なる．APEnet+ は，MPI に類似した通信 API のみを提供しており，プログラミングのコストは MPI による記述とほぼ同等と考えられる．また，APEnet+ とコモディティネットワークによるハイブリッド通信などの研究は現段階では行われていない．一方，本研究では，XcalableACC という高いレイヤのプログラミングを用いて，TCA/PEACH2 と InfiniBand のネットワークを有効に活用できる点において優位であると考えられる．

GPUDirect では，コモディティネットワークの InfiniBand と NVIDIA の Kepler アーキテクチャを搭載した GPU を用いることで，GPU 間直接通信を実現している [29]．これを MPI のインタフェースに適用したものとして，オハイオ州立大学の MVAPICH2-GDR（MV2GDR）がある [10]．TCA/PEACH2 も同様に GDR を用いた通信であるが，InfiniBand による通信では，HCA 間の通信で PCIe とは異な

るプロトコルを用いる．一方，PEACH2 は PCIe のパケットをそのまま利用できるのでプロトコル変換のオーバヘッドがなく，InfiniBand による GDR と比べて低いレイテンシで通信が可能である．ハイブリッド通信でも InfiniBand を用いるが，すべての通信が必ずしも InfiniBand を経由することはないので，オーバヘッドは低減される．

NVIDIA 社が提供する NVLink[18] は，ノード内の GPU 間を直接結合する技術である．現在の実装は，PCIe Gen3 に基づいているが，今後専用の通信バスを提供することでより高速な GPU 間直接通信が可能になると言われている．これは，我々が提案してきた TCA コンセプトであり，これまで HA-PACS/TCA で実証してきたことは意義があるものであると言える．また，NVLink はノード内の GPU 間通信を加速するものであり，ノード間の通信には今までどおり，InfiniBand などのコモディティネットワークを併用することが考えられる．そのようなシステムでは，ノード内の GPU に問題をどう分割するかが重要であり，NVLink による通信は，本論文における TCA を用いた通信に共通するものがある．よって，本論文で考察した TCA 通信と InfiniBand 通信による次元方向での使い分けは，今後このようなシステムにおける問題分割にも一部適用可能であると考えられる．

このように，GPU 間直接通信を実現するシステムや通信ライブラリはいくつかあるが，コモディティネットワークなどと同時並行的または通信の種類によって使い分けを高並列言語のレベルで行う先行研究はまだ存在していない．今後，NVLink と InfiniBand のように，ネットワークもヘテロジニアス性を持つことが一般的になると想定されており，このような環境において本研究のようなネットワークの使い分けが適用可能であると考えている．

2.2　研究の目的

前節で述べたように，GPU を代表とするアクセラレータを搭載したクラスタでは，システムのリソースを最大限利用するには非常に複雑なプログラミングを行う必要があり，アプリケーションの生産性が大きく低下してしまう．そこで，高並列言語に着目し，アプリケーションの生産性とクラスタシステムの効率的利用を両立するフレームワークを提供する．本論文では，大規模分散メモリ環境における次世代のプログラミング言語として開発されている，PGAS 並列プログラミング言語 XcalableMP（以降「XMP」と略す）に着目し，そのアクセラレータ搭載の並列計算機向けの拡張言語 XcalableMP acceleration device extension（以降「XMP-dev」と略す）による GPU と CPU によるワークシェアリングならびに，XcalableACC（以降「XACC」と略す）による TCA/PEACH2 と InfiniBand によるハイブリッド通信を実現し，これらを容易に実現するためのフレームワークを提供することが目的である．ここで，一つのプログラミングフレームワーク上で演算および通信を効率化したほうが，生産性の面でユーザに対しての貢献が大きいと考えられる．しかしながら，本論文ではまず，演算・通信のそれぞれに対して効率化を検討するのが重要と考え，フレームワークの集約については今後の課題としたい．

第 3 章

並列プログラミング言語 XcalableMP

本章では，先行研究である並列言語 XcalableMP ならびに，そのアクセラレータ搭載の並列計算機向けの拡張言語仕様である XcalableMP acceleration device extension および XcalableACC の概要とプログラム例を示す．

3.1 GPU クラスタにおける高並列言語

分散メモリ環境におけるプログラミングでは，MPI や OpenMP など異なるプログラミングモデルを組み合わせる必要がある．アクセラレータを搭載したクラスタシステムでは，さらにアクセラレータを管理するために CUDA や OpenCL，OpenACC などのプログラミングモデルを組み合わせるため，非常にプログラミングのコストが高い．このような環境で，アクセラレータと汎用プロセッサを並行して利用することや，異なる相互結合網による GPU 間通信を行うためには，より複雑なプログラミングが必要になり，アプリケーションの生産性が低下してしまう．そこで，本論文では，GPU クラスタを対象に，高並列言語のフレームワークの中に GPU と CPU によるハイブリッドワークシェアリングおよび TCA/PEACH2 と InfiniBand によるハイブリッド通信を適用することで，プログラミングの生産性を維持しつつ，高い演算・通信性能を実現するフレームワークの構築を行う．

そのために，本研究では，分散メモリ環境向けの高並列言語である XMP とそのアクセラレータ搭載のクラスタ向け拡張言語に着目し，提案するハイブリッドシステムを同言語に適用することで，高い生産性と性能の両立を実現する．

3.2 XcalableMP

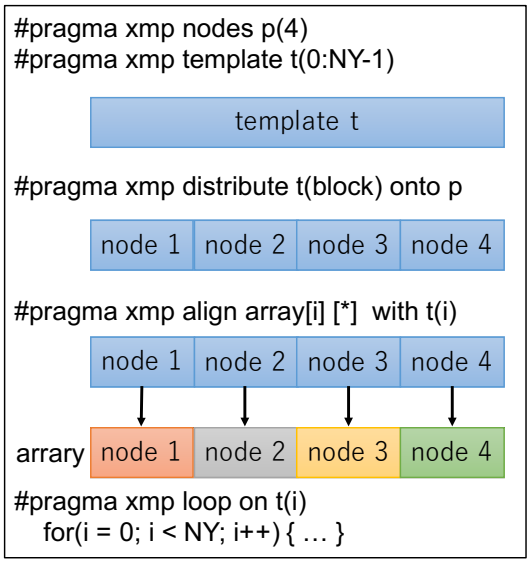
XMP に関しては文献 [30, 31, 32, 24] に詳しいが，ここでは本論文を理解するための最小限の説明を述べる．XMP は，分散メモリ型並列計算機上でのプログラミングを行うための PGAS 並列言語である．C 言語や Fortran 言語で記述された逐次のプログラムコードに OpenMP に類似した指示文を挿入することで，データの分散・同期および並列計算を行うことができる．そのため，従来の MPI による記述と比較して少ない記述量で並列化が可能であり，プログラムのコーディング時間が短縮され，アプリケーションの生産性が大幅に向上する．

```
int array[NY][NX];

#pragma xmp nodes p(4)
#pragma xmp template t(0:NY-1)
#pragma xmp distribute t(block) onto p
#pragma xmp align array[i][*] with t(i)

int main(void) {
   int i, j, sum = 0;
#pragma xmp loop on t(i)
   for(i = 0; i < NY; i++) {
      for(j = 0; j < NX; j++) {
         sum += array[i][j];
      }
   }
#pragma xmp reduction (+:sum)
}
```

(a) XMP のサンプルコード



(b) template によるデータの分散

図 3.1　XMP による記述例と template によるデータの分散

3.2.1　プログラミングモデル

XMP は，「グローバルビューモデル」と「ローカルビューモデル」という 2 つのプログラミングモデルを提供する．グローバルビューモデルは，分散メモリ上に跨った配列を各ノードに分散，並列処理を行うプログラミングモデルであり，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する通信を行うプログラミングモデルである．本論文では，グローバルビューモデルの指示文に対して，ワークシェアリングおよびハイブリッド通信の拡張を行っているため，グローバルビューモデルについてのみ言及する．

グローバルビューモデルは，分散メモリ上において OpenMP-like な指示文による並列プログラミングを可能にする．また，XMP では実行単位のプロセスを「ノード」と定義している．このノードは MPI のプロセスと同等であるため実行モデルは SPMD（Single Program Multiple Data）である．基本的にはソースコード上のデータに対し各ノードで同じ処理が行われるが，XMP の指示文でデータが分割されている場合は，データが各ノードに分散され，各ノードでは割り当てられたデータのみが処理される．メモリアクセスはローカルメモリのデータに対する参照であるが，他ノードのデータを参照するには XMP の指示文を使い，ノード間通信をする必要がある．XMP は SPMD モデルに基づくため，ノード間の同期は基本的に自動では行われない．そのため，ユーザが明示的に **barrier** 指示文等を用いて同期をする必要がある．

図 3.1 に XMP のプログラムの例と template によるデータの分散を示す．図 3.1(a) は，二次元配列 *array* の各要素の総和を計算するプログラムを XMP で記述した例であり，図 3.1(b) は図 3.1(a) におけるデータの分散を示している．「#pragma xmp」から始まる行が XMP の指示文である．**nodes** 指示文はプログラムを実行するノードの集合を宣言する．図 3.1 では，4 つのノードを用い，これは MPI にお

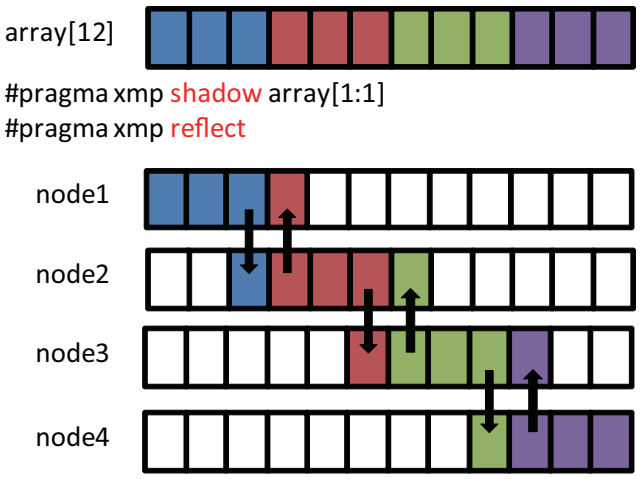


図 3.2　XMP の shadow 領域の動作

けるプロセス数 4 に相当する．次に，**template** 指示文でノードに跨った仮想的なインデックスの集合の宣言を行う．このインデックスはデータの配列・ループ文のイテレータを示す．XMP におけるデータの分散やループ文の分割には，この template が用いられる．**distribute** 指示文は，template t を各ノードにどのようにマッピングするかを宣言する．指示文のオプションとしてデータの分割方法を選択することでき，ここではブロック分割を行っている．最後に **align** 指示文によって，template t で宣言されたデータの分散を実際の配列 $array$ に適用し，データの分割を記述する．これによって，各ノードは割り当てられたデータ分だけをローカルメモリに持つことになる．

loop 指示文は，直後のループ文に対してノード集合 p でワークシェアリングを行う．ループ文のイテレーションの分割は template t によって設定されている．XMP では，データアクセスはローカルメモリを参照するため，イテレーションの分割とデータの分割の整合性をユーザがとる必要がある．これは，データ分割とループイテレーション分割の両方に，同一の template を適用することで記述できる．

また，XMP のデータアクセスはローカルのメモリを参照するが，隣接領域に依存するステンシル計算などの袖領域交換や，ノード間に跨ったデータすべてにアクセスする N 体問題のようなアプリケーションでは，他ノードに存在するデータが必要になる．そこで XMP は，典型的なノード間通信を簡潔に記述するために，**shadow** 指示文と **reflect** 指示文を提供する．XMP では，他ノードと重複してデータを持つ領域を「shadow」と定義している．shadow 指示文と reflect 指示文の動作を図 3.2 に示す．shadow 領域は，他ノードに存在するデータを前もってローカルメモリに確保しておくことで，ローカルメモリ内だけで参照が完結することを可能にする．図 3.2 の shadow 指示文では [1:1] のように shadow 領域の下限と上限を設定する．この例では，隣り合うノードに 1 要素分の shadow 領域を設定する．各ノードでの計算結果を shadow 領域に反映（同期）させる操作は，reflect 指示文を用いることで実行される．ユーザは適当なタイミングで reflect 指示文を挿入し，正しい同期を保証する必要がある．reflect 指示文に対応する箇所ではノード間の peer-to-peer 通信が適宜実施される．また，shadow 領域を配列全体に適用することで全てのノードが同じ配列を持つことができる．これを XMP では「full shadow」と定義している．full shadow の reflect は，MPI_Allgather と同等の通信が行われる．

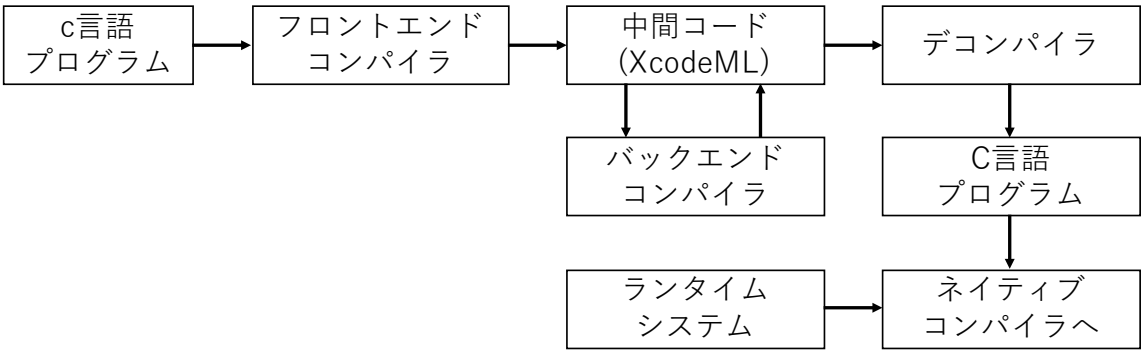


図 3.3　XMP コンパイラの構造

3.2.2　XMP のコンパイルシステム

図 3.3 に XMP コンパイラの構造を示す．図 3.3 より，XMP の指示文が挿入されたコードを XMP のフロントエンドコンパイラがコードの解析を行い，XML で記述された中間コード XcodeML に変換される．この中間ファイルを XMP のバックエンドコンパイラが MPI などの並列化した C 言語のプログラムとして出力する．この XcodeML に変換するフロントエンドコンパイラ，XcodeML から C 言語のプログラムを生成するプロセスは XMP のベースとなっている Omni OpenMP Compiler[33] が用いられている．これはプログラムの解析や変換を行うためのパーサーなどのプログラム，Java のライブラリを含んだプログラム群を有している．本研究やベースとなっている XMP-dev や XACC もまたこれを利用し，プログラムの解析から C 言語のプログラム生成を行っている．

3.3　XcalableMP acceleration device extension

XMP-dev[23] は，アクセラレータを搭載した並列計算機で使える高性能で生産性の高いプログラミングモデルとして提案されている XMP の拡張言語仕様である．ここで言う device は，ホスト（CPU）の処理の一部を請け負うアクセラレータを表す．XMP-dev が扱うアクセラレータは，ホストと独立したメモリ（デバイスメモリ）を持っている．XMP-dev では，XMP にいくつかの指示文を追加することで，分散メモリ環境におけるノード間のデータおよび処理の分割という従来の XMP の機能に加え，ホスト〜デバイス間のデータ転送や，デバイス上で loop 文の並列化などを簡潔に記述することができる．これらの指示文と従来の XMP の指示文を組み合わせることで，アクセラレータを搭載するクラスタ上での並列化が可能になる．CUDA や OpenCL を MPI と組み合わせ使うことなく，プログラムを簡潔に記述できる点が大きなメリットである．アクセラレータ間のデータ交換はホストメモリを介して行う．そのため，ユーザが XMP-dev の指示文でホスト〜デバイス間の通信を指示する必要がある．XMP-dev はバックエンドコンパイラに CUDA を用いた XMP-dev/CUDA[23] と OpenCL を用いた XMP-dev/OpenCL[34] がある．本研究では，XMP-dev/CUDA を用いて GPU/CPU ワークシェアリングの拡張を行う．

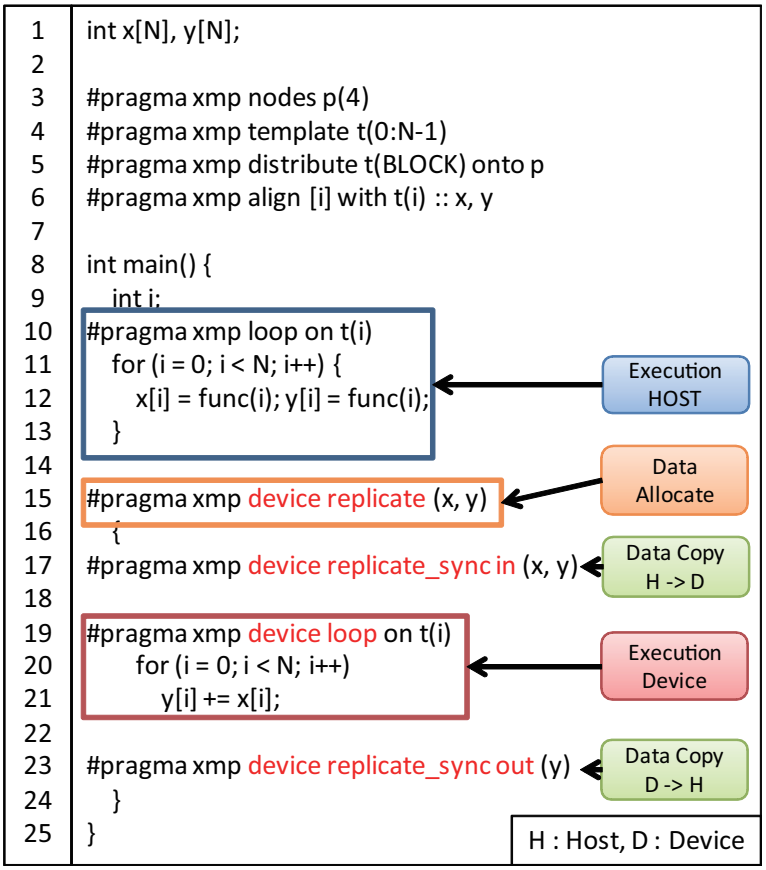


図 3.4　XMP-dev のサンプルコード

3.3.1　XMP-dev のプログラミングモデル

図 3.4 に XMP-dev のプログラム例を示す．XMP-dev は XMP の拡張仕様であるため，従来の指示文をそのまま利用することが出来る．図 3.1 の 3〜6 行目は図 3.1 と同様である．10〜13 行目は XMP の loop 指示文であり，ホストの CPU 上で実行される．15〜24 行目までが XMP-dev の指示文であり，指示文は「#pragma xmp device」から始まる．以下に XMP-dev の拡張指示文を示す．

```
#pragma xmp device replicate (list)
```

device replicate 指示文はデバイスメモリへ配列を確保するものである．アクセラレータでの計算に使う配列データは，ユーザが図 3.4 の 6 行目でホストメモリ，15 行目でデバイスメモリに確保しなければならない．図 3.4 では，16〜24 行目のスコープ内において，デバイス上でのメモリ確保が保証されており，スコープから抜けるとデータは解放される．

```
sync_clause := in (list) | out (list)
#pragma xmp device replicate_sync sync_clause
```

device replicate_sync 指示文は replicate 指示文のスコープ内で利用することができる．これは，ホストメモリとデバイスメモリ間のデータ通信を行う．通信の方向は sync_clause で制御する．「in」はホストからデバイスへ，「out」はデバイスからホストへの通信であることを示す．replicate スコープを抜けた後，ホストでデータを参照する必要がある時には，必ず replicate_sync out を使わなければならない．

```
#pragma xmp device loop
    loop-statement
```

device loop 指示文は XMP の loop 指示文同様に，直後の for loop をデバイス上でワークシェアリングする．この for loop は，XMP-dev のコンパイラがデバイスで動作するデバイス関数とその関数を呼び出すための関数に変換される．アクセラレータ上では，多数のスレッドが動作するため，XMP-dev では 1 スレッドに loop 文の 1 反復の計算を割り当てるように実装されている．

3.4　XcalableACC

XACC は文献 [35, 36] に詳しいが，ここでは本論文を理解するための最小限の説明をする．XACC は，アクセラレータを搭載した並列計算機向けの PGAS 言語である．既存の並列言語 XMP と，アクセラレータ向けの指示文ベースのプログラミングモデルである OpenACC[37] を組み合わせた拡張言語仕様である．さらに，これらに加え，アクセラレータ間のデータ通信をするために XMP の指示文を拡張することで，アクセラレータ搭載の並列計算機におけるプログラムの生産性と性能の両立を可能にする．

先行研究である XMP-dev では，XMP-dev 独自の指示文を用いて CUDA や OpenCL などコードを source-to-source で変換していたが, XACC ではより一般的なプログラミングモデルである OpenACC を用いることでより高い可搬性のあるプログラムを記述することができる．本研究では，OpenACC コンパイラとして筑波大学の田渕や理研 AICS が中心となって開発している Omni OpenACC Compiler[38, 39] を用いる．また，XMP-dev において，アクセラレータ間でデータの交換をする場合，ホスト〜アクセラレータ間のデータ通信とノード間の通信を行う指示文を組み合わせていたが，XACC ではアクセラレータ間の通信だけでなく，リモートのホスト〜アクセラレータ間の通信が可能な指示文を提供しており，プログラミングの見通しが向上している．

3.4.1　プログラミングモデル

図 3.5 に XACC のサンプルコードを示す．OpenACC は，プログラムの一部をアクセラレータにオフロードするための指示文ベースのプログラミングモデルであり，「#pragma acc」から始まる指示文を持つ．2〜5 行目は，分散配列 *a[]* を定義するための XMP の指示文であり，データの分散や並列実行主体への割り当てを記述する．7 行目は OpenACC の **data** 指示文で，8〜14 行目の領域はデバイスメモリでのデータの確保，スコープに入るときに転送が行われ，スコープを抜けるときに，データがホストに書き戻される．9〜13 行目は XMP の **loop** 指示文で各ノードにデータを分散し，OpenACC の **parallel** 指示

```
1  double a[N];
2  #pragma xmp nodes p(4)
3  #pragma xmp template t(0:N−1)
4  #pragma xmp distribute t(block) onto p
5  #pragma xmp align a[i] with t(i)
6  ...
7  #pragma acc data copy (a)
8    {
9  #pragma xmp loop on t(i)
10 #pragma acc parallel loop
11       for (int i = 0; i < N; i++) {
12         a[i] += 1.0;
13       }
14   }
15 ...
```

図 3.5　XACC のサンプルコード

文が XMP の分散配列をアクセラレータ上で実行するようにスレッド分割を行う.

第 4 章

GPU/CPU ハイブリッドワークシェアリング

本研究は，XMP のアクセラレータを持つ並列計算機向けに拡張した言語仕様である XMP-dev の一実装として，GPU と CPU によるハイブリッドワークシェアリングを実現する XMP-dev/StarPU コンパイラおよびランタイムシステム [40, 41, 42] を提案する．XMP-dev は，XMP が本来提供している分散メモリノードへのデータの分割・通信，並列処理の機能に加え，各ノードでの処理の一部を GPU にオフローディングする機能も有している．しかし，GPU を搭載した並列システムで一般的に用いられる実行モデルでは，GPU にオフロードされた部分はすべて GPU により並列処理され，CPU とのワークシェアリングを行うことができない．XMP-dev/StarPU は，StarPU をバックエンドスケジューラとして用いることで，計算をタスクという単位で GPU と CPU へスケジュールすることで，GPU/CPU のワークシェアリングを行うことが可能である．さらに，プログラム中でデバイスごとにタスクサイズをコントロールする機能を提供することで，ユーザが問題の性質や問題サイズ，実行環境によって変わるタスクサイズを容易に最適化することが可能になる．

4.1 StarPU

StarPU に関しては文献 [43, 44] に詳しいが，ここでは本論文を理解するために必要な最小限の説明をする．StarPU では，計算に必要なデータの集合，実行の単位を「タスク」と定義している．StarPU は，このタスクを様々な計算リソースに割り当てたり，タスク間のデータの依存関係を調整したりすることが可能なランタイムシステムである．対象としている計算リソースはマルチコア CPU，GPU および Cell Broadband Engine だけではなく，2015 年 3 月にリリースされた StarPU v1.2.0 からは Intel Xeon Phi も対象としている．本論文では，マルチコア CPU，GPU （特に NVIDIA の CUDA が動作する）についてのみ言及する．また，StarPU はタスク間のデータ依存の制御をするために，全ての計算リソースで共有するデータプールに配列データを登録する必要がある．タスクが生成された時に，必要なデータがデータプールから割り当てられ，計算を実行することが可能になる．

4.1.1　codelet

StarPU では，タスクを制御するために「codelet」と呼ばれる構造体を用いる．これを，タスク生成時にポインタとして渡すことで，どの計算リソースで実行するか，どの関数を実行するか，計算に必要な配列などの情報を登録する．実行される関数が複数存在する場合，関数ごとに codelet を生成する必要がある．codelet の例を以下に示す．

リスト 4.1　StarPU codelet の例

```
1 starpu_codelet cl = {
2   .where = STARPU_CPU|STARPU_CUDA,
3   .cpu_func = cpu_fnction,
4   .cuda_func = cuda_function,
5   .nbuffers = 10
6 };
```

where メンバは，タスクを実行するデバイスを指定する．この例では，CPU と GPU による並列実行であることを示している．cpu_func および cuda_func メンバはそれぞれ，CPU，GPU で実行する関数を指定しており，nbuffers メンバで計算に必要な配列の数を指定する．

4.1.2　メモリ管理

計算に必要なデータを，StarPU のデータプールに登録することで，プログラム中のデータの依存関係を容易に判断することができる．StarPU では，データを starpu_data_handle という型で登録する．タスクはこの handle を参照することで依存関係が保証され，正しい値を参照することができる．StarPU では，計算に必要なデータは計算リソースに最も近いメモリ（CPU はメインメモリ，GPU はデバイスメモリ）にデータが存在するように，計算が実行される前に通信が発生する．例えば，ある GPU 上で更新した値をホストの CPU で参照するとき，デバイスメモリからホストへのデータ転送が実行され，常に最新の値を参照することができる．このデータ参照のポリシーは，StarPU のオプションで制御することが可能である（Read-only, Write-only, Read Write, etc.）．また，GPU で計算した時に使ったデータは再利用性が高いため，ユーザが明示的にデータを破棄しない限り，そのままデバイスメモリ上に存在し続ける．これによってホスト-デバイス間の通信を最小限にすることが可能である．

また，StarPU のデータプールに登録されているデータを，MPI などの通信ライブラリを使って他ノードと通信したい時がある．このように，アプリケーションが直接データプールのデータにアクセスしたい時には，**starpu_data_acquire()** 関数でアプリケーションにデータ管理を移譲し，アプリケーション側でデータを変更した後に，**starpu_data_release()** 関数で管理を StarPU に戻すことで実現できる．

4.1.3　タスクの実行

図 4.1 にシングルノードにおける StarPU の動作を示す．StarPU では配列の確保や初期化を行った後，**starpu_data_register()** 関数を使って StarPU のデータプールにデータを登録する．このプール

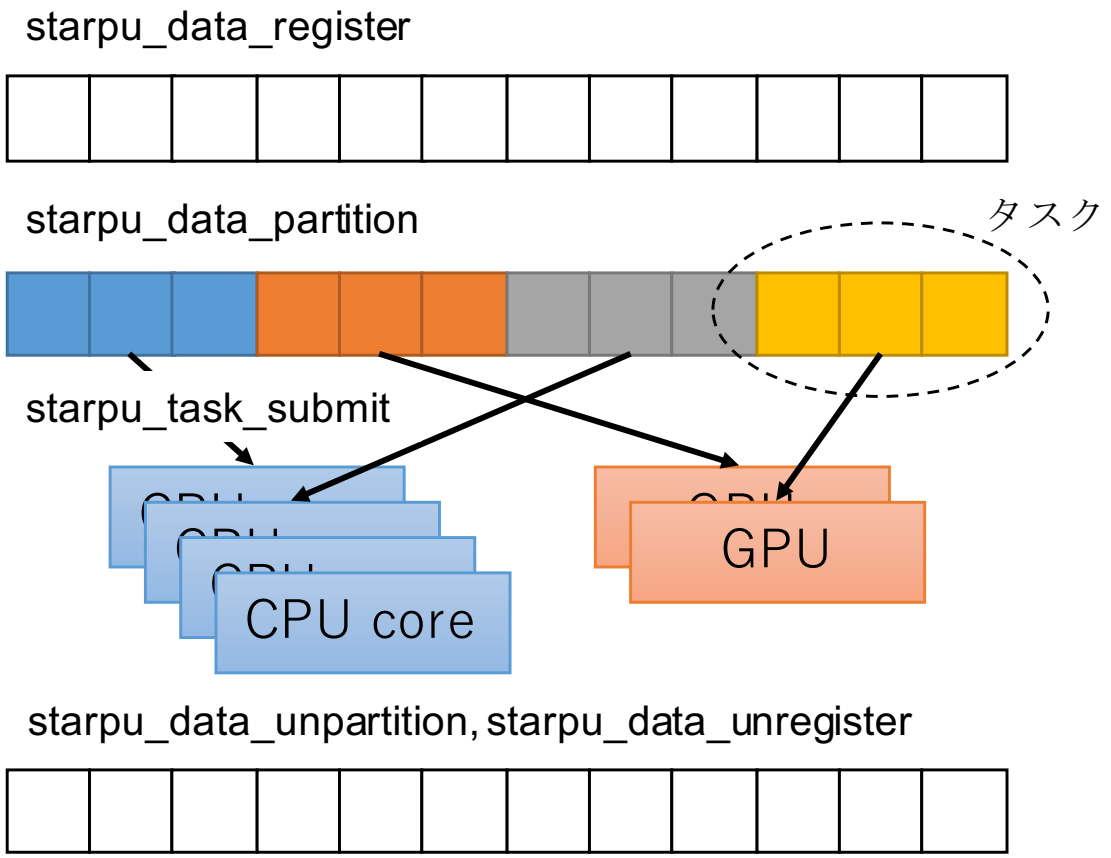


図 4.1　シングルノード上での StarPU の動作

に登録されたデータは，StarPU のタスクとしてデバイス上で実行の対象となる．しかしながら，データプールに登録された直後の配列に対して，StarPU は配列全体を 1 つのタスクとして扱うため，1 つの計算リソース上でしか実行できない．そこで，**starpu_data_partition()** 関数を用いて配列データを細かく分割する．本論文では，この分割した単位を「チャンク」と定義する．複数のチャンクをまとめてタスクすることで，**starpu_task_submit()** 関数を用いて複数の CPU や GPU に計算を割り当てることが可能になる．これによって，ヘテロジニアスな環境でのワークシェアリングが可能になる．計算が終わった後に，各デバイスメモリ上に存在するデータをメインメモリに戻すのが **starpu_data_unpartition()** 関数である．そして，**starpu_data_unregister()** 関数を用いて StarPU のデータ管理を終了する．

4.1.4　サンプルコード

本節では，StarPU を用いて記述されたプログラムのサンプルをリスト 4.2 に示す．リスト 4.2 の 1〜9 行目までが CPU で実行される関数である．11〜17 行目は StarPU の codelet であり，先ほど設定した CPU の実行関数をセットする．この例では，CPU で実行する関数の他にも GPU 用の関数を作成し，CPU と同様に codelet に紐付けする．これによって，StarPU のスケジューラが GPU または CPU のどちらかにタスクを振り分けたとしても実行が可能になる．19 行目からはプログラムの main 文である．22〜23 行目で，StarPU のデータプールに計算に利用する配列を登録する．StarPU では，StarPU が

データにアクセスするにために data_handle_t という型を定義している．25〜28 行目は filter と呼ばれる配列を分割する規則の設定である．これを用いて，31 行目ではデータの分割を行う．34〜44 行目はタスクの生成と発行を行う．ここで，タスクへ codelet の紐付けや必要な配列をセットし，各計算リソースへタスクを分配する．そして，計算が終わった後にデータの管理をアプリケーションに戻すために 51 行目では unregister を行う．以上が，StarPU を用いたプログラムの一例である．

リスト 4.2　StarPU のコード例

```
1  void calc_force_cpu(void *buffers[], void *cl_arg) {
2    double *px = (double *)STARPU_VECTOR_GET_PTR(buffers[0]);
3    double *py = (double *)STARPU_VECTOR_GET_PTR(buffers[1]);
4    ...
5
6    for (i = 0; i < n; i++) {
7      ...
8    }
9  }
10
11 struct starpu_codelet cl_calc = {
12   .where = STARPU_CPU|STARPU_CUDA,
13   .cpu_funcs = {calc_force_cpu, NULL},
14   .cuda_funcs = {calc_force_gpu, NULL},
15   .nbuffers = 7,
16   .modes = {STARPU_R, STARPU_R, STARPU_R, STARPU_R, STARPU_RW, STARPU_RW,
         STARPU_RW},
17 };
18
19 int main(int argc, char **argv) {
20   ...
21   starpu_data_handle_t px_h, py_h, pz_h, m_h, vx_h, vy_h, vz_h;
22   starpu_vector_data_register(&px_h, 0, (uintptr_t)px, N, sizeof(double));
23   starpu_vector_data_register(&py_h, 0, (uintptr_t)py, N, sizeof(double));
24
25   struct starpu_data_filter f = {
26     .filter_func = starpu_block_filter_func_vector,
27     .nchildren = PART,
28   };
29
30   for (t = 0; t < TIME_STEP; t++) {
31     starpu_data_partition(vx_h, &f);
32     ...
33
34     for (i = 0; i < PART; i++) {
35       truct starpu_task *task = starpu_task_create();
36       task->cl = &cl_calc;
37       task->cl_arg = &head_index;
```

```
38          task->cl_arg_size = sizeof(int);
39          task->handles[0] = px_h;
40          task->handles[1] = py_h;
41          ...
42          ret = starpu_task_submit(task);
43          STARPU_CHECK_RETURN_VALUE(ret, "starpu_task_submit");
44        }
45
46        starpu_data_unpartition(vx_h, 0);
47        ...
48
49        } // end of timestep
50
51    starpu_data_unregister(vx_h);
52    ...
```

4.2　XMP-dev/StarPU

4.2.1　XMP-dev/StarPU の概要

StarPU は，ノード内におけるデータの管理，データ転送，タスクの生成と発行などを担い，GPU クラスタなどのヘテロジニアスな環境において，ロードバランスを取ることが潜在的に可能である．しかし，4.1.4 節で示したように，StarPU を使ったアプリケーションの実装は逐次のコードから変更する場合，codelet の記述やデータの分割など，プログラミングコストが大きいことが問題である．また，StarPU のランタイムは MPI によるマルチノード上でデータの分散やタスクの実行が可能であるが，マスターノードによって全てのデータ管理やスケジューリングが行われる．そのため，プログラミングにおいて，データの分散などにノード番号を指定する必要があるためプログラムが複雑になりがちである．これに加えて，MPI などの通信ライブラリを用いる必要があるため，分散メモリ環境では更に複雑になることが容易に想像できる．

一方，XMP-dev は GPU クラスタなどのアクセラレータを搭載した並列計算機を対象としたプログラミング言語である．そのため，複数の GPU を用いた並列計算を，指示文ベースのプログラミングモデルで簡潔に記述することが可能である．しかしながら，XMP-dev は計算のすべてを GPU へオフロードするように設計されているため，GPU が計算を行っている間に CPU が空転してしまい，計算リソースを無駄に消費してしまう．そこで，本研究では XMP-dev と StarPU を組み合わせた XMP-dev/StarPU を提案し，マルチノード上での GPU/CPU ハイブリッド計算を容易に行うことができるフレームワークを提供する．これによって，プログラミングのコストが削減でき，プログラムの生産性が向上することや，システム中の計算リソースを有効活用することで，GPU のみを計算に利用した時よりも性能向上が期待できる．

XMP-dev におけるメリット

XMP-dev の device として StarPU を利用することを考える．現在，XMP-dev の実装において，

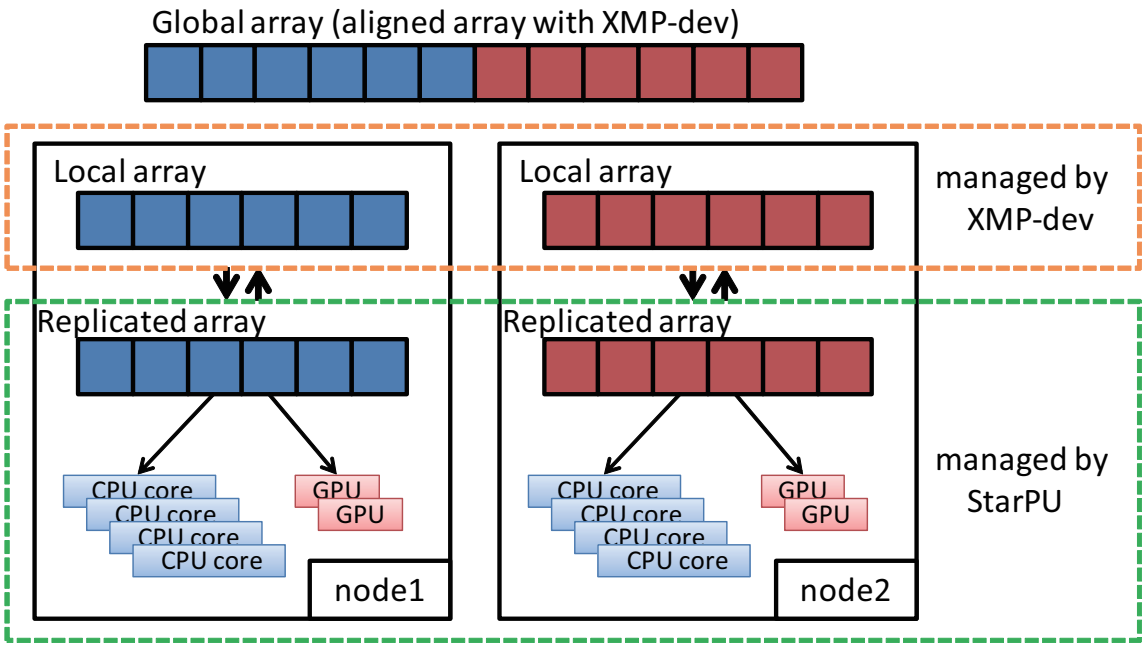


図 4.2　XMP-dev/StarPU の動作イメージ

バックエンドは CUDA と OpenCL があるが，双方とも計算は GPU のみで実行されている．そのため，GPU で計算を行っている間，CPU が空転してしまう．そこで，XMP-dev のバックエンドスケジューラとして StarPU を適用する．これによって，GPU/CPU の計算リソースを余すことなく利用することができるため，従来の XMP-dev に対して性能向上が見込める．

StarPU におけるメリット

StarPU はプログラミングが複雑になりがちなため，様々なアプリケーションに適用することが困難である．そこで，XMP-dev の指示文を拡張することで，StarPU のデータプールへの登録などを実行するための指示文やランタイムシステムを作成する．そして，XMP-dev が生成したデバイス関数を実行の対象とすることで，デバイスでの実行が可能になる．CPU のコードは逐次のコードをそのまま利用することができる．これによって，StarPU を用いて直接プログラムを記述する場合に比べ，容易に GPU/CPU のワークシェアリングを記述することが期待できる．さらに，シングルノードだけではなく，PGAS モデルに基づく分散メモリ環境での並列化を実現する．

4.2.2　実装の方針

本節では，本論文で提案している XMP-dev/StarPU コンパイラの設計，および実装の方針について述べる．図 4.2 に XMP-dev/StarPU の動作イメージを示す．XMP-dev/StarPU では，XMP-dev で実装されたアクセラレータ向けの指示文を拡張し，コンパイラおよびランタイムの変更を行うことで，XMP-dev のフレームワークの中に StarPU を取り込み，GPU/CPU のワークシェアリングを行えるようにする．

図 4.2 より，XMP-dev と同様に Global array は template によってノード間に分散される．XMP-dev

では，各 GPU に 1 ノードを割り当てる，いわゆるフラット MPI による実行であったが，XMP-dev/StarPU では CPU と GPU を含めたシステム 1 つをノードとする．Global array は XMP-dev によって各ノードに分散され，各ノードは Local array として Global array の部分配列を保持する．XMP-dev は，この Local array を GPU に転送し，計算を行い，データを戻した後ノード間でメッセージパッシングなどによってデータを交換していた．XMP-dev/StarPU では，この Local array はノード間通信に利用するのは変わらないが，Local array をそのまま GPU にオフロードするのではなく，ノード間通信用のバッファとして利用する．そして，Local array と同サイズの配列 Replicated array をホストメモリ上に確保し，適宜この Local array と Replicated array の間でデータの同期を行う．Replicated array を StarPU のスケジューラを通して GPU や CPU に割り当てる．StarPU を用いるにあたって，Replicated array をいくつかのチャンクに分割し，複数のタスクを生成して，ノード内の各計算リソースにスケジュールを行う．これによって，マルチノード上における GPU と CPU によるワークシェアリングが可能になる．

ここでなぜ，Replicated array を用いるかについて説明する．Local array を直接複数のチャンクに分割し，それをタスクとすることは可能であるが，Local array はノード間のデータ交換などにも利用されている．そのため，Local array を StarPU のデータプールに登録すると，コンパイラの様々な場所で acquire-release といったデータ管理の移譲のような制御が必要になる．そこで，簡易に実装するために Local array と同様の配列 Replicate array を作成し，データプールに登録をする．この配列をノード間通信などが必要になった時に Local array と同期することで Local array を分割した時と同等な動作が期待できる．

4.2.3　XMP-dev/StarPU のコンパイラ

本研究では，XMP-dev コンパイラを拡張し，XMP-dev/StarPU コンパイラおよびランタイムシステムを実装する．XMP-dev/StarPU では，XMP-dev にはなかった StarPU の制御が必要になる．そのため，従来の XMP-dev の拡張指示文の動作が変わるため，StarPU を扱うにあたってこれらの指示文の動作を再定義する．本節では，XMP-dev の指示文の拡張およびコンパイル後に生成される関数群について説明する．

XMP-dev において，**replicate** 指示文はデバイスメモリ上への配列データの確保を担っている．StarPU では，デバイスメモリ上への配列データの確保は StarPU のメモリマネージャが行うため，確保を明示的に指定する必要がない．そこで，XMP-dev/StarPU では，replicate 指示文は list に列挙された配列を StarPU のデータプール登録するための指示文として再定義する．これによって，replicate 指示文のスコープ内にある変数は，StarPU が管理することを保証する．replicate 指示文の変換後のコードをリスト 4.3 に示す．リスト 4.3 より，配列の様々な情報を持つディスクリプタの初期化は XMP-dev と同様である．しかし，StarPU を扱う際には，タスクという単位で計算を割り当てる必要がある．このタスクの数，イテレーション（for 文の計算範囲），次元数などの情報を保持するための構造体として XMP-dev/StarPU コンパイラでは_XMP_STARPU_TASK_DESC を導入する．これはタスクと同数が確保され，replicate 指示文のスコープの終了とともに解放される．

リスト 4.3　replicate 指示文の変換例

```
1 _XMP_starpu_init_task_desc(&(_XMP_STARPU_TASK_DESC));
2 _XMP_starpu_init_data_ALIGNED(&(_XMP_STARPU_HOST_DESC_px), &(
      _XMP_STARPU_DEVICE_DESC_px), &(_XMP_STARPU_HANDLE_px), (void *)(
      _XMP_ADDR_px), _XMP_DESC_px, _XMP_STARPU_TASK_DESC);
3 ...
4 _XMP_starpu_finalize_task_desc(_XMP_STARPU_TASK_DESC);
5 _XMP_starpu_finalize_data(_XMP_STARPU_HOST_DESC_px);
6 ...
```

次に **replicate_sync** 指示文について拡張した仕様について述べる．4.2.2 節で述べたように，Local array と Replicated array 間のデータの同期はユーザが明示的に行う必要がある．そこで，ホストとデバイス間のデータ転送を担っていた replicate_sync 指示文を，これらのメモリ間の同期として再定義する．sync_clause が in の時は Local array から Replicated array へ，out の時は Replicated array から Local array へのデータのコピーが行われる．最新のデータを計算に利用するためには，プログラマが適宜明示的な同期を記述する必要がある．replicate_sync in，replicate_sync out の変換後のコードをリスト 4.4 に示す．ランタイムの引数は XMP-dev と変わらないが，XMP-dev/StarPU のランタイム関数は_XMP_starpu からネーミングされる．

リスト 4.4　replicate_sync 指示文の変換例

```
1 _XMP_starpu_sync(_XMP_GPU_HOST_DESC_px, 600); //sync_in
2 ...
3 _XMP_starpu_sync(_XMP_GPU_HOST_DESC_px, 601); //sync_out
4 ...
```

最後に device loop 指示文について述べる．XMP-dev の device loop 指示文は，直後の for 文を GPU 上で並列処理するものである．XMP-dev は，1 つの for 文を 2 つの関数（デバイス関数と呼び出し関数）に変換し，置き換えていた．XMP-dev/StarPU では，GPU と CPU によるワークシェアリングや，StarPU の仕様に対応するために関数の形式や数が変わってくる．ここでは，3.3.1 節で変換するのに用いた指示文および for 文のプログラムを，XMP-dev/StarPU コンパイラで変換したコードとしてリスト 4.5 に示す．リスト 4.5 では_XMP_STARPU_から始まる関数が 4 つ生成されている．これらの関係は以下のとおりである．

- ホストプログラムから_XMP_STARPU_FUNC_0 が呼び出される．この関数は，StarPU のタスクの生成から変数のパッキングまで行い，最終的にタスクを発行し，計算を実行する．
- _XMP_STARPU_FUNC_0 でタスクとして発行され，実際に実行される関数が_XMP_STARPU_FUNC_0_CPU および_XMP_STARPU_FUNC_0_GPU である．
- _XMP_STARPU_FUNC_0_CPU には計算部分がそのまま記述されているが，CUDA を用いる_XMP_STARPU_FUNC_0_GPU 関数はその関数内でカーネル関数（_XMP_STARPU_FUNC_0_DEVICE）を呼び出す．

リスト 4.5　device loop 指示文の変換例

```
1  void _XMP_STARPU_FUNC_0_CPU(void *buffers[], void *cl_arg)
2  {
3    double *px;
4    void *_XMP_STARPU_DEVICE_DESC_px;
5    ...
6    _XMP_starpu_double_vector_get_ptr(&px,buffers,0);
7    ...
8    _XMP_starpu_unpack_cl_arg(cl_arg,&_XMP_STARPU_TASK_DESC,_XMP_DESC,&
         _XMP_STARPU_DEVICE_DESC_px, ...);
9    _XMP_starpu_set_iter(_XMP_STARPU_TASK_DESC,&_XMP_loop_init_i, ...);
10
11   for((i)=(_XMP_loop_init_i);(i)<(_XMP_loop_cond_i);(i)=((i)+(_XMP_loop_step_i)))
12     {
13       // CPU calc
14     }
15
16   _XMP_starpu_free_desc(_XMP_STARPU_TASK_DESC);
17   _XMP_starpu_free_desc(_XMP_STARPU_DEVICE_DESC_px);
18   ...
19 }
20
21 __global__ static
22 void _XMP_STARPU_FUNC_0_DEVICE(double px[102400], ...)
23 {
24   ...
25   _XMP_gpu_calc_thread_id(&_XMP_GPU_THREAD_ID);
26   _XMP_gpu_calc_iter(_XMP_GPU_THREAD_ID,_XMP_loop_init_i,_XMP_loop_cond_i,
         _XMP_loop_step_i,&i);
27
28   if((_XMP_GPU_THREAD_ID)<(_XMP_GPU_TOTAL_ITER))
29   {
30     // GPU calc
31   }
32 }
33
34 void _XMP_STARPU_FUNC_0_GPU(void *buffers[], void *cl_arg)
35 {
36   //same : _XMP_STARPU_FUNC_0_CPU
37   ...
38   _XMP_gpu_calc_config_params(&_XMP_GPU_TOTAL_ITER,&_XMP_GPU_DIM3_block_x, ...);
39   ...
40   {
41     dim3 _XMP_GPU_DIM3_block(_XMP_GPU_DIM3_block_x, _XMP_GPU_DIM3_block_y,
           _XMP_GPU_DIM3_block_z);
42     dim3 _XMP_GPU_DIM3_thread(_XMP_GPU_DIM3_thread_x, _XMP_GPU_DIM3_thread_y,
           _XMP_GPU_DIM3_thread_z);
```

```
43      _XMP_STARPU_FUNC_0_DEVICE<<<_XMP_GPU_DIM3_block, _XMP_GPU_DIM3_thread
            >>>(px,py,pz,j,m,vx,vy,vz,...);
44    }
45    _XMP_starpu_free_desc(_XMP_STARPU_TASK_DESC);
46    _XMP_starpu_free_desc(_XMP_STARPU_DEVICE_DESC_px);
47    ...
48  }
49
50  extern "C"
51  void _XMP_STARPU_FUNC_0(double px[102400], ...)
52  {
53    void *cl_arg;
54    int cl_arg_size;
55    void *_XMP_gpu_func_pointer;
56    void *_XMP_cpu_func_pointer;
57
58    _XMP_starpu_get_task_desc(_XMP_STARPU_TASK_DESC,3,7,_XMP_STARPU_HOST_DESC_px,_
          ...);
59    _XMP_starpu_calc_iter(_XMP_STARPU_TASK_DESC,_XMP_loop_init_i,_XMP_loop_cond_i,
          _XMP_loop_step_i);
60    _XMP_starpu_pack_cl_arg(&cl_arg,&cl_arg_size,_XMP_STARPU_TASK_DESC,_XMP_DESC,
          _XMP_STARPU_DEVICE_DESC_px, ...);
61    (_XMP_gpu_func_pointer)=(reinterpret_cast<void *>(_XMP_STARPU_FUNC_0_GPU));
62    (_XMP_cpu_func_pointer)=(reinterpret_cast<void *>(_XMP_STARPU_FUNC_0_CPU));
63    _XMP_starpu_call_calc_func(_XMP_gpu_func_pointer,_XMP_cpu_func_pointer,cl_arg,cl_arg_size,
          _XMP_STARPU_TASK_DESC,7,_XMP_STARPU_HOST_DESC_px, ...);
64  }
```

4.2.4　XMP-dev/StarPU のランタイム

本節は，XMP-dev/StarPU において StarPU を制御するためのランタイムの実装について述べる．XMP-dev/StarPU コンパイラは，指示文を対応するランタイムに置き換える．_XMP_starpu から始まる関数が，本研究で実装を行ったランタイム関数である．まず，replicate 指示文は Replicated array の確保と StarPU のデータプールへの登録を行う．リスト 4.3 中の_XMP_starpu_init は，タスクディスクリプタの構造体の確保から初期化までを行う．_XMP_starpu_init_data_ALIGNED 関数は Replicated array の確保，StarPU のデータプールへの登録および_XMP_STARPU_HOST_DESC の初期化を行う．そして replicate 指示文のスコープの終わりとともに解放される．

replicate_sync 指示文は，Local array と Replicated array 間の同期を行うための指示文である．リスト 4.4 の_XMP_starpu_sync 関数では Local array と Replicate array 間で memcpy が実行される．sync_clause が in の時には Local array から Replicated array へ，out の時にはその逆である．さらに，sync_clause が in の時には Replicated array を複数のタスクに分割する．これによって，細分化したタスクを各計算リソースに割り当てることが可能になる．out の時には，データの集約を行い 1 つの配列に

戻す．

最後に，device loop 指示文のランタイムシステムについて述べる．リスト 4.5 において，_XMP_STARPU_FUNC_0 関数内で_XMP_starpu_get_task_desc により，初期化されたタスクディスクリプタからタスク数などの情報を各配列の HOST_DESC に与える．そして，for 文のイテレーションの範囲を_XMP_starpu_calc_iter で計算し，_XMP_starpu_pack_cl_arg でそれらの変数をパッキングする．リスト 4.5 中の 61，62 行目は関数ポインタに GPU/CPU それぞれで実行される関数を代入し，_XMP_starpu_call_calc_func に与える．この関数内でタスクや codelet の生成からタスクの発行，同期までを行う．

次に StarPU が呼び出した計算関数について述べる．リスト 4.5 中の 1〜19 行目が CPU での実行関数，34〜48 行目が GPU の実行関数である．これらの関数内の前半部分は共通となっている．_XMP_starpu_double_vector_ptr 関数では，StarPU のデータプールに登録されている配列データを受け取る．そして，タスクが発行される前にパッキングされたデータを_XMP_starpu_unpack_cl_arg 関数でアンパックする．次に_XMP_starpu_set_iter 関数でタスクディスクリプタ内のイテレーション情報を取り出す．CPU の関数ではそのまま for 文が挿入され，イテレーションのみ変更される．GPU では直接この関数内で実行をせず，カーネル関数（_XMP_STARPU_FUNC_0_DEVICE）の呼び出しを行う．カーネル関数に関しては XMP-dev が生成するコードを利用している．このようにして，XMP-dev/StarPU では GPU と CPU によるワークシェアリングを実現している．

4.3　タスクサイズとロードバランス

StarPU では，データプールに登録した配列を細かい単位に分割する．これを StarPU ではタスクと呼んでおり，スケジューラが GPU や CPU へスケジューリングを行う．しかし，GPU と CPU が混在するヘテロジニアスな環境では先ほど述べたタスクサイズとスケジューリングの自由度の問題が出てくる．CPU コアあたりの演算性能はたかだか十数 GFLOPS であるのに対して，GPU は NVIDIA の Tesla M2090 において倍精度浮動小数点演算性能は 665GFLOPS に達し，Fermi アーキテクチャの次の Kepler アーキテクチャの Tesla K20X においては 1.31TFLOPS と，1GPU で 1TFLOPS 以上を達成している．このように，理論ピーク性能比ではあるが，演算性能に数十倍近くの差があるにも関わらず，タスクに割り当てる問題サイズを同じにしてしまうと 1 タスクあたりの実行時間が，GPU と CPU で大きく変わってくるのは明らかである．StarPU は，大量のタスクを生成し，それを複数の計算リソースにダイナミックにスケジューリングすることで，演算性能が高い計算リソースにタスクが多く割り振られることを期待している．非常に大規模な問題では，文献 [40] でも述べているようにスケジューリングがうまく動作するが，小規模・中規模であると難しくなる．問題サイズを固定したままタスクサイズを大きくするとタスクの個数が減る．そのため，スケジューリングの自由度は小さくなり，うまく負荷バランスを取ることができない．一方，タスクサイズを小さくするとタスクの個数が多くなり，スケジューリングの自由度が上がる．しかし，GPU では小規模のホスト〜デバイス間の通信が多発してしまい，その結果，GPU の演算性能が生かしきれなくなってしまう．そのため，デバイスごとに適切なタスクサイズを設定する必要がある．

4.4　適応型負荷分散機能

タスクサイズの調整には，GPU と CPU の性能差が大きく影響する．GPU/CPU ワークシェアリングでは，GPU と CPU のタスク当たりの実行時間を最小にすることができれば，総タスク数が少ない場合であっても効率的に計算リソースを利用することができる．例えば，GPU に割り当てるタスクサイズを大きくすることで，ホスト〜デバイス間の転送オーバヘッドが相対的に小さくなり，GPU が十分な性能を出すことが可能になる．一方，CPU に大きなタスクサイズを割り当ててしまうと，実行時間が大きくなりすぎてしまう可能性がある．このように，この両デバイスにおける最適なタスクサイズの差は，問題の性質・問題サイズ，実行環境によって大きく異なるため，従来の固定的な配分比では十分な最適化を行うことは難しい．

この問題を解決するために，CPU core と GPU がそれぞれ担当するタスクサイズの比を示す “CPU Weight” というユーザが制御可能なパラメータを設け，loop 文のワークシェアリングにおいて GPU と CPU 間のタスクサイズのバランスを調整することを提案する．これによって，従来の XMP-dev/StarPU のフレームワークの中で簡潔に適応型負荷分散を記述することが期待できる．タスクサイズのバランスを取るために，まず XMP-dev/StarPU のランタイムで生成した Replicated array を，GPU で計算する領域と CPU で計算する領域に分割する．図 4.3 に，Replicated array の分割のイメージを示す．この配列は図 4.2 の Replicated array である．図 4.3 の青い部分が CPU，赤い部分が GPU の計算領域である．ここでは，要素数 N の配列を 1 次元分割しており，CPU Weight を用いてそれぞれの領域の割合を調整できる．CPU Weight は，CPU が計算する領域の割合を示す 0 から 1 の範囲の浮動小数点数であり，0 に近くなるほど CPU の計算量が減少する．そして，2 つの領域はそれぞれ StarPU によっていくつかの小さなタスクに分割され，各リソースに割り当てが行われる．図 4.3 の例では，それぞれの領域が 3 つに分割され，CPU には小さなタスクサイズを，GPU にはより大きなタスクサイズが割り当てられている．このようにして，CPU Weight を用いることで GPU と CPU 間のロードバランスを調整することが可能になる．XMP-dev/StarPU では，それぞれの配列領域における分割数は環境変数 “XMP_STARPU_NCPU” および “XMP_STARPU_NCUDA” によって設定される．これにより，両リソースにおいて実行される 1 タスクのサイズを指定することが可能であり，ユーザは実行時にこれらの設定を行うことができる．

しかしながら，最適なタスクサイズの決定と同様に，最適な CPU Weight を静的に設定することは困難である．つまり，コンパイラやランタイム内部で自動的に CPU Weight を決定し，それによって性能向上を得ることは難しいと考えている．最大の理由は，StarPU におけるタスクへのデータの割り当ては，基本的にデータプールに管理されている配列を単位としているため，タスクの実際の計算量がその配列サイズに対してどの程度のオーダーになっているかはアプリケーションに依存するからである．また，デバイスメモリへのデータ移動のオーバヘッド等も勘定すると，全体のタスク実行コストをデータのサイズだけから見積もることは極めて難しくなる．そこで本研究では，CPU Weight については適当な初期値を与えるが，計算の途中でこの値にユーザが任意に変更可能とするフレームワークを提供する．すなわち，GPU と CPU に対するタスクサイズの割り当ては，ユーザによって動的に与える事ができるものとする．これを実現するために，CPU Weight をプログラムの実行中に動的に再定義する機能として，

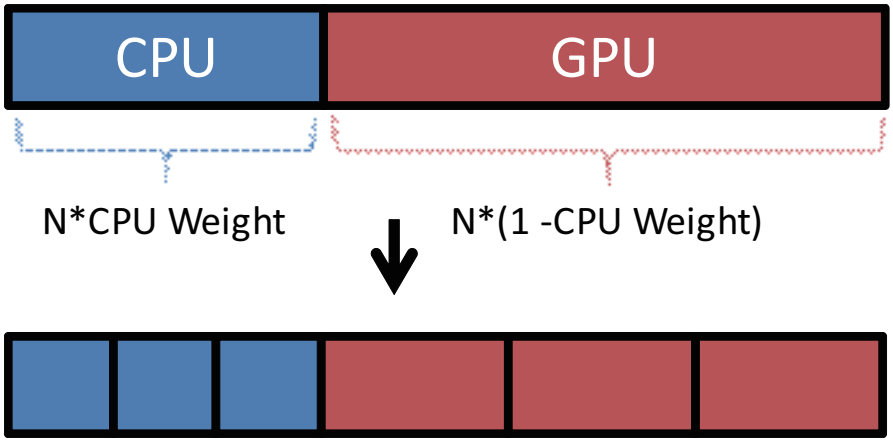


図 4.3　Replicated array の分割イメージ例

“reset_weight” 指示文を導入する．指示文は次のとおりである．

```
double cpu_weight;
#pragma xmp device reset_weight (cpu_weight)
```

ユーザは reset_weight 指示文を，ワークシェアリングを行う loop 文の直前に挿入することで，プログラムの実行特性に応じて自由に CPU Weight を変更できる．なお，reset_weight 指示文のようにアクションを伴う指示文は，API 関数として定義することも可能であるが，あくまで XMP-dev/StarPU 言語の中で閉じた形としたいため，本研究では指示文形式を取る．

しかし，図 4.3 で示しているように，Replicated array は 1 次元分割されているため，先述のようにタスクサイズを実行時間から見積もることは難しい．例えば，行列・行列積では行列サイズ N に対して，計算量は（問題の並列分割手法にも依存するが，最大で）$O(N^3)$ になり，CPU Weight を変更することで実行時間が大きく変わりすぎてしまう可能性がある．このような問題に対しては，プログラム実行中に適宜実行時間を測定し，徐々に CPU Weight を最適な値に近づけていく手法が有効であると考えられる．実際，時間発展を行うシミュレーションでは，タイムステップごとに CPU Weight を変更することで，ある程度の試行ステップを経た上でこれを最適化することが可能である．

リスト 4.6　reset_weight 指示文の利用例

```
1  for (int t = 0; t < TIME_STEP; t++) {
2  #pragma xmp device loop on t(i)
3    for (int i = 0; i < N; i++) {
4      // GPU/でワークシェアリング実行されるループ計算の本体 CPU
5    }
6
7    double cpu_time = xmp_cpu_time();
8    double gpu_time = xmp_gpu_time();
9
10   double cpu_ratio = cpu_time / (cpu_time + gpu_time) * 100;
```

```
11
12    if (cpu_ratio > 51)
13      new_cpu_time -= 0.01;
14    if (cpu_ratio < 49)
15      new_cpu_time += 0.01;
16
17  #pragma xmp device reset_weight(new_cpu_time)
18  }
```

リスト 4.6 に reset_weight 指示文の利用例を示す．ワークシェアリングを行う loop 文中で，イテレーションが終了したら次のタイムステップで使う新たな CPU Weight を計算している．ユーザによる負荷分散を容易にするための補助関数として，XMP-dev/StarPU は，GPU と CPU に割り当てられたタスクの実行時間を得る関数 xmp_gpu_wtime() および xmp_cpu_wtime() を提供している．ここでは，これらを用いて CPU の実行時間がトータルの計算時間の何 % を占めているかを計算する（cpu_ratio）．そして，その割合が 50% に近づくように，つまり GPU と CPU の実行時間がなるべく等しくなるように CPU Weight を調整し，計算によって新しい CPU Weight を次のステップで利用するために reset_weight 指示文で変更を行う．リスト 4.6 では，非常に単純なアルゴリズムで CPU Weight を決定しているが，ユーザが最適なアルゴリズムを用いることでより早い収束を得ることも可能である．例えば，1 回ごとの調整幅を最初は大きくし，逆転してしまったらその半分にして微調整する等のアルゴリズムを用いることが考えられる．また，ワークシェアリングを行う loop 文が，大きな時間発展ループの中に複数ある時，loop 毎に GPU と CPU の実行時間が異なる場合がある．reset_weight 指示文をこまめに用いることで，loop 毎に最適な CPU Weight を設定することができ，全体の性能向上を得ることが期待できる．荒木らの研究 [20] では，複数のプロセッサ間で負荷分散の調整を行うために，時間発展ループ内でプロセッサに割り当てるデータサイズを変化させている．ヒューリスティックな探索手法として，各プロセッサの実行時間と配列サイズから比を求め，それにしたがってデータの再配置を行うことで負荷分散を実現している．例えば，文献 [20] の 3.5 節より，4 つのプロセッサの実行時間がそれぞれ 10, 40, 40, 40 (sec) で，割り当てられた配列サイズが 250, 250, 250, 250 の時，配列サイズを実行時間で割った速度比 25:6.25:6.25:6.25 が得られ，再配置されたデータはそれぞれ 572, 143, 143, 142 となる．このように，単に各演算機の実行時間を近づけるだけではなく，データサイズを含めた調整アルゴリズムを本フレームワーク上で利用することが原理的には可能である．

このような負荷分散制御のためのコードは本来ユーザプログラムには本質的に含まれない部分であり，ユーザ自身が記述すべきでないという考えもある．しかし，先述したように，タスクに割り当てられる配列サイズだけからタスク実行時間を予測することは難しく，何らかの実行時プロファイリングによる調節が最も有効であると考えられる．ユーザ負担を軽減するため，例えばリスト 4.6 のように比較的単純なアニーリング手法をユーティリティ関数として提供する事も考えられる．あるいは，これを標準的な手続きとしてランタイム内に閉じ込め，このような標準的なアニーリングを行うことをデフォルト機能とし，ユーザがより積極的な適応型負荷分散制御を行いたい場合は直接アルゴリズムを記述するという形の実装にすることも可能である．

表 4.1　評価環境（HA-PACS Base Cluster）

CPU	Intel Xeon E5-2670 2.6GHz
GPU	NVIDIA Tesla M2090
Main memory	DDR3 1600MHz 128GB
GPU memory	DDR5 6GB / GPU
Interconnection	InfiniBand QDR (2 rails)
OS	CentOS release 6.1 (Final)
CPU compiler	gcc 4.4.5
GPU compiler	CUDA 4.2
MPI	MVAPICH2 1.8.1
# of CPU/node	16 cores (8 cores × 2 sockets)
# of GPU/node	4

4.5　性能評価

本節では，XMP-dev/StarPU の生産性とベンチマークによる評価を行う．評価には，筑波大学計算科学研究センターで稼働中の GPU クラスタである HA-PACS ベースクラスタ [45, 46] を用いる．評価環境を表 4.1 に示す．本評価では 268 台の計算ノード中の 2 から 16 ノードを用いる．

StarPU は，GPU の通信やカーネル関数の起動などを管理するために，1GPU につき 1CPU core を管理スレッドとして使用する．そのため，ノード内で 4GPU を計算に用いる場合，計算に参加する CPU core は $16 - 4 = 12$ となる．評価に用いるベンチマークは，N 体問題と行列・行列積（以降，単に「行列積」）であり，ともに倍精度浮動小数点演算を行う．

4.5.1　生産性の評価

XMP-dev/StarPU の生産性について評価を行う．複数ノードによる GPU と CPU のワークシェアリングを行うためのプログラムは，MPI と StarPU を組み合わせて記述することで実現できるが，一方で非常にプログラミングコストが高い．StarPU を用いたプログラムは，データの登録や分割，タスクの生成・発行を StarPU の API にそって記述しなければならない．これは，MPI と CUDA を駆使してプログラミングを行うよりもプログラミングコストは小さいが，依然としてプログラミングコストは高いままである．そこで，XMP-dev/StarPU を用いることでプログラミングのコストがどれほど減少するかを確認する．生産性の評価には，N 体問題のプログラムを用い，逐次，MPI+StarPU，XMP-dev/StarPU による記述を行い，SLOC（Source Lines Of Code）を比較する．この SOLC は空白行を含まない．

表 4.2 に N 体問題のソースコードの行数を示す．MPI，StarPU，XMP は，それぞれの指示文や API の行数を示している．例えば，MPI は，通信するための API だけではなく，データ分割に伴うインデックスの計算も含まれている．計算部分は，メインのカーネル部分であり，それ以外の配列の宣言，初期

表 4.2　N 体問題：ソースコードの行数

	MPI	StarPU	XMP	初期化部分	計算部分	合計
逐次				27	31	58
Hand-coding (MPI + StarPU)	11	114		29	72	226
XMP-dev/StarPU			18	27	31	76

化，マクロの宣言などは初期化部分に含まれている．N 体問題では，各ノードで計算した粒子の位置情報（*px, py, pz*）を Allgather で交換するというシンプルな通信のみであるため，コード行数に対する MPI の割合は 4.87% と非常に小さい．しかし，ノード内のワークシェアリングを行うための StarPU による記述の割合は 50.44% と非常に大きな割合を占めている．これは，4.1.4 節で示したように，StarPU のデータプールへの配列の登録や分割，codelet の作成，タスクの発行などを明示しなければならないためである．さらに，CPU 用の実行関数と GPU 用の実行関数を分けて記述する必要があり，表 4.2 において，計算部分の行数が倍以上に増えた要因となっている．その結果，逐次のプログラムに対して Hand-coding のプログラムは 3.89 倍に記述量が増えている．一方，XMP-dev/StarPU は逐次のプログラムに XMP-dev/StarPU の指示文を 18 行加えるだけで，Hand-coding と同等のコードを出力することが可能である．これは，Hand-coding のプログラムに対して XMP-dev/StarPU のプログラムは，33.62% の行数で記述が可能である．XMP-dev/StarPU による記述は，StarPU の API が隠蔽していることや，各デバイスに計算用の関数を記述せず，1 つの関数で済むため，プログラムを簡潔に記述することが可能である．また，逐次のプログラムの行数に対して，1.31 倍の増加にとどまる．以上のことから，XMP-dev/StarPU は Hand-coding による記述に対して生産性があると言える．

4.5.2　適応型負荷分散機能の評価

XMP-dev/StarPU に導入した，適応型負荷分散機能の評価を行う．

まず，適応型負荷分散機能による reset_weight 指示文のコストについて評価を行う．reset_weight 指示文は，CPU と GPU に割り当てるデータサイズの割合を変更する指示文である．図 4.4 に，2 ノードの N 体問題の粒子数 819200 における各 CPU Weight の実行時間を示す．使用するノードごとの GPU 数，CPU core 数をそれぞれ 4，12 とする．これより，CPU Weight が増加するに連れて，reset_weight 指示文の実行時間は減少していることがわかる．reset_weight 指示文の内部では，StarPU の API を用いて一度，GPU のデバイスメモリに分散されたデータをホストメモリ上に書き戻し，CPU Weight に応じたデータの再配置を行っている．そのため，書き戻しのためにホスト〜デバイス間の通信が発生し，CPU Weight が小さい，つまり，GPU に割り当てられたデータサイズが大きい時には，通信時間が大きくなり reset_weight 指示文の実行時間が増加する．一方で，CPU Weight が増加すると，GPU に割り当てるデータサイズは減少するため，ホスト〜デバイス間の通信量が減少し reset_weight 指示文の実行時間も減少する．また，CPU Weight == 1.0，つまり，すべてのデータがホストメモリ上にある場合は，デバイス〜ホストメモリの通信が発生しないため，これが reset_weight 指示文の最小コストであると言える．CPU Weight == 0.0，つまり，すべてのデータがデバイスメモリ上にある場合，最も実行時間が長

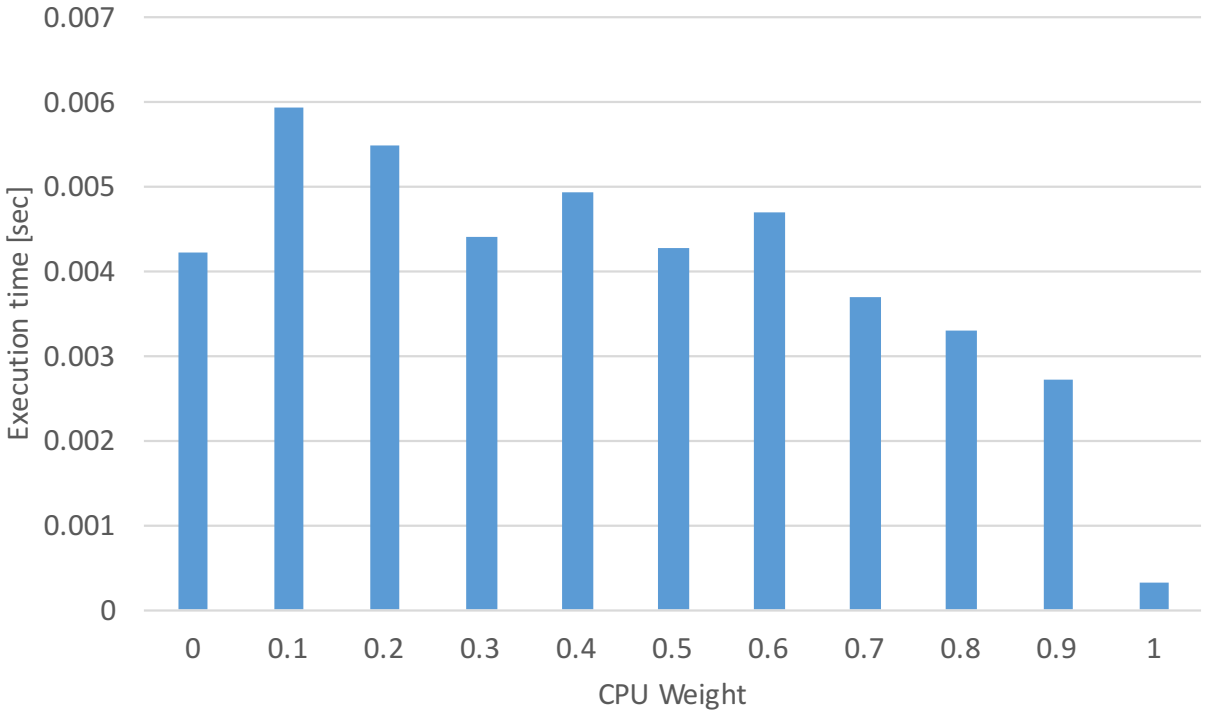


図 4.4　粒子数 819200 における reset_weight の実行時間

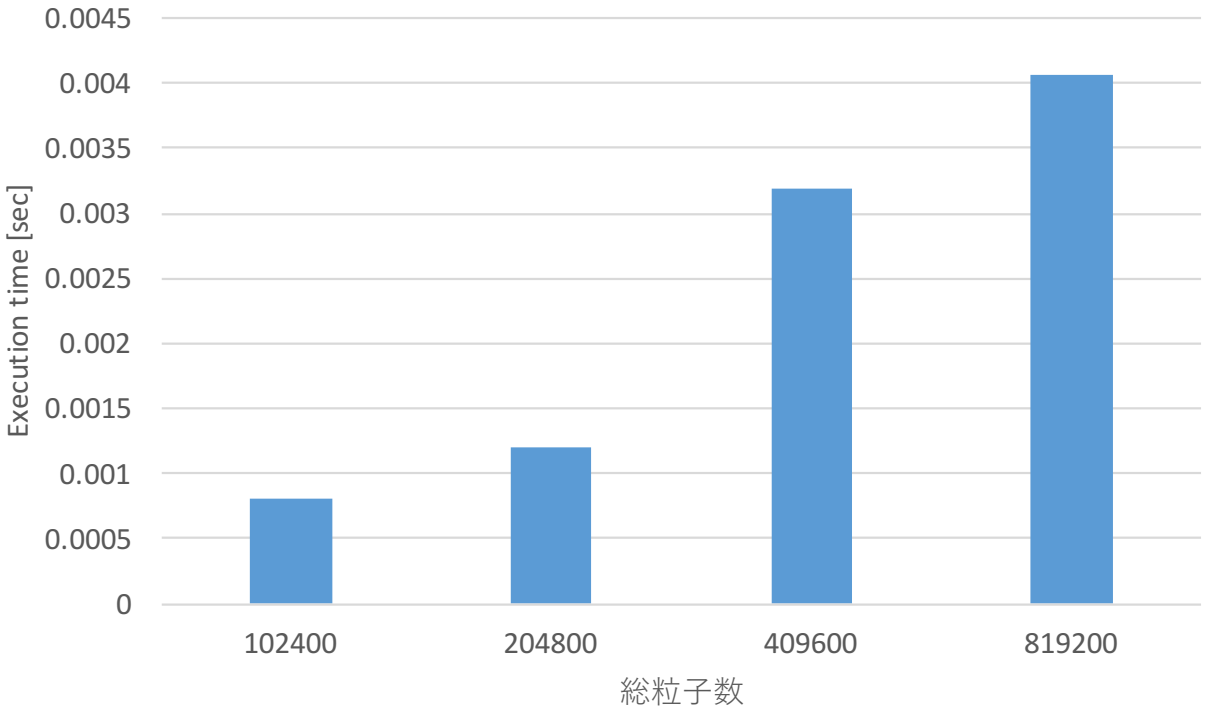


図 4.5　粒子数による reset_weight の実行時間の変化

くなると想定していたが，CPU Weight == 0.1 のほうが大きいという結果になった．この原因として，StarPU の API 内部で，ホストメモリとデバイスメモリに配置されたデータを判別する必要がないことや，デバイスメモリに確保されたデータのアライメントによってホスト〜デバイス間通信が最適化されたと考えられる．これについては，StarPU の API を解析する必要があると考えている．さらに，データサイズと reset_weight 指示文の関係を評価するために，図 4.5 に CPU Weight==0.3 とした時の各データサイズの実行時間を示す．図 4.4 より，問題サイズである粒子数に比例して reset_weight 指示文の実行時間が増加していることがわかる．図 4.4，図 4.4 から，データサイズが大きい，特に GPU に割り当てられたデータサイズが大きいと reset_weight 指示文の実行時間が増加するということがわかる．一方で，データサイズが大きくなるということは，計算の実行時間も増加することが一般的であり，reset_weight 指示文の実行時間のオーダーでは，全体の実行時間にほとんど影響しない．つまり，適応型負荷分散機能をプログラム中に導入しても，reset_weight 指示文のコストが全実行時間に占める割合はほとんど増加しないと言える．

次に，適応型負荷分散機能よる CPU Wegiht の最適化についての評価を行う．CPU と GPU のタスクの実行時間に応じて，CPU Weight をプログラム中の時間発展ループ内で変更する．本評価では，HA-PACS ベースクラスタの 2 ノードを用い，各ノードの GPU 数を 4，CPU core 数を 12 とする．また，ノードあたりのタスク数を 16 とすることで，各リソースには 1 つだけタスクを割り当てる．すなわち，Replicated array の GPU 担当部分を 4 分割，CPU 担当部分を 12 分割している．CPU Weight の初期値は 0.2 とし，これは図 4.3 では CPU : GPU = 1 : 4 に相当する．CPU Weight の変更についてはリスト 4.6 に示したアルゴリズムに則っている．この環境で，最外の時間発展ループを一定回数実行し，CPU Weight の変化と共に GPU と CPU の実行時間が近づいて負荷分散が行われ，性能が最適化されることを確認する．なお，本評価は GPU と CPU の負荷バランスについて着目し，実行時間の測定には MPI 通信の時間を含めていない．N 体問題では，最外の時間発展の loop 内で CPU Weight を変更する．行列積では，最外にタイムステップの loop を作り，その中で行列積を何度も繰り返し，1 イテレーションの実行時間によって CPU Weight を決定する（例えば HPL ベンチマーク中の BLAS DGEMM ルーチンを何度も呼び出すイメージ）．また，行列積では GPU に MAGMA BLAS[25]，CPU に Gotoblas[47] を用いることで，小行列の計算を高速化している．本評価の行列積では，行方向に 1 次元分割をしているため（すなわち，部分行列サイズは行列の 1 つの方向のみを表し，もう一方は常に N で固定である），$C = A \cdot B$ のような演算において，行列 B はすべての要素を GPU に転送する必要があり，GPU のデバイスメモリサイズの制約から行列サイズ 16384×16384 がこの環境で実行できる最大サイズとなっている．現在，XMP-dev/StarPU コンパイラおよびランタイムシステム自体は二次元分割をサポートしているが，配列をタスクに二次元分割した際にデータが不連続となってしまう．しかし，BLAS 自体は引数の配列内のデータが連続領域に格納されていることを想定しているため，結果として二次元分割実装では BLAS を有効に利用できない．そのため本評価では，1 次元分割で確実に配列が連続領域であるようにして測定を行っている．今後は，GPU が十分な性能を出せる問題サイズで評価を行うために，これらの問題を解決していく予定である．

図 4.6 および図 4.7 はそれぞれ，N 体問題と行列積のタイムステップ毎の GPU と CPU の実行時間および CPU Weight の推移を示している．図 4.6，図 4.7 中の青いバーが CPU，赤いバーが GPU の，それぞれ最も遅いタスクの実行時間を表している．そして緑色の折れ線は CPU Weight を示している．左

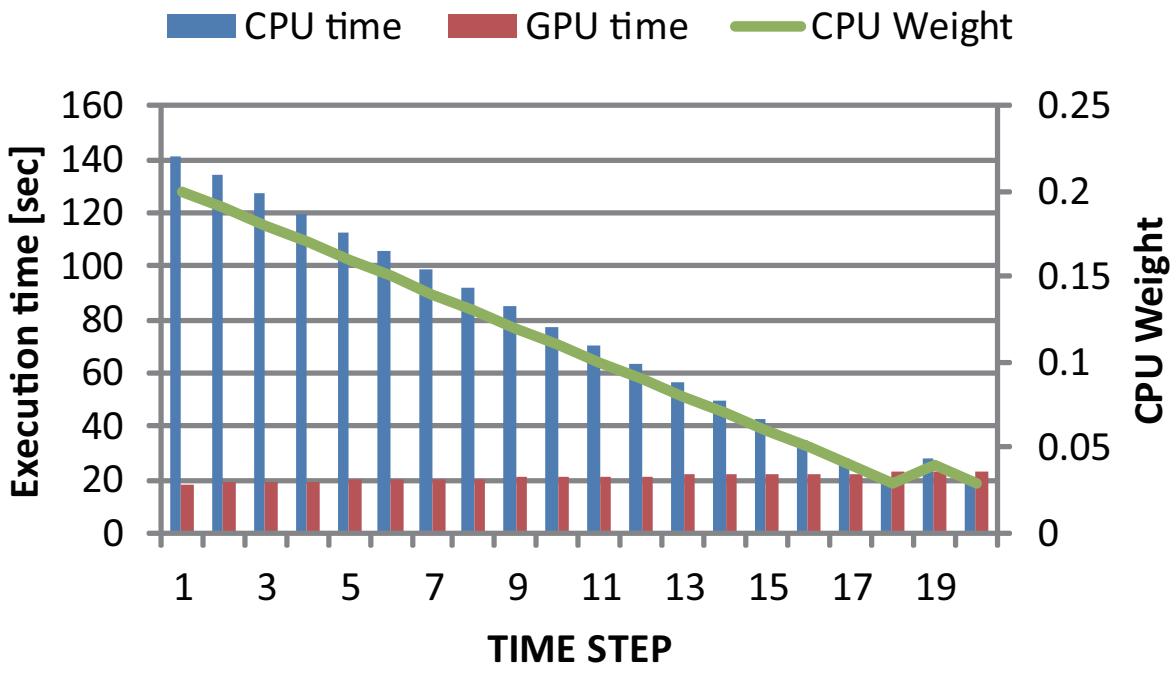


図 4.6　CPU Weight の推移：N 体問題（粒子数 $N = 819200$）

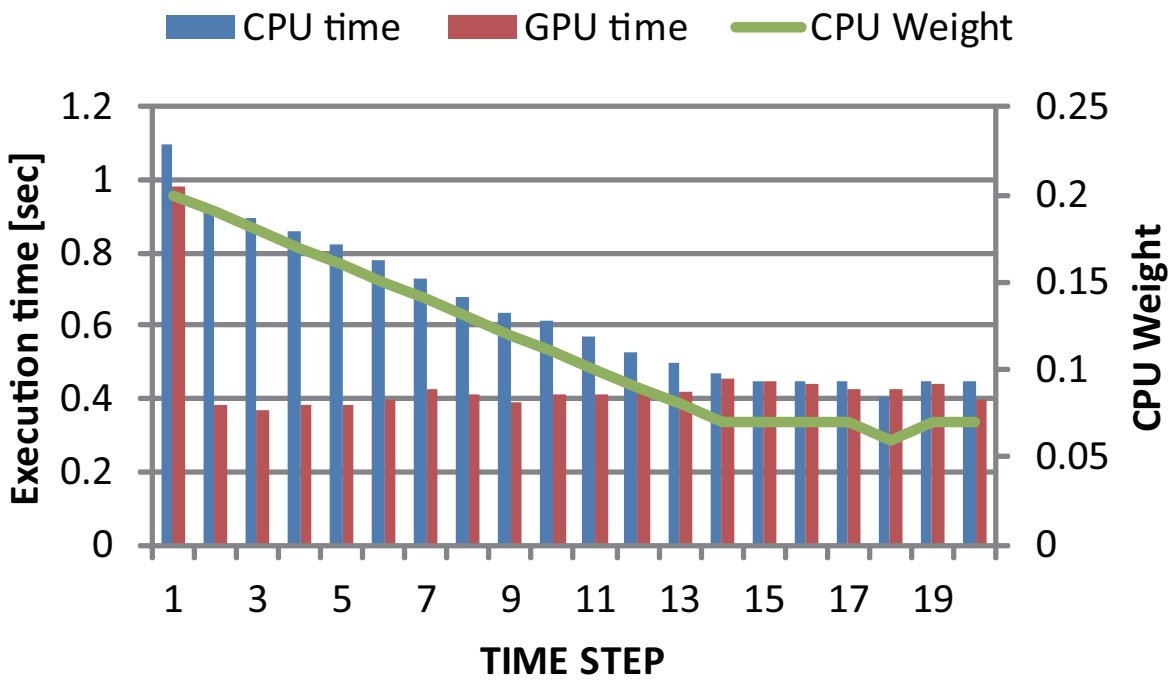


図 4.7　CPU Weight の推移：行列積（行列サイズ 8192×8192）

の縦軸はタスクの実行時間，右の縦軸は CPU Weight，横軸はタイムステップである．両方のグラフにおいて，GPU の演算性能は CPU core 12 個に対して非常に大きいため，CPU Weight が大きい時に CPU がボトルネックになっていることがわかる．図 4.7 より，1 ステップ目の GPU の実行時間が非常に大きいことが観測される．これは以下の理由による．1 次元分割による行列積では，配列 B についてすべてのデータを持っている．また，この配列は読み出し参照のみが行われるため，1 タイムステップ目でデータをホストからデバイスへ転送する．行列積の計算自体は非常に短時間で終わるため，1 ステップ目の実行時間が大きく見える．リスト 4.6 のアルゴリズムに則って CPU Weight を変更していくと，GPU と CPU の実行時間が徐々に均衡していくことがわかる．それに伴って CPU Weight の値も収束し，最終的には両実行時間がほぼ均等になっている．N 体問題では，18 ステップ目，行列積では 14 ステップ目あたりで GPU と CPU の実行時間がほぼ均衡し，CPU Weight が収束していることがわかる．そして，その段階で全体の実行時間が最短になっていることが示されている．

しかしながら，問題や問題サイズ，実行環境によっては必ずしも GPU と CPU の実行時間が近づくとは限らない．そこで，問題サイズや計算リソース数を変更する．図 4.8 および図 4.9 はそれぞれ，2 ノードを利用した時の N 体問題と行列積の結果を示す．図 4.8 では，N 体問題における粒子数を 1024 とし，計算に利用するリソースは図 4.6 と同様である．問題サイズが小さい時，GPU は演算だけでなく，カーネル関数の起動コストが必要となるため，CPU と比較して実行時間が大きくなってしまう．図 4.8 では，CPU Weight を 0.9 に設定しているが，GPU の実行時間は CPU と比べて倍以上大きい事がわかる．CPU Weight を変更しても，カーネル関数の起動にかかるオーバヘッドは変わらないため実行時間の差は変化しない．そのため，このような問題サイズにおいては CPU のみを使う，つまり，CPU Weight が 1.0 が最良値となり，図 4.8 からも同様の結果が得られた．図 4.9 では，行列サイズを 2048×2048 とし，演算に利用する CPU コア数を 4 に制限している．CPU の演算性能よりも GPU の演算性能が非常に高い場合，図 4.9 の結果からも，CPU の演算時間が大半を占めている．そのため，CPU Weight は徐々に減少していき 0.01 の場合においても GPU の実行時間より短くなることはない．よって，CPU Weight は 0.0 つまり GPU のみを演算に利用したほうが良いという結果が得られた．

今回の実験では非常に単純な調整アルゴリズムで CPU Weight の自動調整を確認したが，問題によってはより高速な収束が求められる場合があり，CPU Weight の調整アルゴリズムを変更する必要がある可能性がある．例えば，HPL 内の DGEMM BLAS ルーチンに適応型負荷分散を用いる場合を考える．大島らの研究 [7] や遠藤らの研究 [8, 9] では，DGEMM の対象となる行列のサイズによって CPU のみを演算に使うか，ワークシェアリングを使うかという選択を行うことで，単にすべての DGEMM にワークシェアリングを適用する場合よりも性能を向上させている．本フレームワークでは，行列のサイズのしきい値を設定することで，小さい行列サイズに対しては CPU のみ（CPU Weight == 0.0），大きい行列サイズに対しては適応型負荷分散による最適な CPU Weight の調節を行うことができる．さらに，各演算器の実行時間を比較することで，行列サイズのしきい値に対するヒューリスティックな探索も，ユーザがアルゴリムを記述することで実行することができる．DGEMM BLAS ルーチンは，非常に多くのアプリケーションで用いられているため，様々なアプリケーションに本フレームワークを適用することが可能であると考えている．このようにして，本研究で実装した機能により，柔軟なアルゴリズムの設定や GPU と CPU 間のロードバランスを最適化を，高水準の PGAS プログラミング言語で簡潔に記述するような適応型負荷分散が実現できることを確認した．

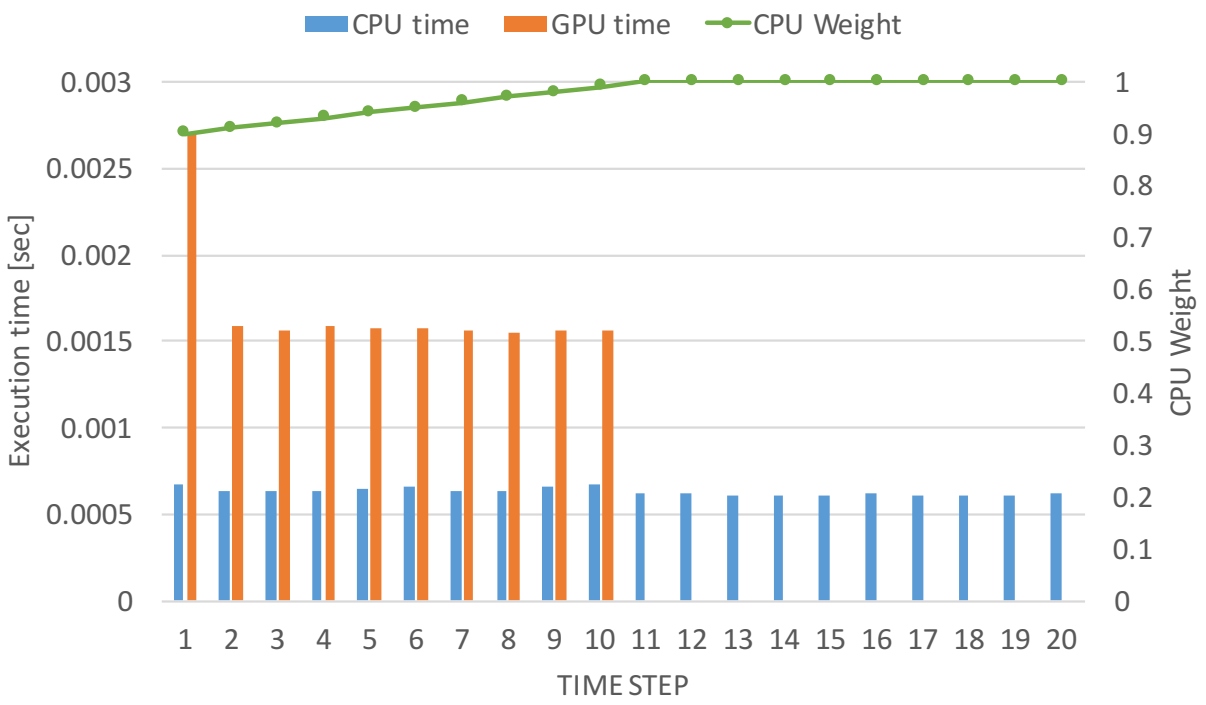


図 4.8　CPU Weight の推移：N 体問題（粒子数 $N = 1024$）

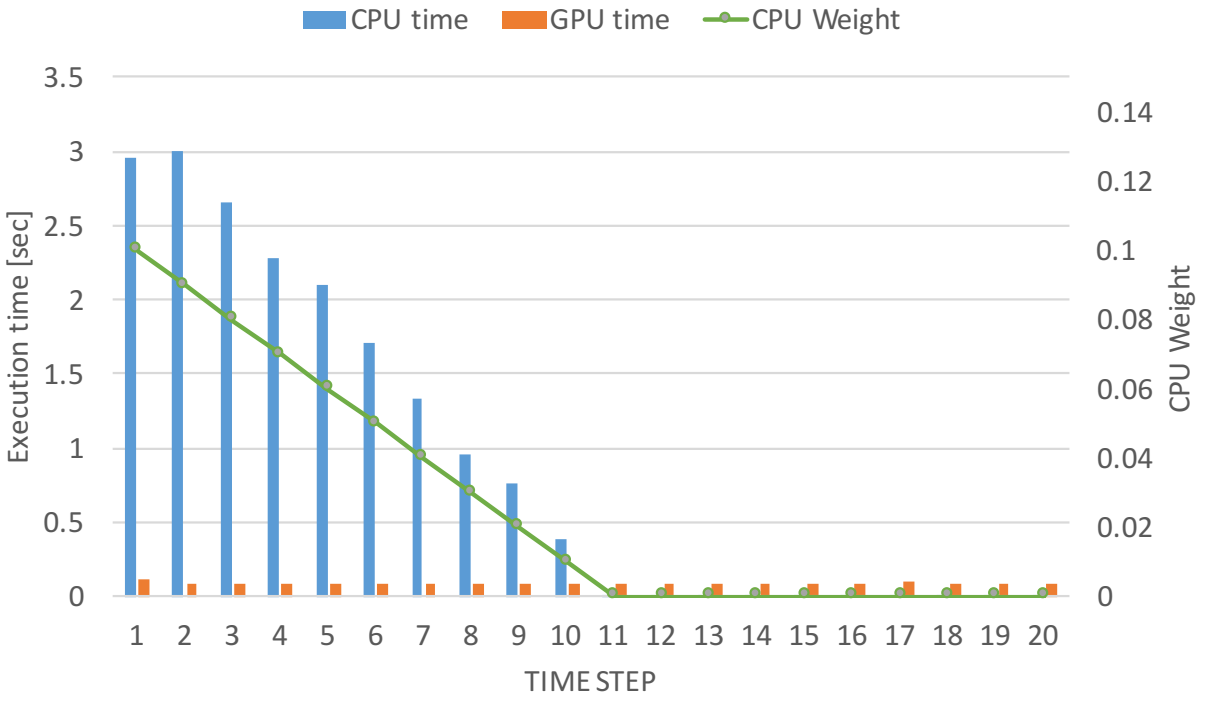


図 4.9　CPU Weight の推移：行列積（行列サイズ 2048×2048）

次に，表 4.3, 表 4.4 に，HA-PACS の 2 ノードを用いて，ノード内の GPU 数を変化せた時の 20 STEP 目の実行時間（ $max(cpu\_time, gpu\_time)$ ）とその時の CPU Weight を示す．

表 4.3　N 体問題：20 STEP 目の実行時間（N：粒子数）

	1GPU		2GPU		4GPU	
N	time [sec]	CPU Weight	time [sec]	CPU Weight	time [sec]	CPU Weight
102400	1.352	0.15	0.710	0.07	0.435	0.04
204800	5.058	0.14	2.797	0.07	1.679	0.03
409600	20.035	0.14	11.052	0.07	6.014	0.03
819200	78.623	0.14	42.613	0.07	23.018	0.03

表 4.3 の $N = 102400$ では，1GPU を使うときには CPU は全体の計算の 15% を担当しているのに対し，4GPU の場合には 3% まで減っている．ノード内の GPU が増えるにつれ GPU のタスクの計算時間は減り，それによって CPU の計算時間も短くなる必要があり CPU Weight も小さくなっている．1GPU から 4GPU にした時に単純に CPU Weight が 1/4 にならないのは，（StarPU の制約により）同時に CPU core 数が 15 から 12 に減り，CPU の計算リソースが減ってしまうためだと考えられる．また，表 4.3 では各タスクの計算量が十分にあるため GPU 数の増加によっておおよそ実行時間が半減していく事がわかる．

表 4.4　行列積：20 STEP 目の実行時間（N：行列サイズ）

	1GPU		2GPU		4GPU	
N	time [sec]	CPU Weight	time [sec]	CPU Weight	time [sec]	CPU Weight
1024	0.005	0.29	0.005	0.21	0.004	0.19
2048	0.023	0.27	0.017	0.09	0.011	0.02
4096	0.164	0.28	0.116	0.13	0.061	0.05
8192	0.184	0.29	0.691	0.15	0.445	0.07
16384	8.276	0.29	5.167	0.16	3.675	0.09

しかし，表 4.4 の $N = 1024$ のような場合では，問題サイズが小さいためオーバヘッドの少ない CPU の計算割合が増えている．行列積では，GPU・CPU ともに BLAS を用いており，各計算リソースの性能が出やすく，かつ問題サイズが小さいため CPU の割合が増えたと考えられる．$N = 16384$ では十分な計算量があるため N 体問題同様に表 4.3 のような変化をしていることがわかる．

4.5.3　N 体問題の評価

最後に，GPU と CPU のワークシェアリング性能について調べる．本評価では，GPU のみを計算に利用した結果に対して，それに CPU を加えた場合の速度向上を調べる．N 体問題および行列積における，収束した時の 1 ステップの実行時間により，それぞれのノード数及び GPU 数を使用した XMP-dev/CUDA

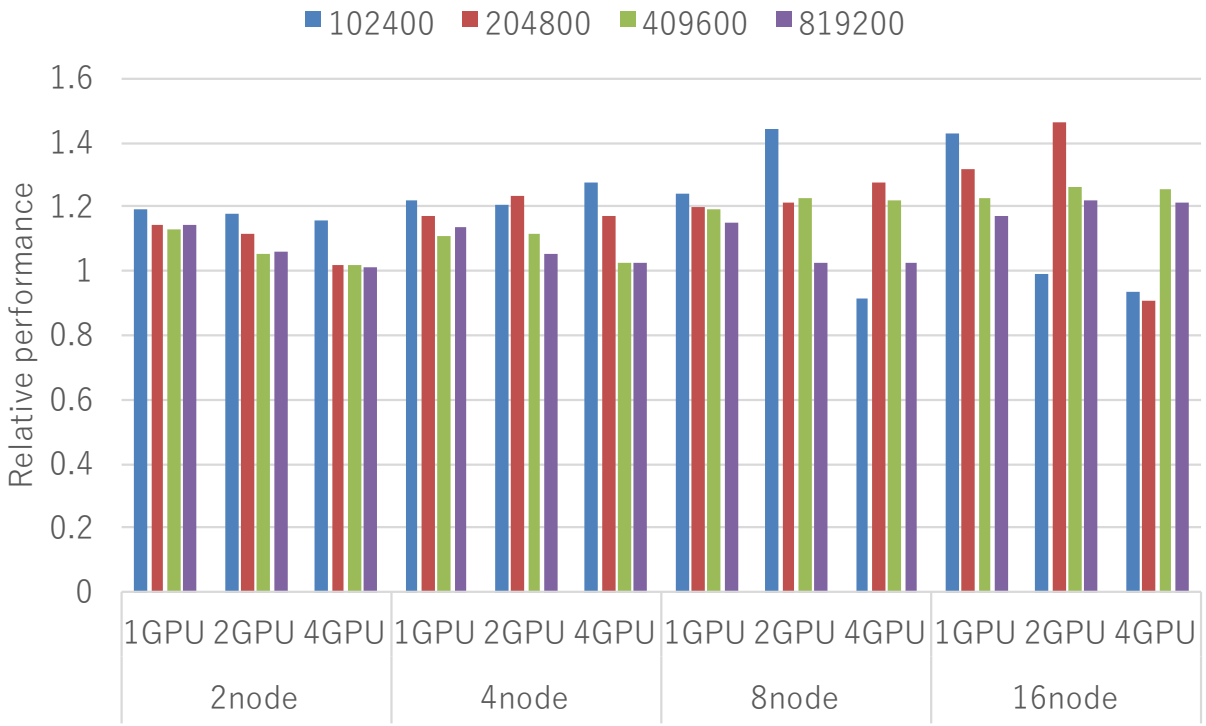


図 4.10　N 体問題：XMP-dev/CUDA に対する相対性能

表 4.5　N 体問題：1GPU に割り当てられた粒子数

	2node			4node		
総粒子数	1GPU	2GPU	4GPU	1GPU	2GPU	4GPU
102400	43520	23808	12288	21504	11648	6080
204800	88064	47616	24832	43520	23808	12288
409600	176128	95232	49664	87040	47616	24832
819200	352256	190464	99328	176128	95232	49664
	8node			16node		
総粒子数	1GPU	2GPU	4GPU	1GPU	2GPU	4GPU
102400	10496	5760	2976	4992	2720	1392
204800	21504	11648	6080	10496	5760	2976
409600	43520	23808	12288	21504	11648	6080
819200	88064	47616	24832	43520	23808	12288

(GPU のみの計算）に対する相対性能を図 4.10，図 4.11 に示す．相対性能が 1 より大きければ，GPU のみの演算よりも高速であることになる．ここでは，HA-PACS の 2 ノードから 16 ノードを用いて，それぞれノード内の GPU 数を 1 から 4 に変化させて強スケーリングの評価を行う．図 4.10 では，全体的に XMP-dev/StarPU で速度向上が得られており，最大で GPU のみの演算に対して 1.4 倍に性能が向上していることがわかる．

一方図 4.11 では，ノード数が少ないとき，特に 2 ノード 1GPU の行列サイズ 16384 において GPU の

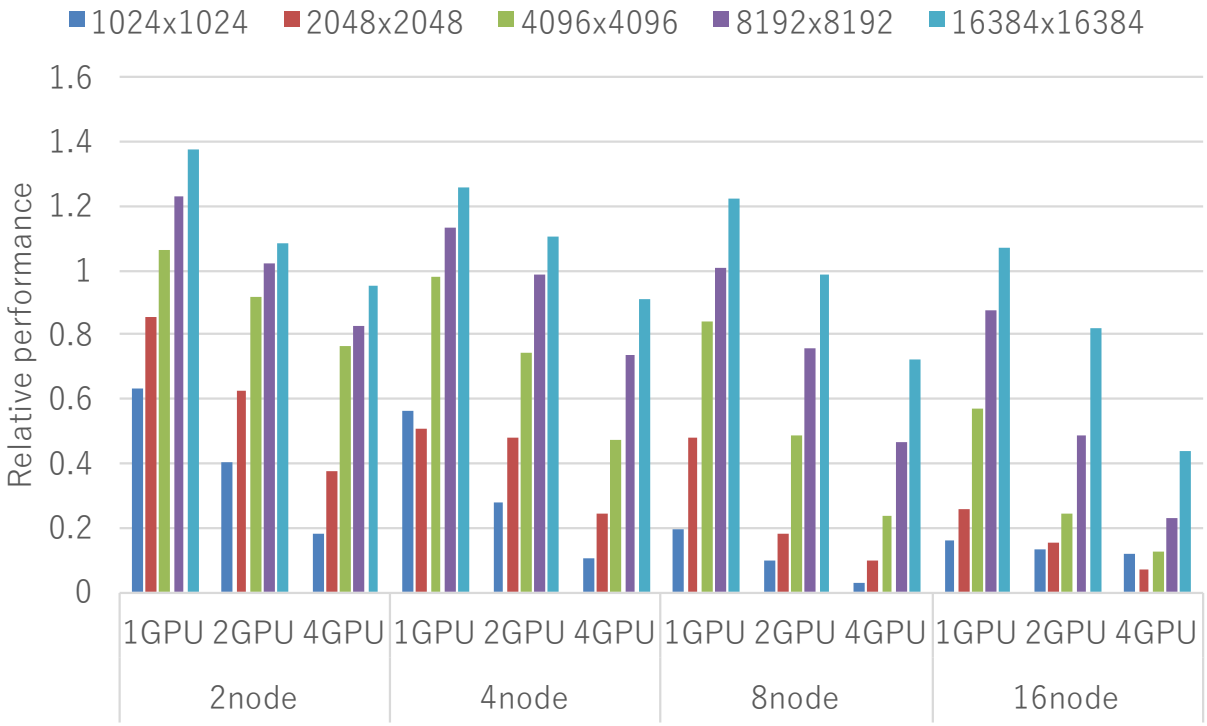


図 4.11　行列積：XMP-dev/CUDA に対する相対性能

みの演算に対して 1.4 倍の性能向上が得られている．しかし，ノード数を増やしていくと徐々に性能が下がり，8 ノードになると多くの場合において GPU のみの計算よりも遅くなっていることがわかる．行列積における強スケーリングでは，1 タイムステップの実行時間が短く，配列の分割や集約のオーバヘッドが見えてしまい速度低下に至ったと考えられる．

次にタスクサイズと速度向上の関係について考察する．表 4.5, 表 4.6 に CPU Weight が収束した時における，GPU に割り当てられたタスクサイズ（N 体問題では粒子数，行列積では部分行列のサイズ）を示す．N 体問題では，ほとんどのケースで GPU のみの場合に対する相対性能が 1 を上回っており，表 4.5 では 1.2 倍以上の性能向上が得られた場合をオレンジ色で示し，相対性能が 1 を下回ったものをブルーで示している．一方行列積は，ほとんどが相対性能で 1 を下回っているため，表 4.6 では相対性能で 1 を上回ったものだけオレンジ色で示している．表 4.5 より，ノード数・GPU 数を増加していくと相対性能が 1.2 倍を超えるオレンジ色の組み合わせが増加，つまり全体の性能向上が得られやすいことがわかる．強スケーリングでは，ノード数・GPU 数を増やしていくにつれてノードに割り当てられる問題サイズは小さくなっていく．そのような状況では，計算量が大きい時に有利な GPU の性能を引き出すことが難しい．このような時に，CPU の演算性能が加わることで，性能が伸びにくくなってしまうところを CPU が補っていると考えられる．また，GPU に大きすぎるタスクを渡すと，CPU の演算性能を十分に発揮させることができず大きな速度向上を得ることができない．一方表 4.6 では，オレンジ色の部分が各ノード数において左下の部分に集まっている．GPU に割り当てられた部分行列サイズをみると，オレンジ色の部分では比較的大きいサイズが割り当てられているが，相対性能が 1 を超えていない部分ではこれが小さ

表 4.6　行列積：1GPU に割り当てられた部分行列サイズ

	2node			4node		
全体行列サイズ	1GPU	2GPU	4GPU	1GPU	2GPU	4GPU
1024	348	199	101	192	103	57
2048	737	445	238	435	243	120
4096	1474	901	496	788	476	253
8192	2908	1740	972	1556	911	496
16384	5816	3440	1925	2990	1781	983
	8node			16node		
全体行列サイズ	1GPU	2GPU	4GPU	1GPU	2GPU	4GPU
1024	107	58	28	58	29	12
2048	238	124	53	126	63	30
4096	465	253	126	253	126	62
8192	819	491	250	471	253	124
16384	1576	931	496	860	496	248

いことがわかる．これは，行列積においては全体的に問題サイズが，総リソース量に比べて十分大きくなく，強スケーリングにおいて，GPU に割り当てられる部分行列サイズが小さすぎ，配列の分割や集約のオーバヘッドが見えてしまい速度低下に至ったと推測される．これを確認するために，XMP-dev/CUDA および XMP-dev/StarPU それぞれ 2 ノード 1GPU に対する性能を比較する．図 4.12，図 4.13 に N 体問題について，図 4.14，図 4.15 に行列積についての相対性能を示す．図 4.12 より，ノード数や GPU 数を増加させていくとそれに応じて性能が向上している．同様に CPU の計算リソースを加えた図 4.13 においてもスケールしており，CPU の計算リソースの追加がオーバヘッドになっていないことがわかる．図 4.13 のグラフ中で使用した GPU 数が同じ時，例えば，8 ノード 4GPU と 16 ノード 2GPU（どちらも総 GPU 数は 32）の性能は，前者よりも，後者の方が高い性能であることがわかる．これらの違いは，計算に参加する CPU core 数が影響しており，8 ノード 4GPU では $96(=(16-4)\times 8)$ コア，16 ノード 2GPU では $224(=(16-2)\times 16)$ コアであり，16 ノード 2GPU のほうが計算に参加する CPU コア数が多い．図 4.12 では，総 GPU 数が等しい時は性能がほぼ同じであるが，図 4.13 では，総 GPU 数が同じであっても，総 CPU core 数が多いほうがほとんどの場合において性能が高いことがわかる．これより，CPU の演算リソースが性能向上に寄与していることがわかる．しかし，図 4.10 より，問題サイズが小さい時，16 ノード 4GPU の性能が悪い．図 4.12 および図 4.13 でも問題サイズが小さい時にはスケールしていない．そのため，このようなサイズでは CPU Weight によるロードバランスの調節はうまくいかないことがわかる．

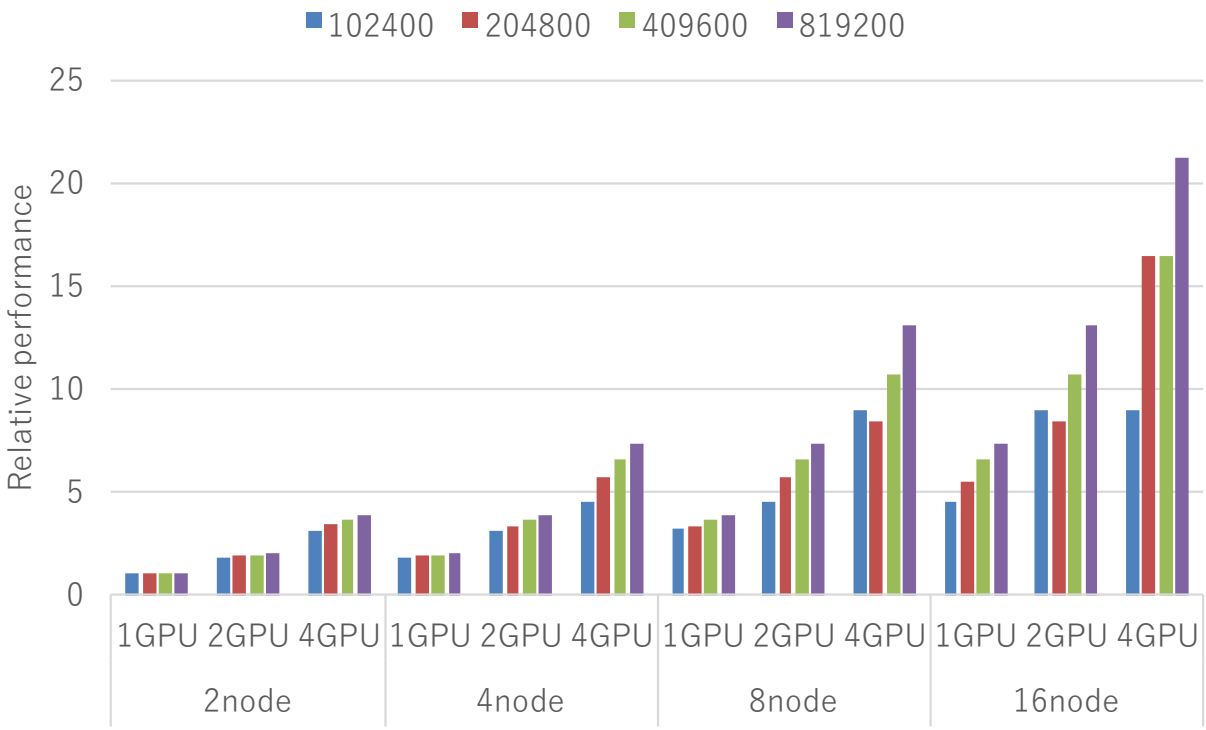


図 4.12　N 体問題：2node 1GPU に対する XMP-dev/CUDA の相対性能

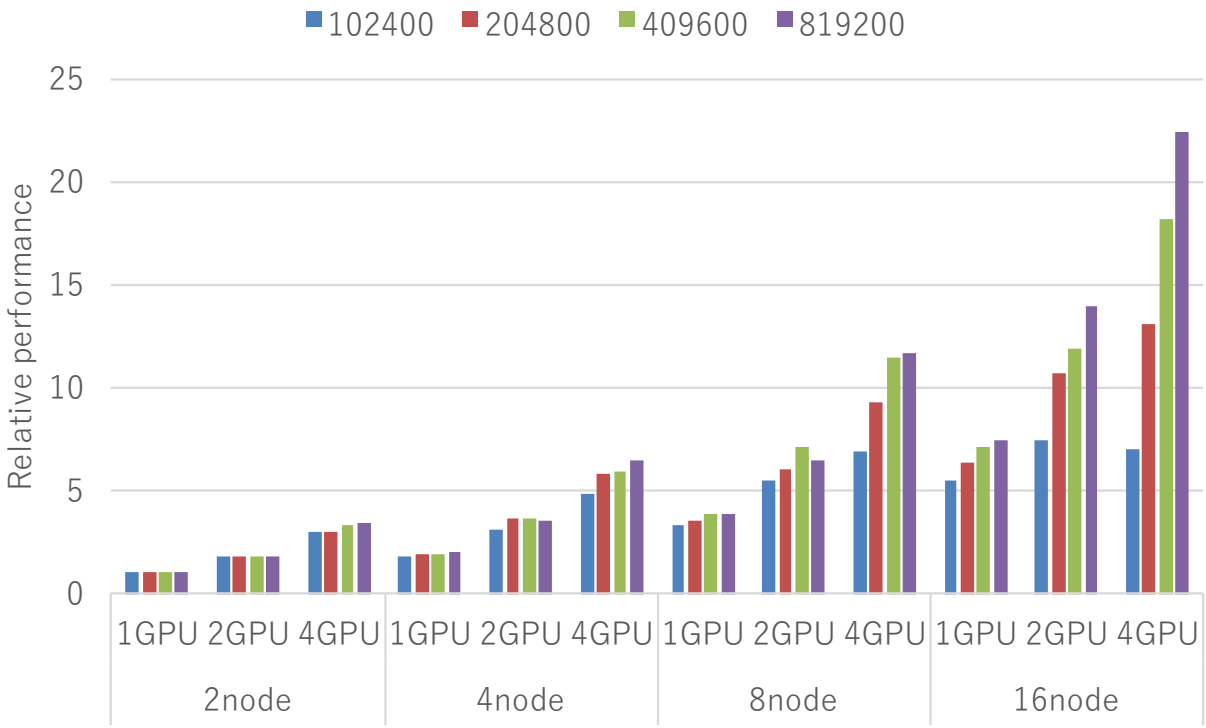


図 4.13　N 体問題：2node 1GPU に対する XMP-dev/StarPU の相対性能

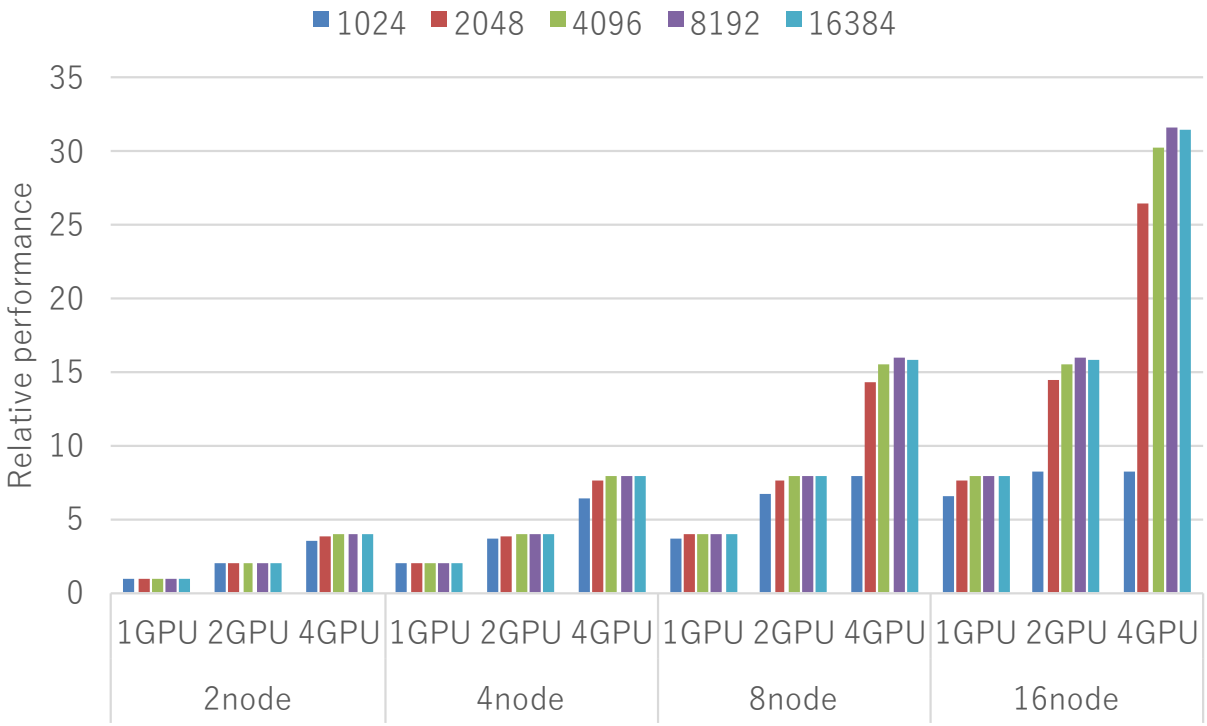


図 4.14　行列積：2node 1GPU に対する XMP-dev/CUDA の相対性能

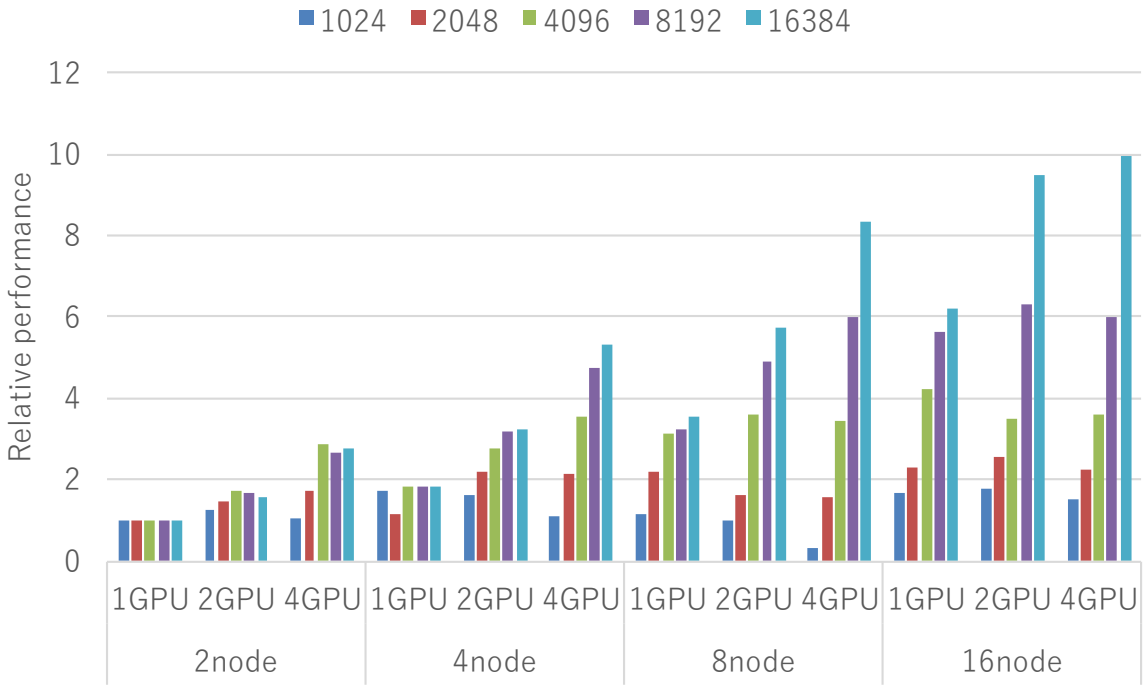


図 4.15　行列積：2node 1GPU に対する XMP-dev/StarPU の相対性能

4.5.4　行列積の評価

一方行列積は，図 4.11 より，問題サイズが小さい時に GPU のみの演算に対して性能が悪くなっている．この原因として，BLAS に与える部分行列のサイズが影響していると考えられる．図 4.14 より，GPU のみの計算では行列サイズが 1024 の時には 8 ノード 4GPU で性能が頭打ちになっているが，行列サイズ 8192 以上では理想的な性能向上が得られている．XMP-dev/StarPU では，図 4.15 から分かるとおり，どの組み合わせにおいてもほとんど性能が出ていない．これは GPU/CPU それぞれの BLAS に渡す部分行列のサイズが原因だと考えられる．XMP-dev/StarPU では CPU Weight によって GPU/CPU の計算量を決め，タスク数に応じてサイズが決定する．部分行列の行方向のサイズは表 4.6 に示すとおり，2 冪数ではないことがわかる．しかし，BLAS はキャッシュやメモリアクセスを最適化することで各リソースの性能を引き出している．内部実装に依存するが多くの場合，ブロッキングサイズやメモリアクセス，プロセッサコア数を考慮して 2 冪数であることが多い．大島らの研究 [7] や遠藤らの研究 [8, 9] でも，アクセラレータによって最適な行列サイズを事前に調査することで，高い演算性能を達成している．例えば，NVIDIA の GPU では行列のブロックサイズを 72 の倍数としており，これは，搭載されている GPU のピクセルシェーダ数が 6 個であることから導き出された値である．また，ClearSpeed では行列のブロックサイズを 288 の倍数としており，これは，ClearSpeed に搭載されている演算コアの数が 96 個であることが要因になっていると考えられる．そのため，リスト 4.6 のような単純なアニーリングアルゴリズムではうまく動作しないことがわかる．よって，アクセラレータなどの演算器の特性を理解することで，用いるアルゴリズム中で行列のブロックサイズの調整には，演算器の実行スレッド数やコア数の倍数を刻み幅として用いることで，より早い CPU Weight の探索だけでなく，高い演算性能を達成する CPU Weight を得ることができる．

これまでの評価において，CPU と GPU の実行時間がなるべく近づくようなアプローチを行っていた．しかしながら，行列積では多くの場合において，逆に性能低下を招いてしまった．特に，行列積は GPU を用いることで非常に高い性能が得られることが知られており，CPU と並行して演算に用いることが有効である範囲が小さい．そこで，ある程度データサイズが大きいときには GPU のみを使う，つまり，CPU Weight == 0.0 にすることが有効であると考えられる．図 4.16 に，XMP-dev/CUDA に対して，XMP-dev/StarPU の CPU Weight 値を 0 にした場合の相対時間を示す．これより，行列サイズが 8192 以上であれば XMP-dev/CUDA の 8〜9 割程度の性能であることがわかる．そこで，このような場合では GPU と CPU の実行時間が均衡する最適な CPU Weight ではなく，それぞれの実行時間が最小になるようなアルゴリズムをもちいて CPU Weight に調節する必要がある．このように，アプリケーションによって異なる CPU Weight の最適化は異なり，自動チューニングでは対応が困難である．しかし，本提案手法のように調整部分をユーザに委ねることで比較的容易にチューニングが可能になる．

4.6　本章の結論

本章では，GPU クラスタ上における，GPU と CPU によるハイブリッドプログラミングを行えるフレームワークの提案をした．本フレームワークは，アクセラレータ搭載の並列計算機向けの言語である

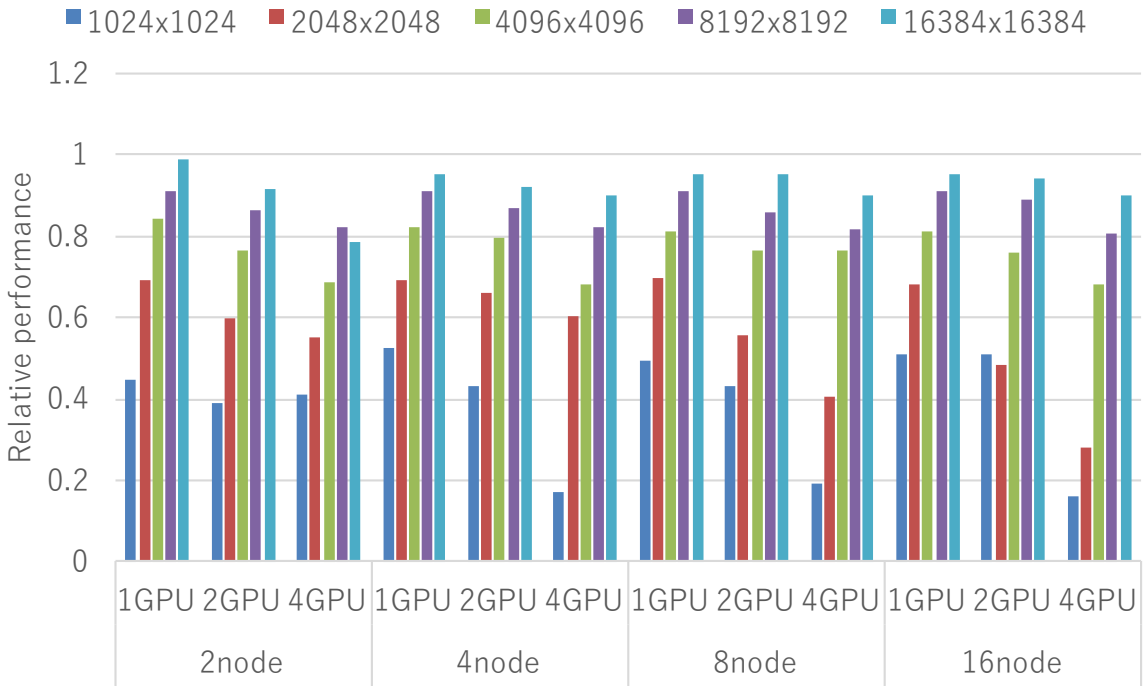


図 4.16　行列積：XMP-dev/CUDA に対する相対性能（CPU Weight == 0）

XMP-dev と，ヘテロジニアスな環境での CPU とアクセラレータ間での負荷分散が可能なランタイムシステムである StarPU を組み合わせた，XMP-dev/StarPU の実装を行った．XMP-dev は，分散メモリ環境においてグローバルなデータ分割，通信や並列処理が可能であり，その対象は GPU のみを対象としていたが，バックエンドスケジューラとして StarPU を適用することで，マルチノード環境での GPU と CPU によるワークシェアリングを可能としている．生産性の評価において，Hand-coding（MPI + StarPU）に対して，本フレームワークは 33.62% のコード行数で同様のプログラムを記述できることを示した．

StarPU はローカルノードのデータに対して，等分割によるタスクの生成のみに対応している．しかしながら，単純にデータを等分割しただけでは，GPU と CPU 間の演算性能の差が影響し，ロードバランスが非常に悪くなり，結果として GPU のみを演算に用いた場合に対して性能が低下してしまう．そこで，GPU と CPU の大きな演算性能の差に着目し，各演算リソース間での最適なデータサイズを割り当てるために，XMP-dev/StarPU に適応型負荷分散機能を導入した．同時に，データの割合をプログラムの実行中に動的に変更するための XMP-dev/StarPU の指示文を提供している．N 体問題と行列積を用いて，適用型負荷分散の評価を行ったところ，ユーザが提示したアルゴリズムに則って負荷のバランスを調整するパラメータである CPU Weight がある値に収束していくことを確認した．負荷の割合を最適化した結果，N 体問題において，GPU のみの演算に対して最大で 1.4 倍の性能向上を達成した．一方，問題サイズが小さいときや並列性を高くした時に，GPU に十分な計算量が確保できず，実行効率が非常に悪いことがあり，このような場合は GPU のみで計算を行うように，XMP-dev/StarPU コンパイラおよびランタイムシステムの変更を行う必要があると考えられる．行列積では，並列度を上げるにつれて逆に性能が

低下していくことを確認した．特に，行列積はキャッシュを有効に使うことで計算リソースの性能を最大限利用しているため，収束に至った CPU Weight では部分配列のサイズが最適なキャッシュやメモリアクセスでなかったためだと考えられる．このような場合，本フレームワークでは CPU Weight の調節をユーザに委ねており，演算器のコア数や同時稼働スレッド数に基づくチューニングが可能である．このように， XMP-dev/StarPU によって GPU クラスタ内の計算リソースの利用を最適化し，かつ，生産性の高いフレームワークであることを示した．

第 5 章

密結合並列演算加速機構 Tightly Coupled Accelerators

本章では，先行研究である TCA（Tightly Coupled Accelerators）および PEACH2（PCI Express Adaptive Communication Hub version 2）の概要と，TCA/PEACH2 の基礎通信性能を示す．

5.1 GPU クラスタにおける GPU 間通信

従来，ノード跨ぐ GPU 間における通信はホストメモリを経由しているため，レイテンシが非常に高いことが問題であった．そこで NVIDIA 社では，GPU 間の直接通信を実現するために GDR 機能を提供し，PCIe で接続された InfiniBand HCA による GPU 間直接通信を実現し，ホストを経由した通信よりもレイテンシが格段に小さくなった．しかしながら，アプリケーションの強スケーリング性能を求める場合，InfiniBand を経由した GPU 間通信ではプロトコル変換などのオーバヘッドがボトルネックとなり，性能低下の原因となっている．そこで，より高速な GPU 間直接通信機構として，筑波大学計算科学研究センターでは，PEACH2（PCI Express Adaptive Communication Hub version 2）に基づく密結合並列演算加速機構 TCA（Tightly Coupled Accelerators）の開発が進められている．TCA/PEACH2 は，PCIe のパケットを直接通信に利用するため，最小限のオーバヘッドで GPU 間通信が可能である．PEACH2 は，FPGA によって実装されているため機能拡張の柔軟性が高い．中でも，ハードウェアブロックストライド機能により，ブロックストライド通信が非常に低レイテンシで実行することが可能である．さらに，PEACH2 は PCIe Gen2 に基づいているが，現在 PCIe Gen3 に基づく PEACH3 [48, 49] が開発されており，低レイテンシだけでなく高いバンド幅性能も持ち合わせている．このように，将来性も高い TCA/PEACH2 を本研究では用いることとする．

5.2 TCA アーキテクチャと PEACH2

TCA アーキテクチャおよび PEACH2 は文献 [14, 15, 16, 17] に詳しいが，本節ではその概要を説明する．

5.2.1　Tightly Coupled Accelerators

密結合並列演算加速機構 TCA は，ノード間のアクセラレータ（GPU）間を直接結合することで，アクセラレータ間通信のレイテンシを改善することを目的にしたコンセプトで，筑波大学計算科学研究センターが中心となって開発が進められている．現在は，PCIe をノード間通信に拡張することで TCA を実現している．PC クラスタにおける GPU，Intel MIC（Many Integrated Core processor）あるいは FPGA 等は PCIe によってホスト CPU および並列ネットワークと接続されているため，この構成であらゆるアクセラレータを対象にできる．

これを実現する実装として，筑波大学計算科学研究センターが中心となって開発を進めている PEACH2 がある．この PEACH2 ボード同士を PCIe 外部ケーブルで接続し，TCA システムを構成する．さらに TCA コンセプトの実証実験クラスタとして，筑波大学計算科学研究センターの GPU クラスタ HA-PACS (Highly Accelerated Parallel Advanced system for Computational Sciences) [46] の拡張部として HA-PACS/TCA を構築し，運用している．

図 5.1 に，HA-PACS/TCA のノード構成を示す．HA-PACS/TCA のノードは，2 ソケットの Intel Xeon E5-2680v2 CPU と 4 枚の NVIDIA K20X (Kepler アーキテクチャ) GPU を搭載し，CPU0 側には PEACH2 ボードが，CPU1 側には InfiniBand HCA（QDR 2 rails）が接続されている．図中，GPU には CPU が直接接続されているように見えるが，実際には CPU に内蔵されている PCIe スイッチを介して PEACH2 または InfiniBand HCA に接続されているため，実際の通信は CPU を介さず行われる．InfiniBand は HA-PACS/TCA のすべてノードを単一スイッチでフラットに接続している．一方，TCA/PEACH2 のみで大規模なクラスタを構成することは，外部接続ケーブル長の限界や，PCIe パケットのホップ数の増加に伴う性能面での制約により困難であるため，図 5.2 のように 16 ノードを TCA/PEACH2 で結合している．この集団を「サブクラスタ」と呼び，HA-PACS/TCA では，64 ノードが 4 つのサブクラスタに分かれている．しかし，同時に，64 ノードすべてが InfiniBand によっても結合されているため，あるノードから見ると，TCA/PEACH2 で直接通信可能なノードは 15 台あり，それらを含めたすべてのノードとは InfiniBand 経由で通信ができる．

5.2.2　PEACH2 チップ

各ノードの PEACH2 ボードには，PEACH2 チップが搭載されている．PEACH2 チップは，PCIe パケットの中継処理や DMA 転送などを行い，FPGA (Altera 社 Stratix Ⅳ GX[50]) で実装されている．図 5.3 に PEACH2 チップの構成を示す．このチップは，4 つの PCIe Gen2 ×8 ポートを持ち，1 つはホスト CPU（CPU & GPU side）と接続し，残り 3 つのポート（To PEACH2）を隣接ノードの PEACH2 ボードとの接続に使用する．このように，PEACH2 はハードウェアの制約により，1 つのノードから外部へ伸びるリンクが 3 つに限られる．一方で，PEACH2 のみで多くのノードを接続すると，ホップ数が増加し，低レイテンシ通信が有効に使えない．そこで 16 ノードまでの PEACH2 による直接結合を考え，その中で最小のホップ数を実現できるトポロジとして 2×8 の 2 重リングトポロジとしてサブクラスタを構成する．PEACH2 には高度な DMA コントローラ (以降「DMAC」と略す) が 4 チャネル搭載されて

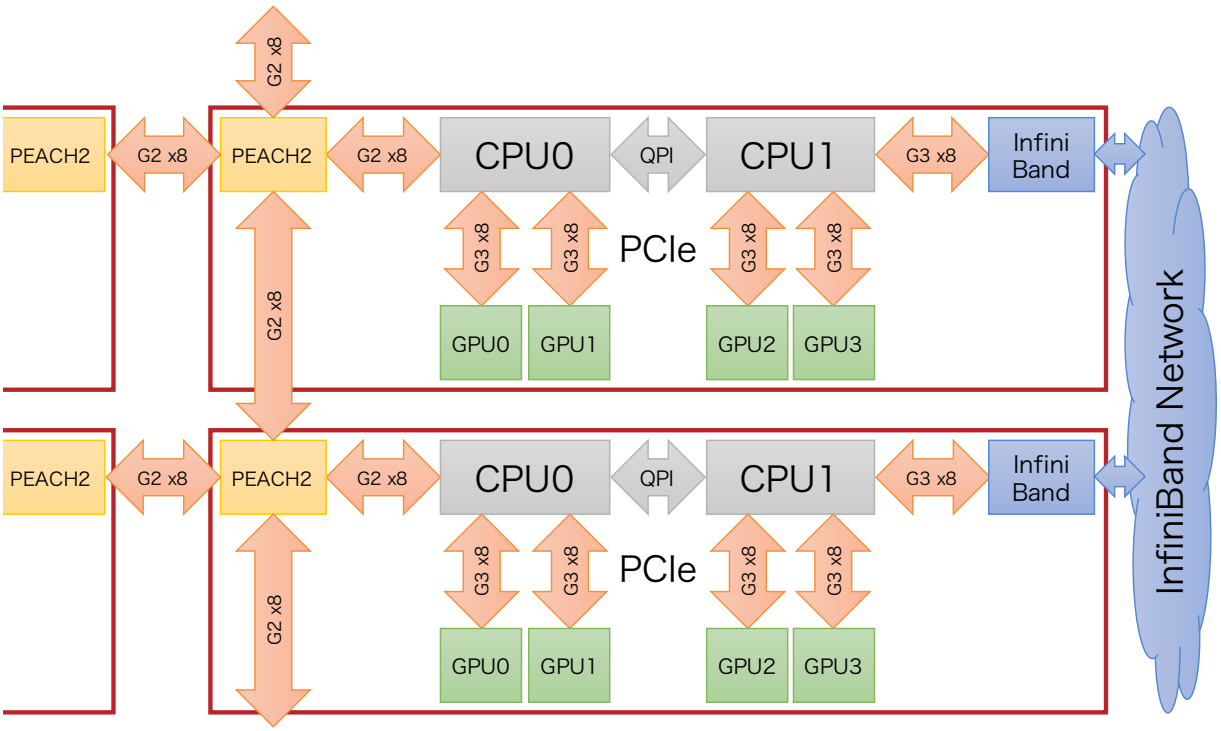


図 5.1　HA-PACS/TCA のノード構成

おり，高速な DMA や Chained DMA などが可能である．

5.2.3　PEACH2 による通信

PEACH2 では，DMA と PIO の 2 つの通信方式を提供している．

PEACH2 の DMA 通信は，リモートノードに対するアクセスは，基本的に RDMA Put プロトコルのみをサポートする．ホスト上であらかじめ読み込み元，書き込み先の PCIe アドレス，サイズを指定したディスクリプタを作成し，これらをアドレスポインタで連結する．これらの読み込み元，書き込み先にホストメモリ，デバイスメモリを指定できる．そのため，GPU 間のデータ転送だけでなく，デバイスメモリから直接リモートのホストメモリにデータを転送することも可能である．DMA 通信を始める時は，ディスクリプタの先頭のアドレスを指定することで，連続して DMA 処理を行うことが可能である．また，DMA 通信は連続領域に対するデータ転送だけでなく，バースト長およびギャップを指定することで，ブロックストライド転送を行うことができる．ブロックストライド転送はステンシル計算などで必要となる隣接ノードとの袖領域交換において頻繁に用いられるが，従来の MPI では，データを Pack/Unpack して送る必要がある．PEACH2 では Chained DMA を使うことで，Pack/Unpack が必要無くなり，メッセージ長がある程度小さい場合において高いバンド幅が得られる．このように，あらかじめデータ通信パターン（通信相手と通信領域）が定められていれば，Chained DMA 機構により柔軟かつ高速な通信が実現できる．

PEACH2 の DMA 通信では，ディスクリプタの登録方法によって主に 2 つの通信モードを提供している．

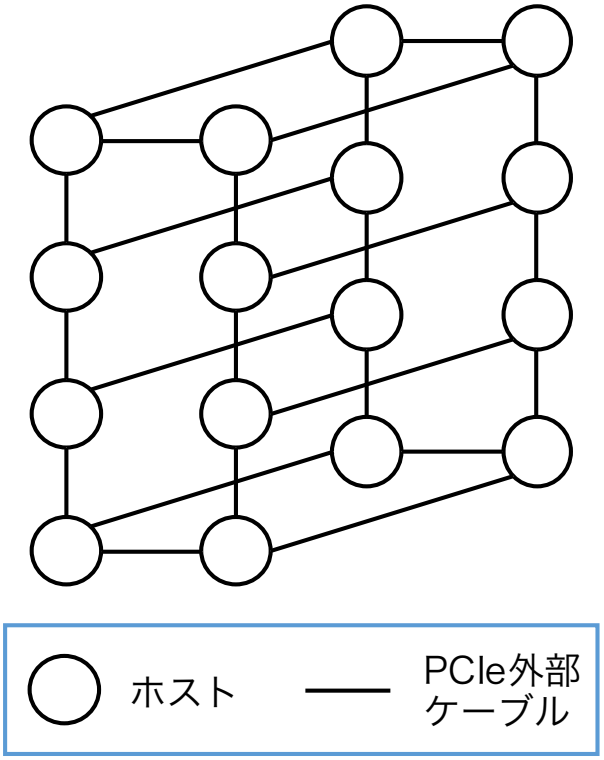


図 5.2　サブクラスタ：TCA ネットワーク構成

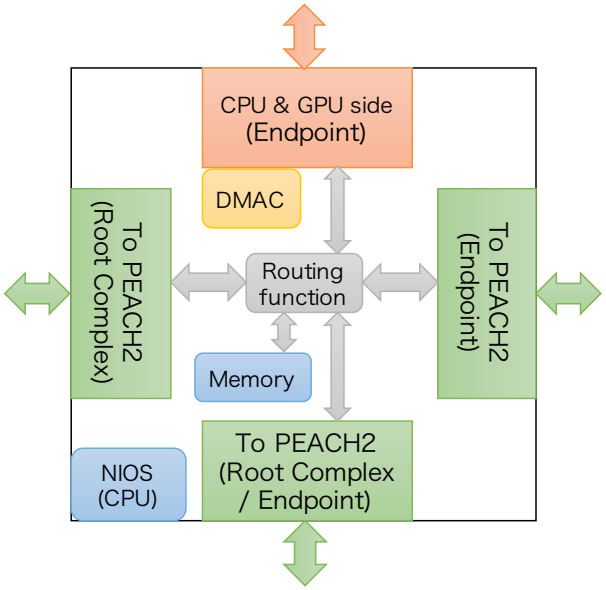


図 5.3　PEACH2 チップの構成

ホストメモリモード

ホスト CPU のメモリ上に必要なサイズのディスクリプタを作成しておき，通信開始時に必要なディスクリプタをホストメモリから読み出し，通信を行う．ホストメモリを使って多数のディスクリプタを管理できるため，連結が長い場合に有効である．しかし，ホストメモリへの読み出しが必要なため，最低レイテンシは約 2.3μsec にとどまる．

内蔵メモリモード

PEACH2 の FPGA に内蔵されているメモリにディスクリプタを登録する．ディスクリプタを格納するメモリに高速なアクセスが可能なため，通信のレイテンシが短縮される．特にデータサイズが小さいときに優位な性能をもち，256 Byte までの通信で約 2.0μsec の低レイテンシを実現する．しかし，内蔵メモリの容量には限りがあるため，最大で 1024 個のディスクリプタまでしか登録することができない．

通信データサイズが小さい場合，内蔵メモリモードはホストメモリモードに比べより低レイテンシであるが，ある程度通信データサイズが大きくなるとその差はほとんど無くなる．本論文における袖領域交換では通信サイズが 256 Byte より大きくなるため，ホストメモリモードと内蔵メモリモードによる性能差が全体の性能に影響することは無いと考えている．一方で，ホストメモリモードと内蔵メモリモードの切り替えは flag を変更するだけで行えるため，より高速な通信が期待できる「内蔵メモリモード」を評価に

使用する．

PIO（Programmed I/O）通信は，CPU の store 命令によってリモートに直接書き込みを行う．そのため，DMA 通信において発生した DMAC の起動などのオーバヘッドが削減できるため，通信レイテンシが極めて低く，小さなデータの転送に向いている．DMA 通信は，GPU 間通信やデバイスメモリとホストメモリ間の通信が可能だが，PIO 通信はホストメモリ間の通信のみを提供する．しかし，最小レイテンシが約 0.9μsec と非常に高速な通信が可能である．

5.2.4　GPUDirect Support for RDMA

PEACH2 では GPU 間の直接通信を行うために，NVIDIA が提供する GDR を用いる．GDR は，CUDA5.0 以降および Kepler アーキテクチャ世代以降の GPU を用いることで，GPU 上のデバイスメモリを PCIe アドレス空間にマッピングすることができる．同じ PCIe アドレス空間に属する GPU 同士では，ホストメモリを経由することなく直接 PCIe 上でデータの転送が可能になる．TCA/PEACH2 では，ノード間に跨って PCIe アドレス空間を共有することができるため，ノード間の GPU 間直接通信を実現することができる．

前述のとおり，各ノード内の PEACH2，GPU 群，InfiniBand HCA は CPU 内の PCIe スイッチを介して接続されるが，2 つの CPU ソケットを跨いだアクセスにおいて重大な性能低下問題が発生する．Sandy Bridge 以降の世代の Intel Xeon CPU では，ソケット間を跨ぐ QPI（QuickPath Interconnect）を介して双方に接続されている PCIe デバイス間通信が可能であるが，その際のバンド幅が著しく低下することが知られている [15]．図 5.1 において，各ノード内の GPU0，1 と GPU2，3 間の GDR を行う場合この問題が発生する．このため，現状では PEACH2 がアクセスする GPU は CPU0 側の GPU0，1 のみに限定している．

5.2.5　PEACH2 を用いたプログラミング

PEACH2 による GPU 並列プログラムは，NVIDIA 社が提供する CUDA 環境上で動作することが前提である．つまり，TCA/PEACH2 を用いたプログラムでも，GPU の管理（メモリの確保，ホスト・デバイス間のデータ転送，カーネル関数の起動など）は，一般的な GPU プログラミングモデルと同様である．これに加え，PEACH2 による通信を行うためには PCIe アドレスを直接指定する必要があり，これは一般的な配列のポインタとは型が異なる．TCA/PEACH2 を用いたプログラミングでは *tcaHandle* と呼ばれるメモリハンドルを定義し，PCIe アドレスの管理を容易にしている．RDMA Put プロトコルによる通信では，リモートノードの書き込み先アドレスが必要になる．そのため，送信先の *tcaHandle* が必要となり，TCP/IP や MPI を用いてノード間で交換を行う．これは，InfiniBand を用いた通信でも同様な制限があり，一般的な手法であると言える．

DMA による通信を行うためには，TCA API である **tcaSetDMADesc_Memcpy()** を用いて送信先の *tcaHandle*, 使用する DMA のチャネル番号, 配列のオフセットなどを含むディスクリプタを作成し，ディスクリプタをポインタで連結する．**tcaStartDMADesc()** で，ディスクリプタを設定した DMA のチャネル番号を指定し，連結されたディスクリプタの通信が連続して開始される．この通信は非同期で

行われ，**tcaWaitDMADesc()** によって通信の完了を待機する必要がある．

TCA/PEACH2 による通信を何度も行う場合，あらかじめ通信の設定をしておくことで，DMA による通信を起動するだけで通信を開始することができる．このようにすることで，時間発展ループ内で何度も通信を行う場合に TCA/PEACH2 の低レイテンシ通信を最大限活用することが可能になる．

5.3　GPU 間通信の基礎性能

本節では，TCA/PEACH2 と InfiniBand の通信性能について議論するために，隣接ノード間 GPU 通信の基礎評価を行う．ここでは特に，単純な連続データ通信だけでなく，多次元ステンシル計算で必要となる，多次元の袖領域（ステンシル計算で隣接領域間の境界となる部分）の通信に着目する．ハイブリッド通信において TCA/PEACH2 が得意とする通信と InfiniBand 通信をどのように組み合わせるかが性能の鍵となるため，GPU の配置と利用するネットワークの組み合わせに関する最適化を視野に入れ，基礎評価を行う．評価環境として，表 5.1 に示す HA-PACS/TCA の 2 ノードを用いる．TCA/PEACH2 による通信でホップ数が増えないように，図 5.2 の TCA/PEACH2 ネットワーク上で隣接するようにプロセスを配置する．InfiniBand 経由の GPU 間通信を利用するための MPI として，オハイオ州立大学が開発している MV2GDR[12] を用いる．MV2GDR は，TCA/PEACH2 と同様に NVIDIA の GDR を用いて，InfiniBand を経由した GPU 間高速通信を実現している．HA-PACS/TCA の InfiniBand HCA は，PCIe Gen3 ×8 上に QDR ×4 クラスのインタフェースが 2 rails 実装されているが，本論文では，主に小メッセージにおけるレイテンシに着目し，TCA/PEACH2 の 1 リンクに対して InfiniBand の 1 リンクとしたときの性能比較を行う．よって，性能評価では InfiniBand の 1 ポートのみを利用する．5.2.4 節で述べたように，QPI を経由した GPU 間直接通信は性能が極端に低下してしまう問題があるため，図 5.1 の環境では，TCA/PEACH2 の測定には GPU0 同士，MPI の測定には GPU2 同士の性能を測定する．一方，ハイブリッド通信では，TCA/PEACH2 の性能を有効に利用するため，CPU0 側の GPU0, 1 を対象とする．しかし，MV2GDR が InfiniBand 経由で GPU 間通信を行う場合，QPI を経由する必要があり，QPI を経由しない MV2GDR よりも性能が低下してしまう．

図 5.4 にノード間の GPU 同士で 1 次元配列の Ping-Pong 通信を行った場合の性能を示す．凡例の「TCA」は，TCA/PEACH2 経由による GPU0 同士の通信，「MV2GDR」は InfiniBand 経由による GPU2 同士の通信，「MV2GDR-QPI」は InfiniBand 経由による QPI を跨いだ GPU0 同士の通信を示す．これより，TCA/PEACH2 は MV2GDR に対して 512KB まで優位な性能を示し，特に比較的小さいメッセージにおいて高い性能を示していることがわかる．しかし，MV2GDR-QPI の通信では 256KB までのバンド幅が最高でも 290MB/s 程度しか出ていない．デフォルトの MV2GDR は，以下のようにデータサイズによってプロトコルスイッチ [51] が行われる．ここでメッセージサイズを x とする．

$x \leq 8\mathrm{KB}$
: GDR による GPU 間直接通信．

$8\mathrm{KB} < x \leq 256\mathrm{KB}$
: 一度ホストにデータをコピーし，その後 GDR を用いてリモートの GPU へデータを書き込む．

表 5.1　評価環境（HA-PACS/TCA）

CPU	Intel Xeon E5-2680v2 2.8GHz × 2 sockets
GPU	NVIDIA Tesla K20X × 4
Main Memory	DDR3 1866MHz 128GB
GPU Memory	GDDR5 6GB / GPU
Interconnection	InfiniBand: Mellanox Connect-X3 Dual-port QDR TCA: PEACH2 Board
OFED	Mellanox OFED-2.2-1.0.1
OS	CentOS release 6.4
MPI	MVAPICH2-GDR 2.0 （MV2_USE_CUDA=1）
GPU Compiler	CUDA 6.0 NVIDIA Driver 340.32

256KB $< x$

ローカルホストへデータ転送，ホスト間のメッセージパッシング，リモートノードでのホストからデバイスへのデータ転送という 3 つの通信をパイプラインで処理する．これによって GDR による通信よりも通信効率が良くなる．

本来 QPI を経由する通信は性能が低下してしまうが，512KB からはホストメモリを経由するため「MV2GDR-QPI」の性能は「MV2GDR」と同様になっていることがわかる．これは，QPI によって極端に性能が低下するのは，デバイス間での PCIe による通信の場合だけであり，QPI を超えたホストメモリのアクセスはわずかな性能低下にとどまるためである．MV2GDR は，実行時の環境変数でホスト経由の通信を行うようにするデータサイズを切り替えることができる．本論文における測定では，GDR のみで通信するデータサイズを 8KB とするのが我々の調査によって最適であることがわかっており，「MV2GDR-QPI-Tuned」がこれに該当する．つまり，8KB までは GDR による GPU 間直接通信，それ以上ではホスト経由の通信に切り替わる選択が MV2GDR の中で行われる．

次に，3 次元配列の袖領域通信について基礎評価を行う．図 5.5 に 3 次元配列の袖領域アクセスパターン，図 5.6 および図 5.7 に各通信パターンの Ping-Pong 通信性能を示す．3 次元配列では，jk-平面はメモリアドレスが連続するブロック転送で，ik-平面は N 要素の連続転送が $N \times N$ 周期で現れるブロックストライド転送である．ij-平面は，1 要素の転送が N 周期で現れるストライド転送になる．一般的に，ブロックストライド通信やストライド通信は，細粒度の通信を何度も行う必要があり，通信効率が悪い．そのため，InfiniBand 上の MPI 等では，バッファとして用いる配列にデータを Packing し，相手ノードに転送をした後，Unpacking をすることで通信の効率を向上している．TCA/PEACH2 では，ブロック転送およびブロックストライド転送には Chained DMA を用い，ストライド転送には Pack/Unpack を行う．

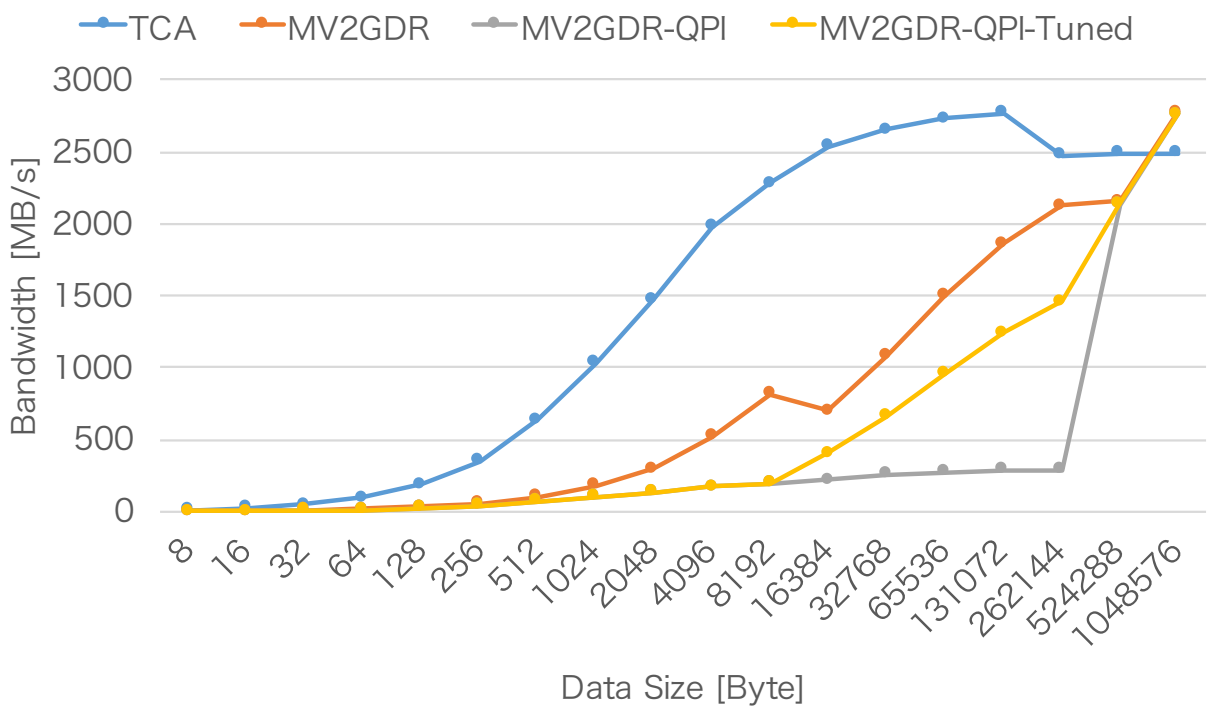


図 5.4　1 次元配列の Ping-Pong 通信性能

図 5.6 および図 5.7 はそれぞれ，サイズ N^3 の 3 次元配列における各面のデータを隣接ノード間で通信する場合の Ping-Pong 通信のレイテンシとバンド幅を示している．ブロック転送について，TCA/PEACH2 はサイズが小さい場合においても非常に高いバンド幅を示している．TCA/PEACH2 と MV2GDR の性能は，$N = 320 \sim 384$ で逆転する．図 5.6 では Latency の折れ線が重なっているように，TCA/PEACH2 のブロックストライド転送は，ブロック転送とほぼ同等の通信性能がある．これより，PEACH2 の DMAC に搭載しているハードウェアブロックストライド機能が非常に有用であることがわかる．一方，MV2GDR は Pack/Unpack を行う必要があるため，ブロック転送に比べ性能が低く，$N = 576 \sim 640$ と大きなサイズまで TCA/PEACH2 の性能が優位である．特に，図 5.6 のようにデータサイズが小さい時には Pack/Unpack が大きなレイテンシ増加の原因になっていることがわかる．ストライド転送では，ともに Pack/Unpack が必要であるため全体の性能差は他の通信方向よりも小さく，$N = 512$ で逆転する．

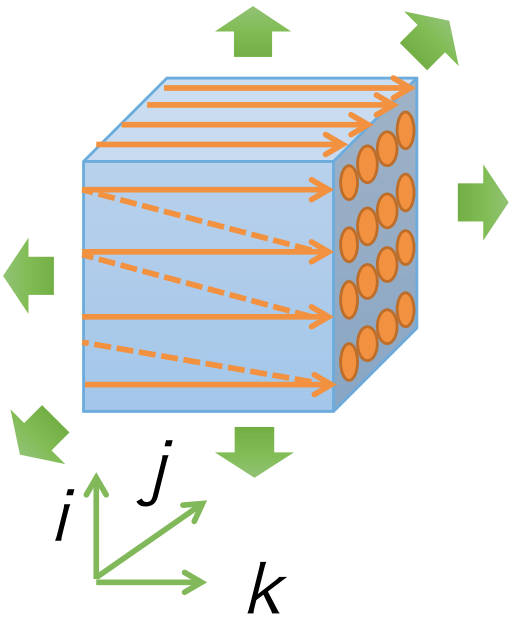


図 5.5　3 次元配列の袖領域交換の通信パターン

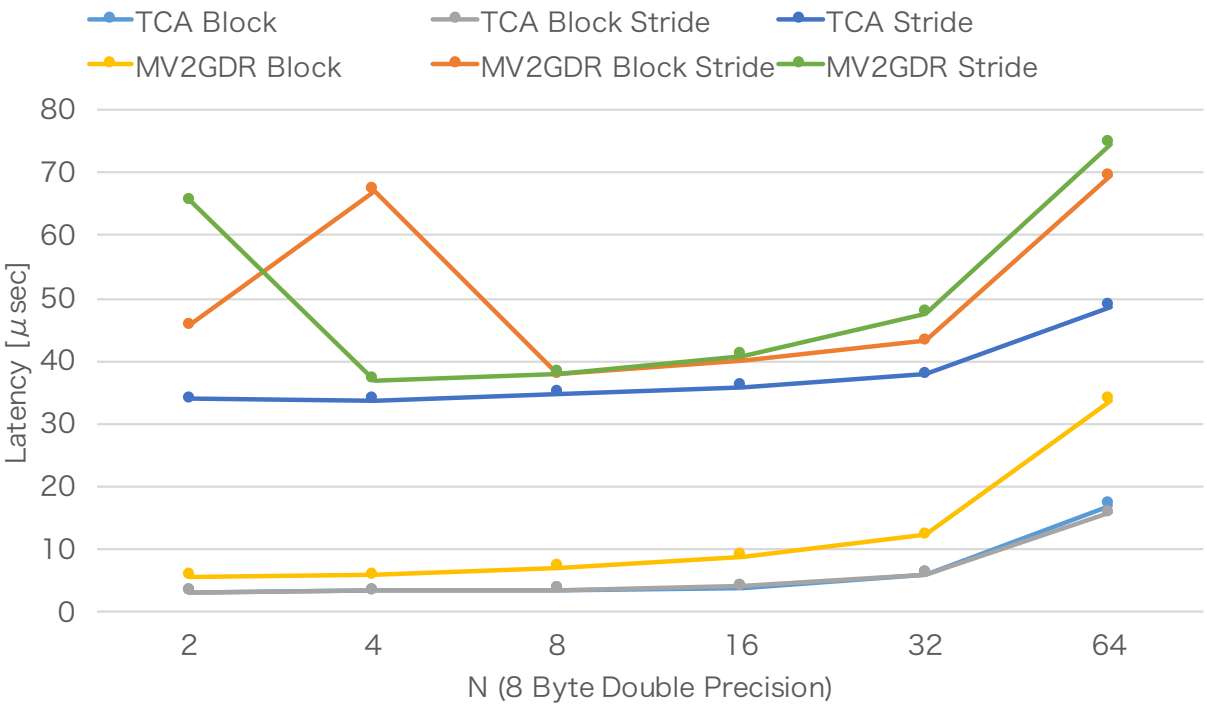


図 5.6　3 次元配列の Ping-Pong 通信によるレイテンシ

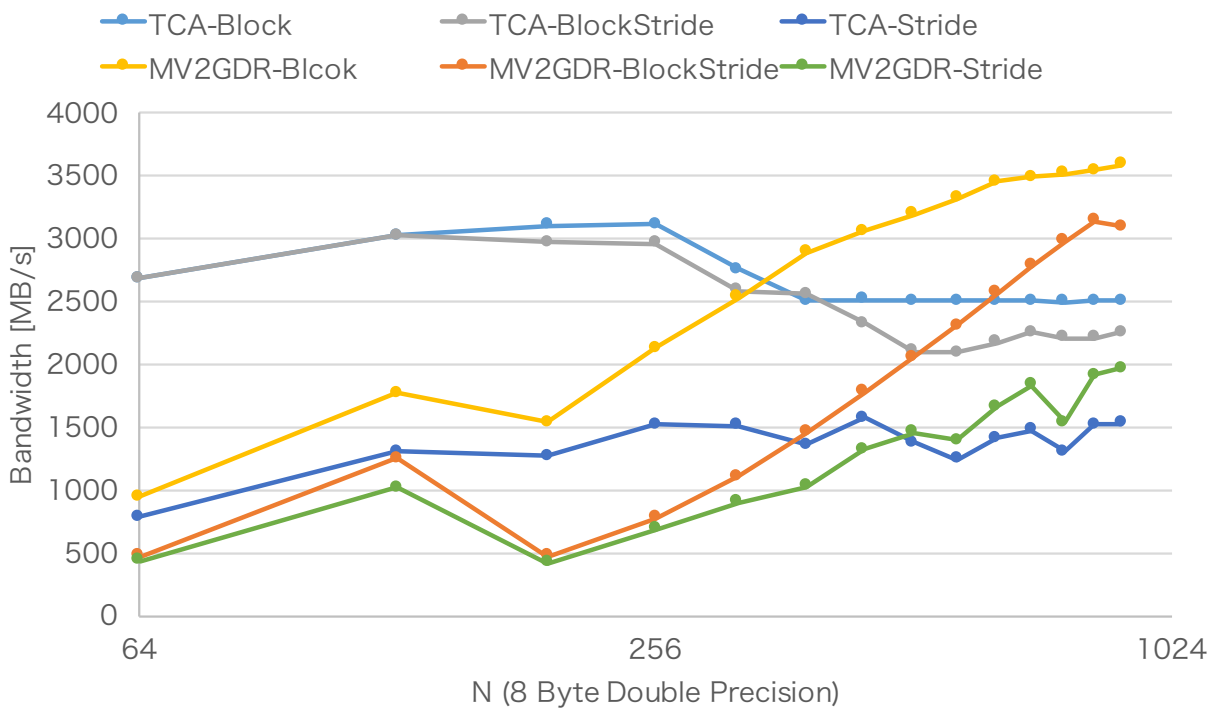


図 5.7　3 次元配列の Ping-Pong 通信によるバンド幅

第 6 章

TCA/InfiniBand ハイブリッド通信

本研究は，アクセラレータ搭載の並列計算機向けの高並列言語である XcalableACC の通信ランタイムに，提案する TCA/PEACH2 および InfiniBand による TCA/InfiniBand ハイブリッド通信を適用する．GPU クラスタでは，すべてのノードが InfiniBand で接続されていることが一般的であり，その局所的なネットワークを加速するように TCA/PEACH2 を用いることで，InfiniBand のみの接続では達成できない速度向上を実現する．

6.1 TCA/InfiniBand ハイブリッド通信

本節では，これまでに提案している TCA/PEACH2 と InfiniBand によるハイブリッド通信 [52] の概要と実装について述べる．

6.1.1 ハイブリッド通信の概要

TCA/PEACH2 は低レイテンシ通信を実現するが，PEACH2 の適用範囲は PCIe 等のハードウェアの制約や実装効率の面で数十ノードを一組とするサブクラスタを構成するにとどまっている．しかし，より大きな問題やサイズに対応するためにはサブクラスタを跨いだ通信が必要とされる．一方で，システム内の全てのノードを InfiniBand によってフラットに接続することは自然であり， TCA/PEACH2 は InfiniBand によるクラスタにおけるローカルな通信を加速するネットワークと捉えることができる（そのローカルな高速通信が可能な単位がサブクラスタである）．そこで，高いバンド幅やスケーラビリティを持つ InfiniBand ネットワークに，局所的な通信に対して低レイテンシ通信を実現する TCA/PEACH2 を加えることによって，スケーラビリティと InfiniBand のみでは得られない通信性能の向上を目的としたハイブリッド通信を実現する．このことにより，単に 2 つの通信路を束ねて全体のバンド幅を向上させるのではなく，それぞれの通信の特徴に応じたチャネルを選択することで，全体の通信性能を向上させることを目指す．このハイブリッド通信は，ステンシル計算における袖領域交換などにおいて，TCA/PEACH2 と InfiniBand の通信がそれぞれを補完しあうことで全体の通信性能が向上するだけではなく，サブクラスタを跨いだ集団通信においても有効であると考えている．松本らは，TCA/PEACH2 のサブクラスタ内における集団通信を実装し評価を行っている [53, 54]．アプリケーションを強スケー

リングさせる際には，集団通信の各メッセージサイズが小さくなるため，InfiniBand では大きなボトルネックとなってきた．TCA/PEACH2 では，強スケーリングした際のメッセージサイズでも高い性能が得られる．そこで，TCA/PEACH2 と InfiniBand によるサブクラスタ間通信を組み合わせることで，各サブクラスタ内では TCA/PEACH2 の高速な通信を活かし，最終的にサブクラスタ間の InfiniBand を経由して交換するデータ量を最小限にすることで，全体として低レイテンシ通信を行うことが期待できる．このような階層的な低レイテンシの集団通信は，単に InfiniBand のバンド幅を増やすだけでは実現することはできず，InfiniBand に TCA/PEACH2 を加えることによるハイブリッド通信によって実現できる．このように，提案する TCA/InfiniBand ハイブリッド通信では，それぞれのネットワークの特徴から，データサイズが小さく，距離が短い通信は TCA/PEACH2，データサイズが大きく，サブクラスタを跨ぐような距離が長い通信に関しては MPI/InfiniBand を割り当てる．一方で，TCA/PEACH2 と InfiniBand の通信を bonding して同一方向のネットワークのバンド幅を増やすことも可能である．しかし，特性が異なるネットワーク同士では，特に TCA/PEACH2 の低レイテンシ通信が有効に活用できないため，本論文ではこのような利用は想定していない．

6.1.2　実装の方針

5.3 節より，TCA/PEACH2 と InfiniBand 経由の通信はデータサイズや通信パターンによって性能特性が異なることがわかった．そのため，単純に TCA/PEACH2 通信では通信できない相手に対して MV2GDR の通信を用いるだけでは，多くの場合において通信性能のバランスが取れず，全体の性能低下につながってしまう恐れがある．そこで，TCA/InfiniBand ハイブリッド通信では，通信パターンによって TCA/PEACH2 または MPI の通信を適宜選択し，TCA/PEACH2 と MPI の特徴を活かした通信を行うことを考える．本論文では，袖領域交換および集団通信（Allgather，Broadcast，Allreduce）に対してハイブリッド通信を適用した．

袖領域交換は次元数によって通信の種類が異なる．2 次元配列の袖領域交換では，ブロック通信（行方向）とストライド通信（列方向）の通信パターンがあり，3 次元配列では，図 5.5 より，jk-平面はブロック通信，ik-平面はブロックストライド通信，ij-平面はストライド通信がある．図 6.1 にハイブリッド通信による袖領域交換の流れを示す．隣接するノードに対して，ブロックストライド通信またはストライド通信が発生する場合，Initialize フェイズであらかじめ Pack/Unpack に必要なバッファ配列をデバイスメモリ上に確保しておく．本論文では，実行中に袖領域のサイズが変わるようなことがないことを前提とし，TCA/PEACH2 および MPI の通信を永続通信として設定する．この前提は，通常の Domain Decomposition によるステンシル計算で一般的に成り立つものである．そして，時間発展ループ中に通信が実行された時に，非連続領域を Pack し， MPI と TCA/PEACH2 の通信を開始する．これらの通信は非同期で実行され，Wait 関数でそれぞれの通信の完了を待つ．その後，Unpack を行うことで袖領域交換が完了する．

次に，ハイブリッド通信による集団通信の概要を示す．まず，Broadcast 通信について述べる．本研究では，Broadcast の通信アルゴリズムとして図 6.2 のように $\log_2 p$（p はプロセス数を示す）ステップで通信が完了するアルゴリズムを用いる．図 6.2 では，8 つのプロセスが 2 つのサブクラスタに属している．これらのサブクラスタ間は TCA/PEACH2 による通信ができないため，Step 1 では MPI を用いて

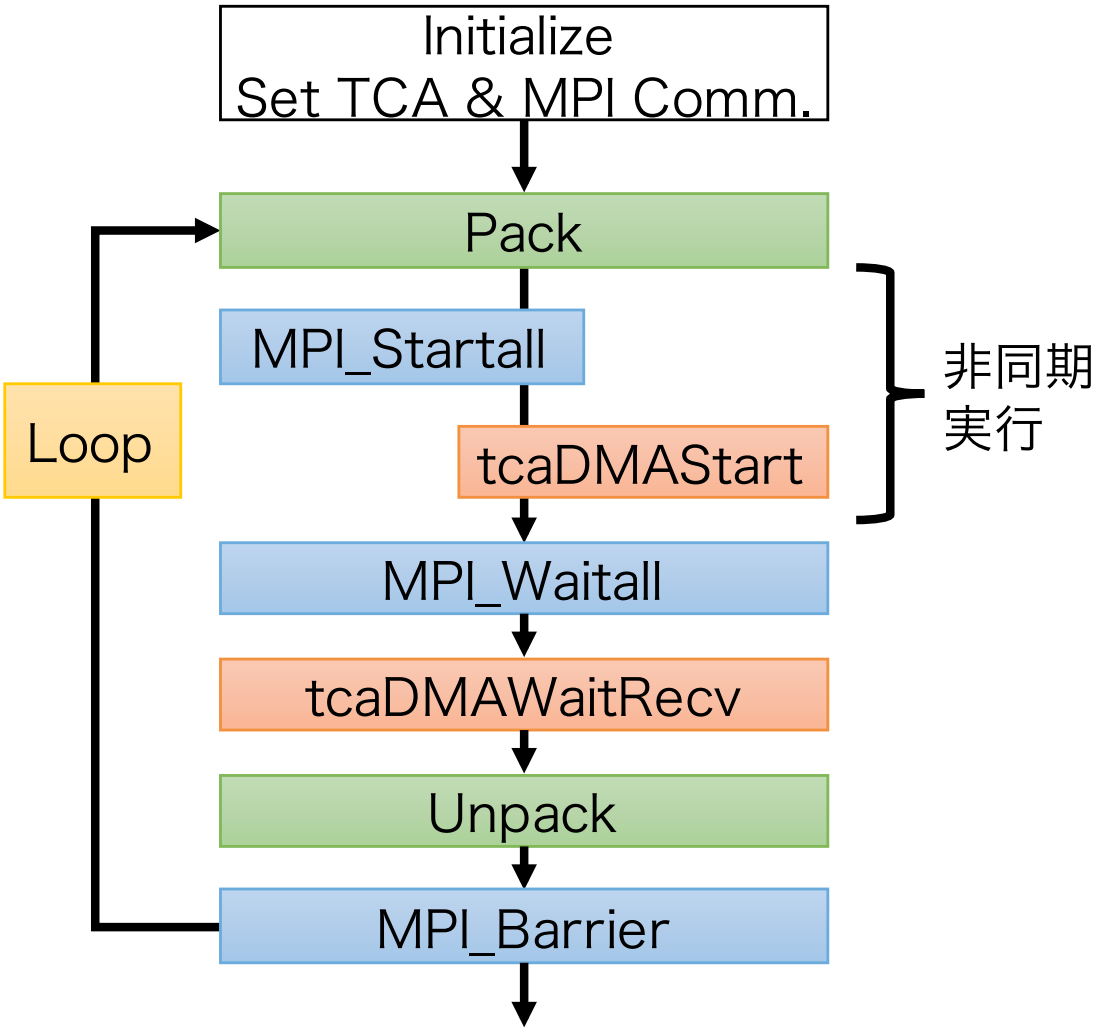


図 6.1　袖領域交換の流れ

Root プロセスである Rank0 から Rank4 にデータを転送する．データを受け取ったプロセスは，以降のステップでは Root プロセスのように他のプロセスにデータを転送する．Step 2 以降は TCA/PEACH2 による通信が可能な領域となり，本実装では Chained DMA 機構を用いて，DMA 通信を 1 回だけ発行することで通信が完了するようにしている．

次に，Allgather 通信について述べる．Allgather にはいくつかの通信アルゴリズムが存在するが，本論文では通信路を最大限利用するために「Recursive Doubling」アルゴリズムを用いる．図 6.3 に Allgather の通信アルゴリズムを示す．これより，自ノードとデータを交換するプロセスとの距離はステップが進むごとに倍になっていく．同様に，Recursive Doubling アルゴリズムでは，交換されるデータのサイズもステップが進むごとに倍になっていく．つまり，最初の数ステップはデータサイズが小さく，低レイテンシ通信が必要となり，後半のステップではデータサイズ非常に大きくなるため，高いバンド幅が必要になる．そこで本論文における実装では，ハイブリッド通信の性能を最大限に発揮するために，前半のステッ

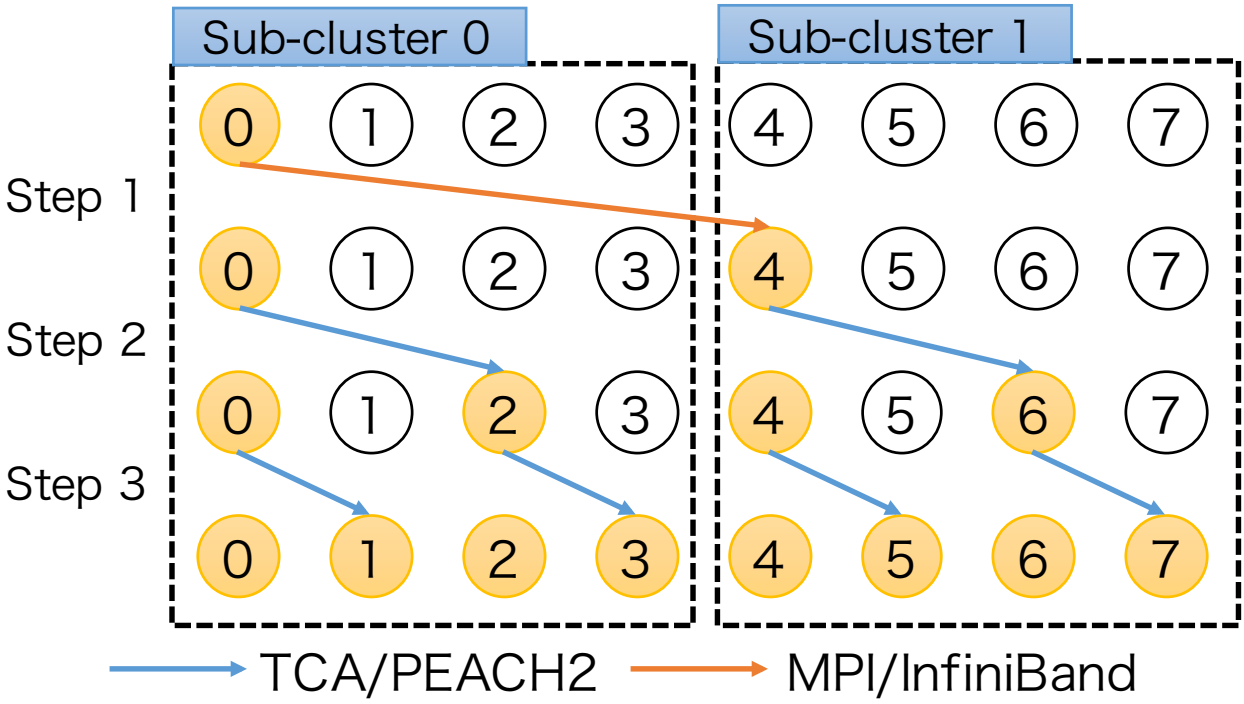


図 6.2　Broadcast 通信のアルゴリズム

プは TCA/PEACH2 による低レイテンシ通信，最終ステップのみ InfiniBand による高バンド幅通信を割り当てることとする．最終的には，InfiniBand によるサブクラスタを跨いだ通信によって，全てサブクラスタ内のプロセスとデータを交換することが出来る．

最後に，Allreduce 通信について述べる．本研究では，Allreduce の通信アルゴリズムとして Allgather と同様に Recursive Doubling アルゴリズムを用いる．Allreduce では，通信ステップが進むごとにデータを交換するプロセスとの距離は倍になるが，交換するデータは各ステップで常に同じという部分が Allgather とは異なる．また，Allreduce は全プロセスが持っているデータに対して演算を行い，その結果を全プロセスで共有する必要がある．Recursive Doubling アルゴリズムによる Allreduce は，通信ステップごとに演算が必要である．GPU 間で直接データを交換している場合，データがあるデバイスメモリ上で演算を行う必要があり，CUDA カーネルを何度も起動しなければならない．しかしながら，このカーネル起動のオーバヘッドは約 3.23μsec であり [55]，TCA/PEACH2 の DMA 通信では約 2.0μsec の最小レイテンシと比較して非常に大きいことがわかる．そのため，演算時間がオーバヘッドとなり，Allreduce 全体の時間が非常に大きくなってしまう．そこで，本実装では，以下のような順序で Allreduce 通信の実装を行う．

1. TCA/PEACH2 の DMA 通信を用いて，自ノード内のデバイスメモリからホストメモリにデータを転送する．
2. TCA/PEACH2 による PIO 通信でサブクラスタ内のホストメモリ間でデータの交換を行い，その後対応する演算を行う

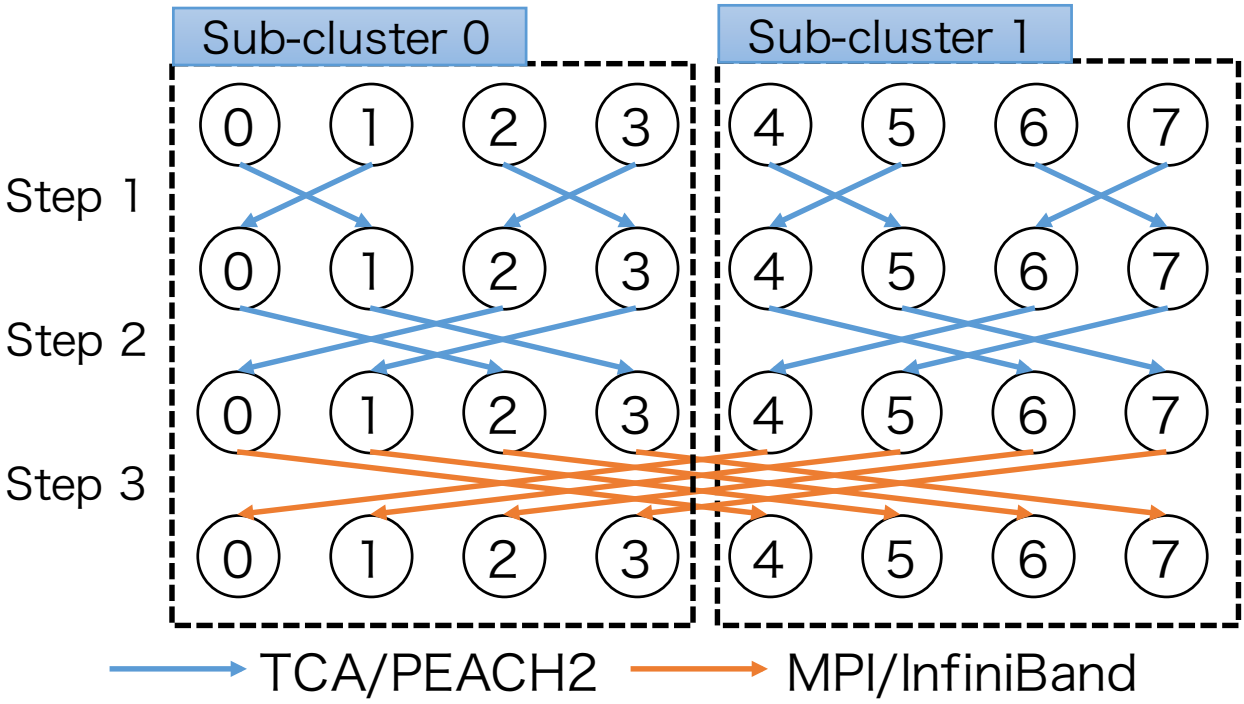


図 6.3　Allgather 通信のアルゴリズム

3. サブクラスタ間の通信には MPI_Isend，Irecv を用いてデータの交換を行い，2. と同様に対応する演算を行う
4. 最終的に自ノードのホストメモリからデバイスメモリに PEACH2 の DMA 通信を用いてデータの転送を行う

TCA/PEACH2 の小さなデータサイズにおける PIO の通信レイテンシは，DMA に対して非常に高速であり，最低レイテンシ（8 Byte まで）は約 1.0μsec[53] である．

6.1.3　XACC におけるハイブリッド通信の実装

本節では，XACC による袖領域交換を行うための指示文 **reflect_init** と **reflect_do** に対して，提案するハイブリッド通信を適用した実装について述べる．これらの指示文の概要について 3.2.1 節に示すとおりである．

reflect_init 指示文

reflect_init 指示文は，同期を行う袖領域を設定する指示文である．この指示文は，図 6.1 の Initialize フェイズに当たる．ハイブリッド通信では，前述のとおり，通信方向の特性に応じて TCA/PEACH2 または MPI の通信を割り当てる．そのために，XACC の言語処理系から reflect の対象とする配列の情報を手に入れ，各通信方向に対する行数，ブロック長，ギャップ長から通信パターンを判別する．リスト 6.1

に refect_init 指示文のランタイムの一部を示す.

リスト 6.1　reflect_init 指示文のランタイム拡張

```
1  xmp_nodes_index(adesc−>array_nodes, i+1, &dim_index);
2
3  if (count == 1) {
4    int target_lo_rank, target_hi_rank;
5
6    if (lo_rank != −1)
7      target_lo_rank = lo_rank;
8    else
9      target_lo_rank = MPI_PROC_NULL;
10
11   MPI_Recv_init(reflect−>lo_recv_array, width, MPI_BYTE, target_lo_rank,
12                 0, MPI_COMM_WORLD, &reflect−>req[0]);
13   MPI_Send_init(reflect−>lo_send_array, width, MPI_BYTE, target_lo_rank,
14                 1, MPI_COMM_WORLD, &reflect−>req[1]);
15   ...
16  }
17  else if (count > 1) {
18   size_t pitch = reflect−>stride;
19   size_t width = reflect−>blocklength;
20
21   if (widht <= _XMP_PACK_SIZE) { // Block Stride Communication
22     if (dim_index % 2 == 0) {
23       if (lo_rank != MPI_PROC_NULL) {
24         tcaSetDMADescInt_Memcpy(tca_reflect_desc, &h[lo_rank], 0,
25                                 &h[_XMP_world_rank], 0,
26                                 width, count, dma_flag, wait_slot, wait_tag);
27       }
28       if (hi_rank != MPI_PROC_NULL) {
29         tcaSetDMADescInt_Memcpy(tca_reflect_desc, &h[hi_rank], 0,
30                                 &h[_XMP_world_rank], 0,
31                                 width, count, dma_flag, wait_slot, wait_tag);
32       }
33     } else {
34       if (hi_rank != MPI_PROC_NULL) {
35         tcaSetDMADescInt_Memcpy(tca_reflect_desc, &h[hi_rank], 0,
36                                 &h[_XMP_world_rank], 0,
37                                 width, count, dma_flag, wait_slot, wait_tag);
38       }
39       if (lo_rank != MPI_PROC_NULL) {
40         tcaSetDMADescInt_Memcpy(tca_reflect_desc, &h[lo_rank], 0,
41                                 &h[_XMP_world_rank], 0,
42                                 width, count, dma_flag, wait_slot, wait_tag);
43       }
```

```
44      }
45    } else { // Stride Communication
46      if (dim_index % 2 == 0) {
47        if (lo_rank != MPI_PROC_NULL) {
48          tcaDescSetMemcpy2D(tca_reflect_desc, &h[lo_rank], lo_dst_offset, pitch,
49                                               &h[_XMP_world_rank], lo_src_offset, pitch,
50                                               width, count, dma_flag, wait_slot, wait_tag);
51        }
52        if (hi_rank != MPI_PROC_NULL) {
53          tcaDescSetMemcpy2D(tca_reflect_desc, &h[hi_rank], hi_dst_offset, pitch,
54                                               &h[_XMP_world_rank], hi_src_offset, pitch,
55                                               width, count, dma_flag, wait_slot, wait_tag);
56        }
57      } else {
58        ...
59      }
60    }
61  }
```

リスト 6.1 の 3 行目では，XACC 処理系から得られた通信方向の配列の行数が 1 であれば，それは連続領域となる．つまり，この通信方向には MPI の通信を割り当てることが最適である．6〜9 行目では，XMP のノードマッピング上において，自ノードよりも低い Rank のノードが存在するかどうか（周期境界条件でない時）の判定をし，11〜14 行目で lo_rank が存在するなら非同期の point-to-point の通信を設定する．ここでは省略をしているが，自ノードよりも高い Rank である hi_rank に対しても同様な判定を行い，point-to-point の通信を設定する．一方，17 行目では，通信方向の配列の行数が 1 より大きいかを判定し，21 行目でブロック長があるサイズより大きい場合，ブロックストライドであると判定する．本実装では，このしきい値をスカラー 1 要素としており，スカラー 1 要素であった場合 45 行目からの処理が実行される．*tcaDescSetMemcpy2D()* 関数は，TCA/PEACH2 のディスクリプタを設定する関数であり，ブロックストライドやストライド通信のような非連続通信の設定を行う．PEACH2 による通信では，同じ通信方向に異なる PEACH2 からのパケットが流れる時，または複数の PEACH2 から同時にパケットを受信した時にパケットの衝突が発生する．この衝突によって，通信路のバンド幅が低下することが知られている [56]．袖領域交換の通信パターンでは，後者の衝突が発生する可能性がある．そのため本実装では，一度に複数の PEACH2 からパケットを受信することがないように，通信するノードを pairwise して，ノードごとに DMA のディスクリプタを連結する順番を制御している．まず，1 行目の *xmp_nodes_index()* 関数を用いて，XMP のノードマッピング情報から，現在の通信面に対して自ノードが何番目であるかという情報を得る．これを用いて，22 行目で偶数番である場合 lo_rank に対する通信，hi_rank に対する通信の順番で DMA のディスクリプタを連結し，33 行目より奇数番目であった場合，hi_rank に対する通信，lo_rank に対する通信の順番で連結を行う．このようなフロー制御を行うことで，通信経路上におけるパケットの衝突を防ぐことが可能になる．一方，MPI 通信では常に InfiniBand のスイッチを経由して通信が行われるため，このような衝突は発生しない．45 行目からのストライド通信の設定においても，ブロックストライド通信同様にフロー制御を行う．ブロックストライド通信と異なり，

ストライド領域は通信の効率を上げるために，Pack/Unpack する必要がある．本実装では，**reflect_init** 指示文のランタイム内では Pack/Unpack は行わず，事前に Pack/Unpack 用のバッファに対する DMA ディスクリプタの設定を行う．

reflect_do 指示文

reflect_do 指示文は，同期通信を実際に実行するための指示文である．この指示文は，図 6.1 の Pack から **MPI_Barrier()** までのフェイズである．まず，ストライド通信がある場合，通信の前後に Pack/Unpack が実行される．Pack/Unpack 用のバッファは前述の reflect_init 指示文の中で確保している．そして，通信は **reflect_init** 指示文ですでに設定が完了しているので，**reflect_do** 指示文は，**MPI_Startall()** および **tcaDMAStart()** 関数で MPI の point-to-point 通信および TCA/PEACH2 の DMA 通信を開始する．これらは非同期通信であるため，異なる通信路を同時に利用することができ，MPI だけの通信に対して高いバンド幅を得ることが可能である．そして，**MPI_Waitall()** および **tcaDMAWaitRecv()** 関数でそれぞれの通信の完了を待つ．

6.2　性能評価

本節では，XACC によるハイブリッド通信の生産性とベンチマークによる通信性能の評価を行う．評価には，5.3 節と同様に HA-PACS/TCA を用い，評価環境は表 5.1 に示すとおりである．また，評価に用いるベンチマークは，2 次元ラプラス方程式および 3 次元姫野ベンチマークであり，それぞれのソースコード例を付録 A.1，A.2 に示す．OpenACC コンパイラとして，筑波大学の田渕らが開発している Omni OpenACC Compiler[39] を用いる．データの分割は 2 次元分割までとし，TCA/PEACH2 の通信がすべて 1 ホップで完了する分割の組み合わせとする．本論文におけるハイブリッド通信では，サブクラスタ内で仮想的にグループを分割することでサブクラスタを跨いだ通信を再現する．本評価ではデータサイズを固定し，分割方法およびノード数を変化させた強スケーリングを行う．本来は複数のサブクラスタを利用し，本当に TCA/PEACH2 通信が分離された状況で実験すべきであるが，実験環境の制約により，ここではサブクラスタが「仮想的に」さらに小さいサブクラスタで構成されている想定で，最大 16 ノードまでの実験を行う．本評価では，ラプラス方程式および姫野ベンチマークはそれぞれ 2 次元分割までとするため，分割（i, j, k）の k 次元方向は分割されず，常に 1 となる．そのため，ラプラス方程式における分割（4, 2）と姫野ベンチマークにおける分割（4, 2, 1）では同じプロセスマッピングとサブクラスタの分割になる．図 6.4 に分割（4, 2, 1）および分割（2, 8, 1）のプロセスマッピングと仮想サブクラスタの割り当て例を示す．図中の数字が Rank 番号，それぞれの色が仮想的に分割したサブクラスタ，点線は各分割パターンにおいて使用されないパスを示している．分割（i, j, k）において，j 次元が分割された場合 TCA/PEACH2 の通信（ラプラス方程式ではストライド通信，姫野ベンチマークではブロックストライド通信）が発生する．i 次元が分割された場合 MV2GDR によるブロック通信が発生し，この通信が仮想的なサブクラスタ間の通信となる．本論文の姫野ベンチマークでは k 次元は分割しないため，ストライド通信は発生しない．例えば，図 6.4 の分割（2, 8, 1）において，Rank 0 ～ 7 と Rank 8 ～ 15 はそれぞれ別の仮想サブクラスタに属し，この仮想サブクラスタ間では TCA/PEACH2 通信を一切行わない．分

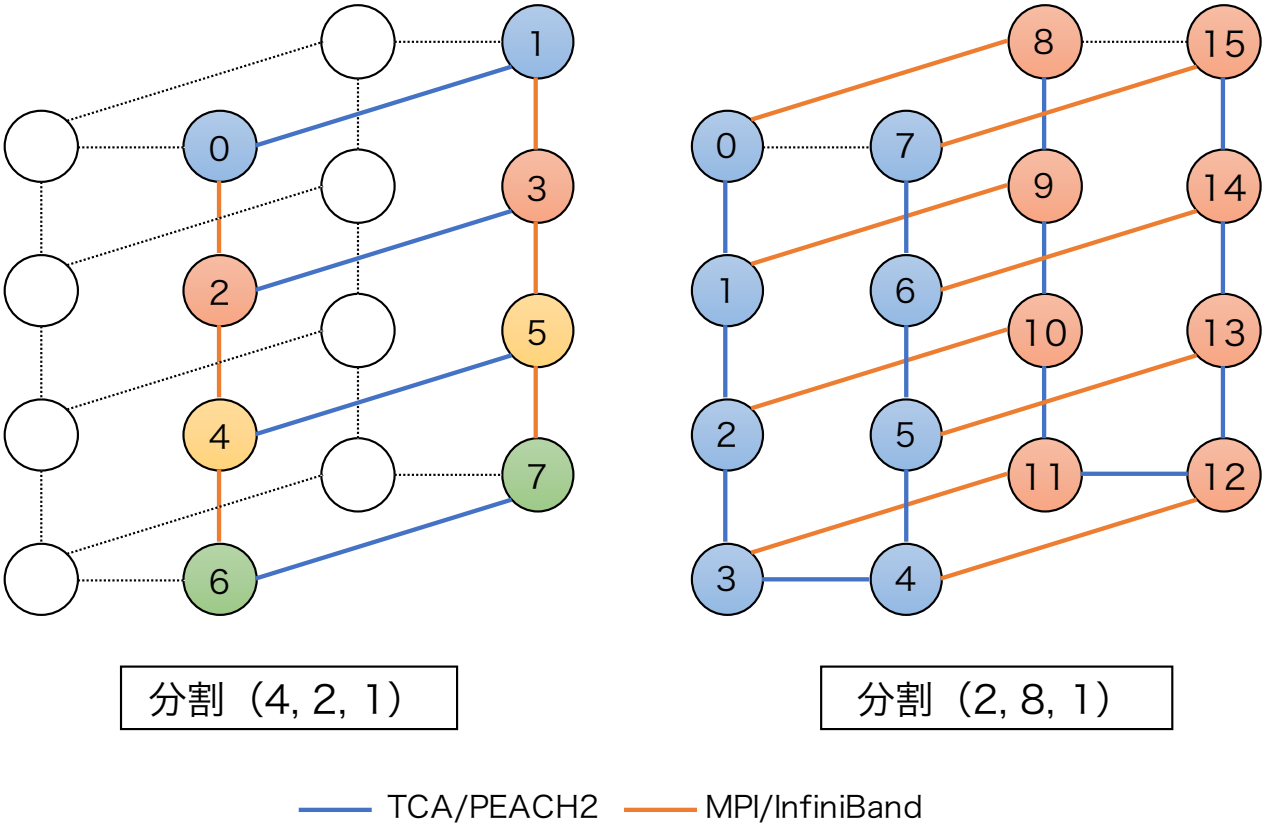


図 6.4　プロセスマッピングと仮想サブクラスタの分割

割（2, 8, 1）の Rank6 に着目すると Rank5 および 7 は TCA/PEACH2，Rank 14 は MV2GDR による袖領域交換が行われる．また，計算に使用する GPU は 5.3 章の基礎評価と同様に，TCA/PEACH2 およびハイブリッド通信では CPU0 側の GPU0，MV2GDR は CPU1 側の GPU2 とし，1 ノードにつき 1GPU のみを計算に用いる．

6.2.1　生産性の評価

XACC による TCA/InfiniBand ハイブリッド通信の生産性について評価を行う．GPU を搭載した分散メモリ環境において，TCA/PEACH2 のような GPU 間直接通信機構とコモディティネットワークである InfiniBand を組み合わせて利用するには，MPI だけでなく TCA/PEACH2 の独自の API を含めたプログラミングが必要である．特に，2 つのネットワークは通信の特性が異なるため，ユーザは通信の種類によって最適なネットワークを選択する必要もあり，プログラミングのコストが高くなる．そこで，XACC の通信ランタイムに本ハイブリッド通信を適用することで，プログラミングコストがどれほど減るか検証を行う．生産性の評価には，姫野ベンチマークを用い，逐次，Hand-coding (TCA/PEACH2+MPI+OpenACC)，XACC による記述を行い，SLOC を比較する．ハイブリッド通信における姫野ベンチマークでは，連続アクセス方向に MPI，ブロックストライドアクセス方向に TCA/PEACH2 の通信を割り当てた時の行数を示す．

表 6.1　姫野ベンチマーク：ソースコードの行数

	MPI	TCA	OpenACC	XMP	初期化部分	計算部分	合計
逐次					72	27	99
Hand-coding (MPI + TCA + OpenACC)	141	16	18		110	29	345
XACC			12	23	88	29	152

表 6.1 に姫野ベンチマークの行数を示す．4.5.1 節と同様に，MPI，TCA，OpenACC，XMP は，それぞれの指示文や API，インデックスの計算などの行数を示している．これより，Hand-coding の行数は逐次に対して 3.17 倍行数が増加していることがわかる．Hand-coding の行数が増えた要因として，ノード分割によるインデックスの計算や通信の設定が必要になったことが挙げられる．姫野ベンチマークで使用される通信の種類は，3 次元配列の袖領域交換通信とスカラーの Allreduce 通信である．Allreduce は **MPI_Allreduce()** を用いることで実行できるが，袖領域交換では，インデックスの計算の他に通信を行うノードの決定や配列の Pack/Unpack などの計算が必要になる．TCA/PEACH2 による通信では，TCA/PEACH2 独自のディスクリプタを用いて通信の設定を行う必要がある．これは MPI のプログラミングモデルとは異なる．加えて，MPI と TCA/PEACH2 の通信を配列の次元ごとに適宜選択しなければ，通信の性能を十分に活かすことができない．一方，XACC によるハイブリッド通信は，逐次に対して 1.54 倍の行数の増加にとどまっている．XACC によるハイブリッド通信のプログラム例を付録 A.2 に示しめす．XACC では，袖領域交換の通信は *#pragma xmp shadow* 指示文で指定した配列に対して，*#pragma xmp reflect_init (list) acc* で通信するノードの選択，TCA/PEACH2 と MPI の通信を設定し，*#pragma xmp reflect_do (list) acc* で実際の通信と同期を行うことが可能である．そのため，Hand-coding では煩雑であったインデックスの計算などを記述する必要が無いため，行数が大幅に少なくなった．Hand-coding と比較して，XACC では OpenACC の指示文の数が減っていることがわかる．これは，Hand-coding では通信の設定時にデバイスメモリ上のアドレスを指定するための指示文が必要であるが，XACC ではそれがランタイム内で直接デバイスメモリを指定しているためである．また，インデックスの計算を XACC のランタイム内で行うことで，MPI などの通信ライブラリを用いたプログラミングでよく起こるインデックス計算ミスなどのヒューマンエラーを未然に防ぐ効果もある．以上のことから，XACC によるハイブリッド通信は Hand-coding（MPI + TCA/PEACH2 + OpenACC）による記述に対して生産性が高いと言える．

6.2.2　XACC によるオーバヘッドの評価

本評価では，8192 × 8192 の 2 次元配列に対して，MPI+TCA/PEACH2 のハイブリッド通信を Hand-coding したプログラムと XACC で記述したプログラムを用いて，袖領域交換の設定（**reflect_init**）に要する時間と実際の通信時間（**reflect_do**）を比較する．**reflect_init** では，MPI の通信設定，TCA/-PEACH2 の通信設定およびディスクリプタの連結，Pack/Unpack 用のバッファの確保などを行う．また **reflect_do** では，**reflect_init** で設定した通信の起動および完了待ちを行う．袖領域交換の実行時間は 1000 イテレーションの平均を取り，それぞれ 10 回の平均時間を表 6.2 に示す．評価には HA-

PACS/TCA を 16 ノード用いて，図 6.4 の分割（2, 8, 1）と同様なプロセスマッピングとサブクラスタの分割を行う．

表 6.2　袖領域交換の実行時間 [μsec]

	reflect_init	reflect_do
Hand-coding	0.166	116.472
XACC	0.176	125.926

これより，**reflect_init**，**reflect_do** ともに実行時間に大きな差はないが，わずかに XACC の性能が低いことがわかる．**reflect_init** の時間は全プログラム実行では誤差となるため，**reflect_do** に着目すると，XACC におけるこの部分の実行時間が，Hand-coding に比べ 8.1% 増加している．これは，XACC では **reflect_do** が実行されるたびに，通信やその完了待ちをするかどうかの判定を各通信面に対して総当りで行う必要があるためだと推測される．しかし，全実行時間ではこれにさらに計算時間が追加されるため，全体としての性能低下は極めて低いと考えられ，記述の簡単化による生産性の向上を考えると十分に有効と言える．

6.2.3　ラプラス方程式の評価

2 次元ラプラス方程式の隣接ノード間における袖領域交換を TCA/PEACH2 のみ，MV2GDR（InfiniBand/MPI）のみ，ハイブリッド通信で行う．本評価の問題サイズとして Small（8192×8192），Large（16384×16384）の 2 種類を用いる．分割（i, j）は，（行方向，列方向）の分割を示し，例えば分割 2×4 の場合，分割されたデータサイズは Small で 4096×2048 となる．

図 6.5 と図 6.6 に 2 次元ラプラス方程式の実行性能を示す．グラフの縦軸は実行性能である GFLOPS 値，横軸は分割方法と使用したノード数である．2 つのグラフより，3 つの通信の性能差があまり大きくないことがわかる．これは，ノード間のデータ分割は 2 次元であるが，OpenACC によるスレッド分割が行方向の 1 次元しか行えない Omni OpenACC Compiler の制限による．ラプラス方程式の評価では，実行時間に対して通信時間は多くて一割程度にとどまっているが，今後 2 次元分割によりキャッシュが有効に使えることが期待できるため，計算時間が短くなり，実行時間全体に占める通信時間の割合が大きくなるため，通信性能の向上が全体の性能向上に大きく影響すると考えられる．

図 6.5 と図 6.6 より，ノード数が少ない場合は実行性能に顕著な差は無いが，問題サイズ Small，Large ともに 16 ノード分割を行った時に性能差が生まれていることがわかる．データサイズを固定し，ノード数を増やしていく強スケーリングでは，分割されたデータが徐々に小さくなっていき，それに伴い袖領域交換が必要なデータサイズも減少する．16 ノードを用いた場合，1 ノードの通信データサイズは Small で 64KB，Large で 128KB となる．このため，ある程度大きなデータを送るときに高いバンド幅を発揮する MPI 通信は，プロトコル変換などのオーバヘッドにより十分な性能を出すことができない．一方，TCA/PEACH2 通信は PCIe パケットを直接通信に利用することが可能なため，オーバヘッドが最小限であり，通信データサイズが小さい場合に効果的である．同様に，ハイブリッド通信は，MPI 通信に対して性能が向上していることがわかる．特に，図 6.5 の分割 2×8 において，ハイブリッド通信が MPI に対

して高速であることがわかる．行方向の分割数が 2 であるため，MV2GDR による通信は 1 方向のみで，その他の 2 方向に対しては TCA/PEACH2 が用いられる．この場合，MV2GDR による通信が 8KB で 1 回，TCA/PEACH2 による通信が 32KB で 2 回発生する．このように，通信データサイズが小さいため，ハイブリッド通信における TCA/PEACH2 は有効なデータサイズであり，ハイブリッド通信全体の性能向上が得られたと考えられる．

6.2.4　Broadcast の評価

本節では，ハイブリッド通信を適用した Broadcast 通信について MV2GDR の **MPI_Bcast()** 関数との性能比較を行う．評価には 8 ノードおよび 16 ノードを用いて，図 6.7 のように 2 つのサブクラスタに分割した環境で評価を行う．図 6.7 の数字は Rank 番号，各色のグループは仮想的に分割したサブクラスタを示している．仮想的に分割したサブクラスタ間では TCA/PEACH2 通信を一切行わないこととする．Broadcast では，TCA/PEACH2 および MPI の通信は常にブロック転送となり，各通信ステップにおける通信データサイズは常に一定である．Root プロセスは共に Rank0 とする．

図 6.8 および図 6.9 に，それぞれ 8 ノード，16 ノードにおける Broadcast の通信時間と，MV2GDR に対するハイブリッド通信の相対性能を示す．グラフでは通信時間の差が小さいように見えるが，左辺の縦軸が log スケールで表示されているためで，実際の通信時間の差は見た目以上に大きい．そこで，MV2GDR 通信に対するハイブリッド通信の相対性能を折れ線で表現し，グラフの右辺の縦軸に値を示す．相対性能が 1.0 より大きい場合は，ハイブリッド通信が MV2GDR に対して高速であることを示している．図 6.8 より，8 ノードの時は，MV2GDR に対して最大 21% の性能向上が得られている．しかし，$2KB \sim 8KB$ のデータサイズにおいて，相対性能が 1.0 より低くなっている．この原因として，MV2GDR の通信性能が影響していると考えられる．HA-PACS/TCA システムでは，PEACH2 および InfiniBand が接続されている側の GPU をそれぞれ通信の対象とすることで，最短経路による通信が可能になり，通信性能を最大限利用することができる．MV2GDR およびハイブリッド通信では，ともに MV2GDR/InfiniBand の通信を用いている．しかし，ハイブリッド通信は 5.3 節で述べたように，MV2GDR/InfiniBand の通信が QPI を経由しているため，QPI を経由していない MV2GDR/InfiniBand 通信とは性能が異なっている．さらに，ハイブリッド通信の内部で実行している MV2GDR 通信は，プロトコルスイッチを明示的に行っているため，QPI を経由しないデフォルトの MV2GDR と異なるタイミングで通信が切り替わる．このために，$2KB \sim 8KB$ においてハイブリッド通信の相対性能が低下したと考えられる．図 6.8 のデータサイズが 512KB より大きい時は，TCA/PEACH2 と MV2GDR の性能が逆転するデータサイズを超えてしまい，MV2GDR に対する TCA/PEACH2 の優位性が無いため相対性能が 1.0 を下回っている．

図 6.9 の 16 ノードの結果は，8 ノードの結果と同様な傾向であり，MV2GDR に対して最大 14% の速度向上が得られている．しかしながら，8 ノードの Broadcast に対して，16 ノードで得られる速度向上率が小さい．この原因として，TCA/PEACH2 通信のホップ数の増加が影響していると考えられる．16 ノードのハイブリッド通信を用いた Broadcast は 4 ステップ（$= \log_2 16$）の通信で構成されており，Step 1 ～ 3 は TCA/PEACH2 の通信，Step 4 は MV2GDR の通信となる．Step 1 ～ 2 の TCA/PEACH2 通信は隣接ノードへの DMA 転送となるが，Step 3 の TCA/PEACH2 通信は 2 つ先にノードへの DMA

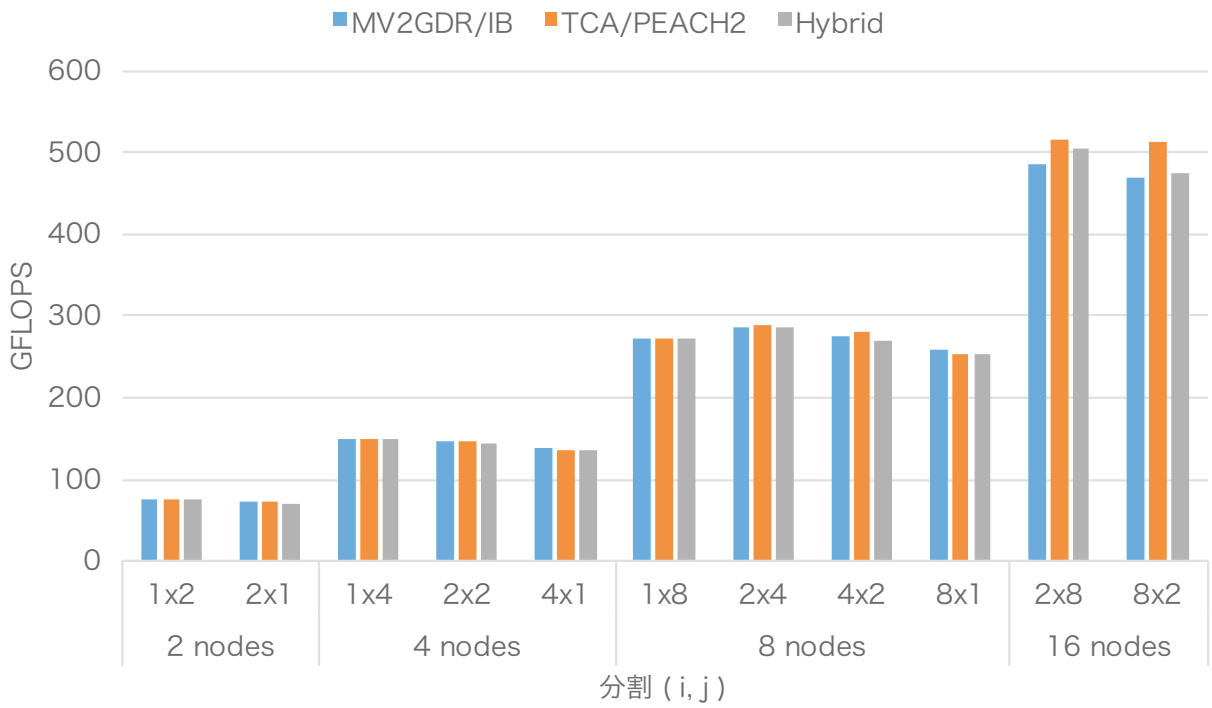


図 6.5　ラプラス方程式：データサイズ Small

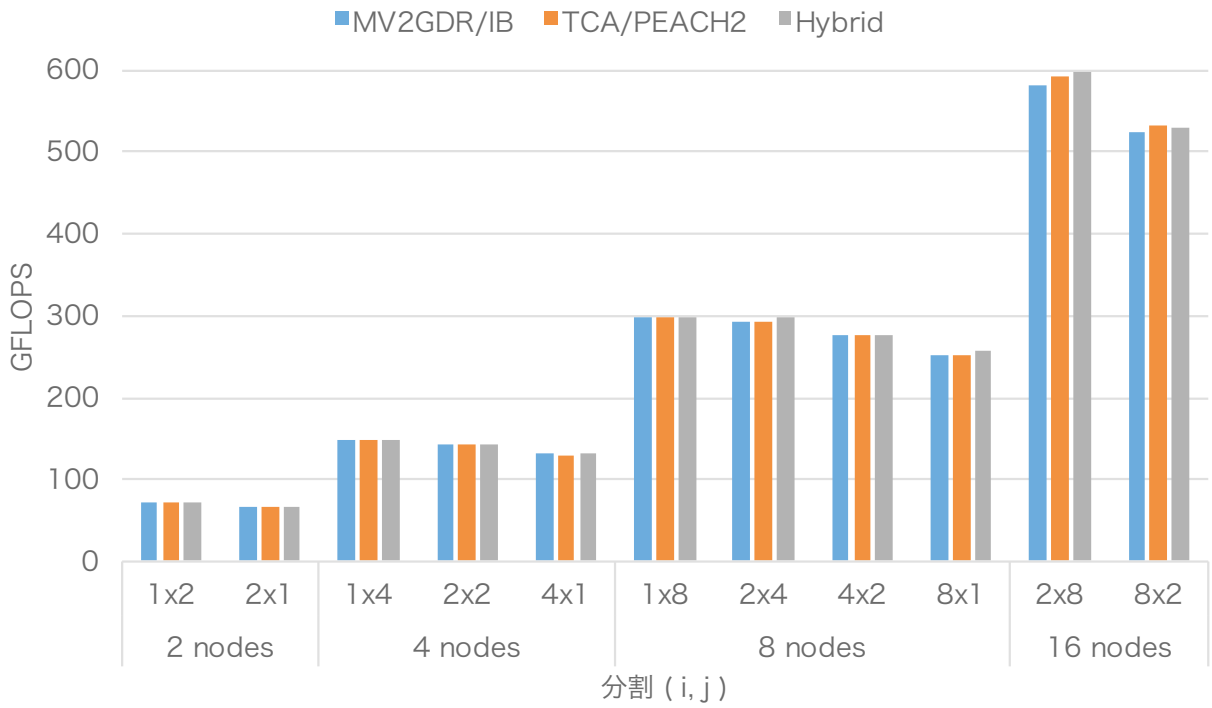


図 6.6　ラプラス方程式：データサイズ Large

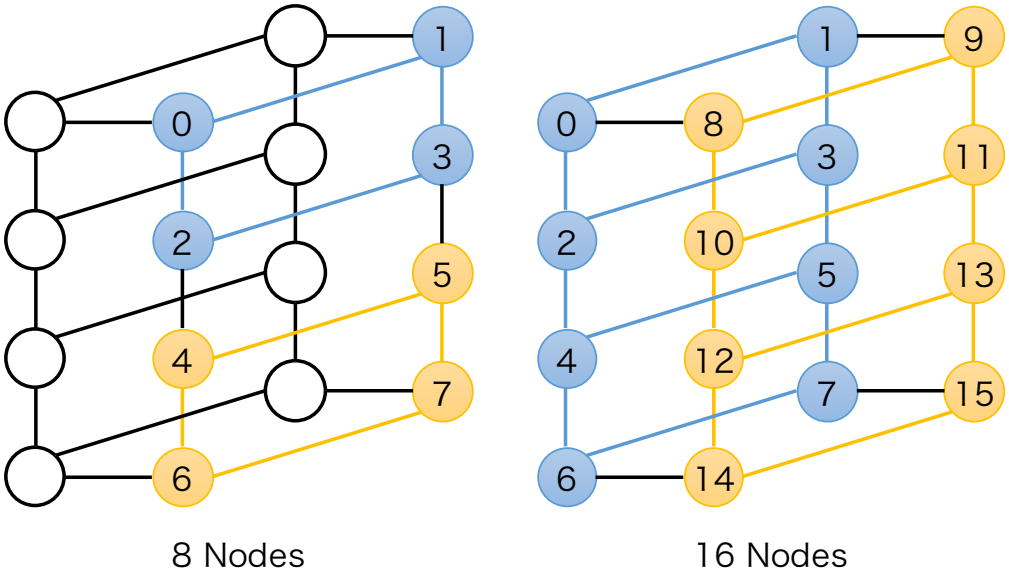


図 6.7　サブクラスタの分割とプロセスマッピング

転送となる．そのため，1 ホップにつき 0.3 ～ 0.4μsec 程度 TCA/PEACH2 のレイテンシが増加し，特にデータサイズが小さい時に性能低下を招く可能性がある．そこで，サブクラスタ間だけではなくサブクラスタ内で TCA/PEACH2 のホップ数が増加する通信を MV2GDR に割り当てることによって，より性能が向上する可能性がある．16 ノードよりも大きい環境ではさらにホップ数が増加するため，このような通信の割り当ての最適化がより重要になってくる．これについては今後の課題とする．

6.2.5　Allgather の評価

本節では，ハイブリッド通信を適用した Allgather 通信について MV2GDR の **MPI_Allgather()** 関数との性能比較を行う．Broadcast と同様に 8 ノードおよび 16 ノードを用いて，図 6.7 のように 2 つのサブクラスタに分割することを想定する．Allgather では通信ステップが増えるたびに通信データサイズは倍になり，最終的に全ノードで共有するデータサイズは「Send Size × Number of processes」となる．また，PEACH2 のハードウェアの制限により，ハイブリッド通信ではスカラー値の Allgather を実行できないため，Double Precision 2 要素（16 Byte）の Allgather から評価を行う．図 6.10 および図 6.11 に，それぞれ 8 ノードおよび 16 ノードにおける Allgather の通信時間と，MV2GDR に対するハイブリッド通信の相対性能を示す．グラフは，Broadcast と同様に左辺の縦軸が log スケールの通信時間，右辺の縦軸が相対性能を示している．相対性能は 1.0 より大きい場合は，ハイブリッド通信が MV2GDR に対して高速であることを示している．図 6.10 より，ハイブリッド通信は $1KB \sim 4KB$ を除き 32KB まで MV2GDR に対して高い性能を示しており，最大で 46% の性能向上が得られた．同様に図 6.11 より，ハイブリッド通信は $512B \sim 1KB$ を除き 16KB まで MV2GDR に対して高い性能を示し，最大で 40% の性能向上が得られた．$1KB \sim 4KB$ または $512B \sim 1KB$ で一時的に性能が低下する原因は，Broadcast と同様に MV2GDR の通信性能やプロトコルスイッチが関係していると考えている．8 ノードおよび 16 ノードにおいて，最終的に MV2GDR に対するハイブリッド通信の相対性能が 1.0 より小さくなるデータ

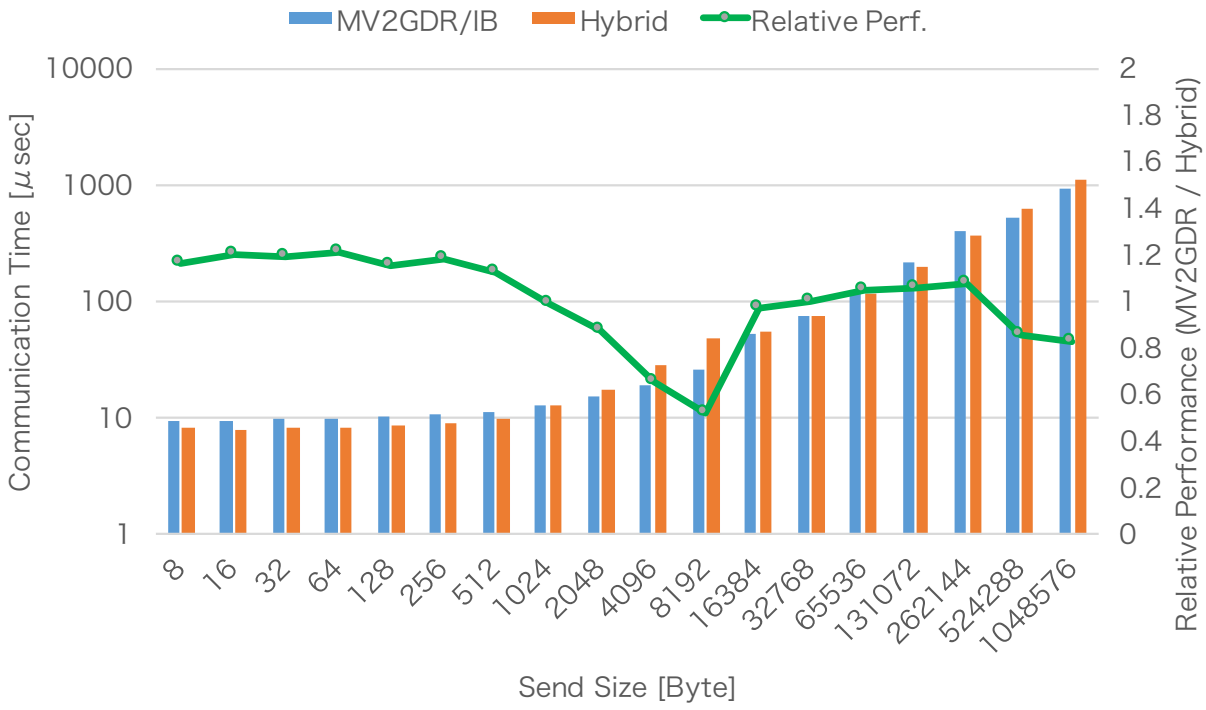


図 6.8　Broadcast：8 ノード

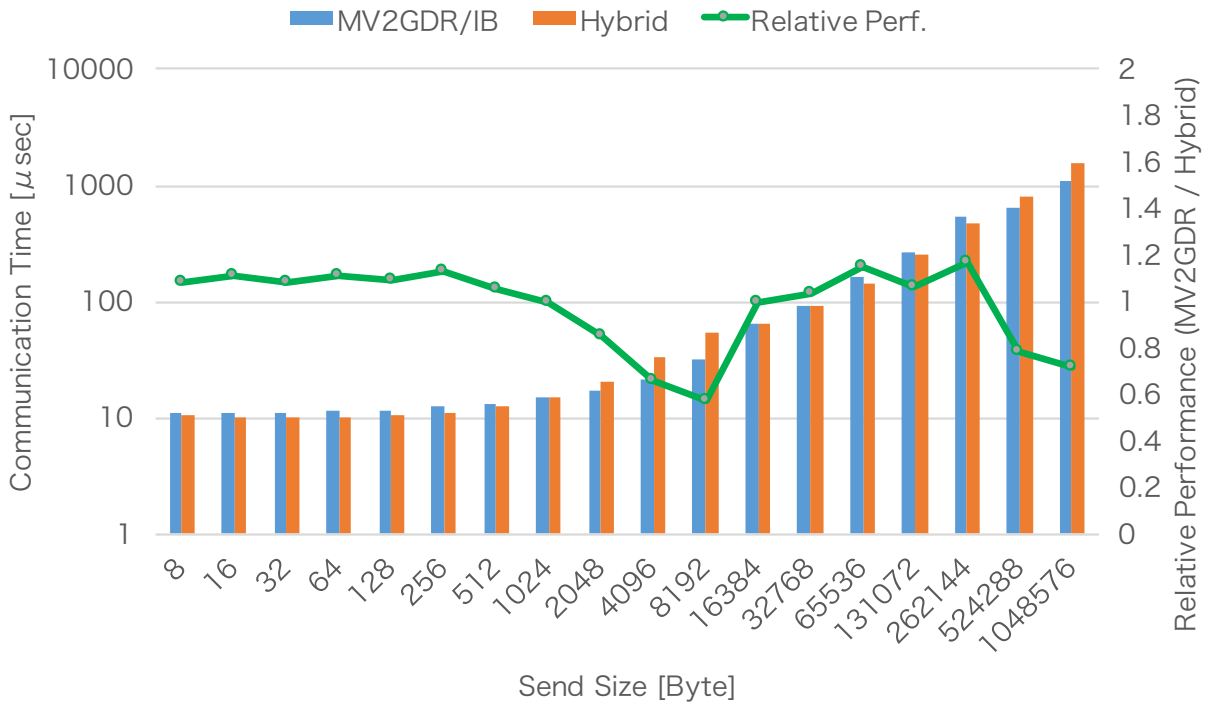


図 6.9　Broadcast：16 ノード

サイズはそれぞれ 64KB，32KB である．このとき，最終的に全プロセスで共有されるデータサイズはともに 512KB（$= 64KB \times 8nodes$，$32KB \times 16nodes$）となり，MV2GDR に対して TCA/PEACH2 に優位性があるデータサイズを超えてしまうため，相対性能が 1.0 より小さくなったと考えられる．

6.2.6　Allreduce の評価

本節では，ハイブリッド通信を適用した Allreduce 通信について MV2GDR の **MPI_Allreduce()** 関数との性能比較を行う．Broadcast，Allgather 同様に 8 ノードおよび 16 ノードを用い，サブクラスタの分割も図 6.7 であると想定する．また，本評価ではスカラーの精度浮動小数点数 float 型（4 Byte），演算は総和（SUM）による通信性能を比較する．スカラーの Allreduce は，Conjugate Gradient 法や姫野ベンチマークなど，HPC アプリケーションで一般的に用いられる通信である．Allreduce は通信ステップが増加するごとにデータを交換するノードとの距離は大きくなるが，データサイズは常に一定である．表 6.3 に，8 ノードおよび 16 ノードにおける Allreduce の通信時間と MV2GDR に対するハイブリッド通信の速度向上率を示す．Speedup が 1.0 より大きい場合，ハイブリッド通信は MV2GDR の通信に対して高い性能であることを示す．

表 6.3　Allreduce の性能 [μsec]

	MPI_Allreduce	Hybrid_Allreduce	Speedup (MPI / Hybrid)
8 ノード	18.322	13.390	1.36
16 ノード	22.835	16.007	1.42

表 6.3 より，8 ノード，16 ノードともにハイブリッド通信が MV2GDR の MPI_Allreduce に対して高い性能を示していることがわかる．特に Allreduce は，Broadcast および Allgather とは異なり，8 ノードよりも 16 ノードにおける通信性能が高い．スカラーの Allreduce は特にレイテンシが大きく影響する通信であり，MV2GDR/IB だけを用いた通信では TCA/PEACH2 に対して最低レイテンシが大きい．また，ノード数が増えると通信ステップ数が増加し，この通信レイテンシ性能の差がさらに加算されるため，ノード数が増えたときにより高い性能を示していると考えられる．

6.2.7　姫野ベンチマークの評価

3 次元姫野ベンチマークの隣接ノード間における袖領域交換およびスカラーの Allreduce を，MV2GDR（InfiniBand/MPI）のみ，ハイブリッド通信で行う．本評価の問題サイズは Large（$256 \times 256 \times 512$），Middle（$128 \times 128 \times 256$），Small（$64 \times 64 \times 128$）とする．問題の分割（i, j, k）は図 5.5 に対応している．本評価には，i 方向と j 方向を分割し，k 方向は分割せず常に 1 とする．6.1.2 節で述べたように，MPI と TCA/PEACH2 の通信特性から i 方向（ブロックアクセス）の通信を MV2GDR，j 方向（ブロックストライドアクセス）の通信を TCA/PEACH2 に割り当てる．i 方向の MV2GDR の通信がサブクラスタ間の通信に当たる．姫野ベンチマークの通信は，袖領域交換の他に各時間発展ループの最後に収束判定に必要な変数に対する Allreduce 通信を行う．本評価では，ハイブリッド通信には 6.2.6 節で用いたハイブリッド通信，MV2GDR には **MPI_Allreduce()** による通信を用いる．しかしながら，分割

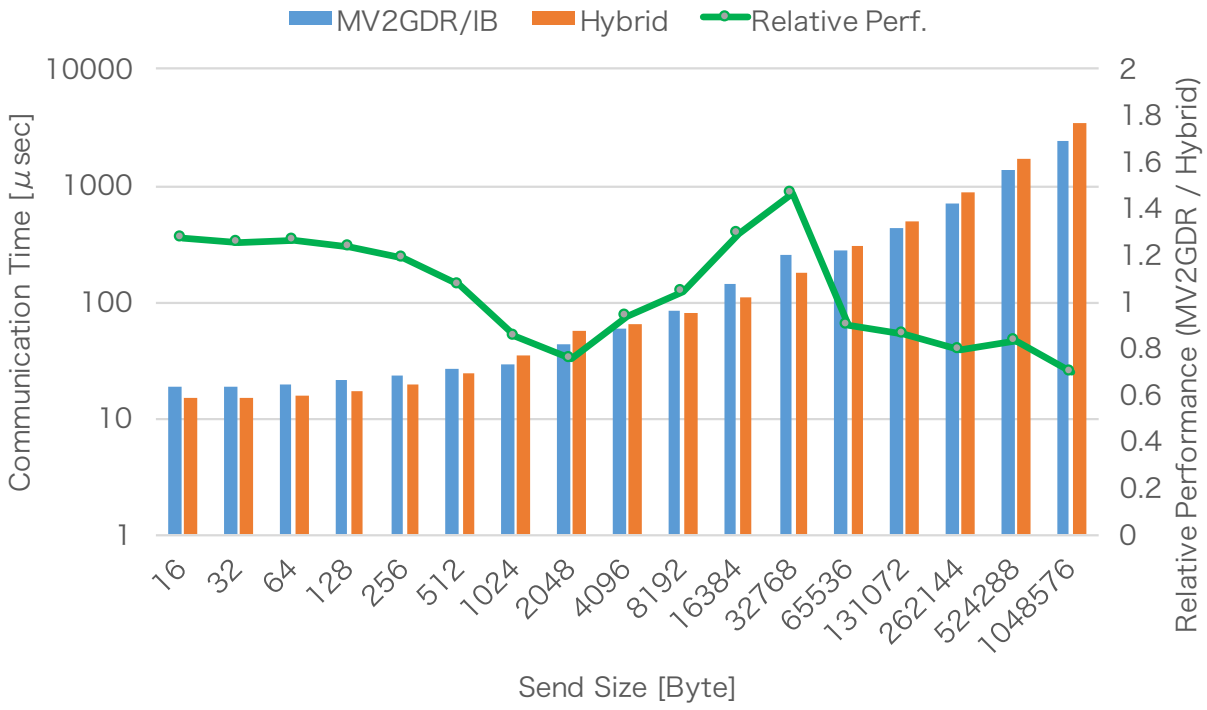


図 6.10　Allgather：8 ノード

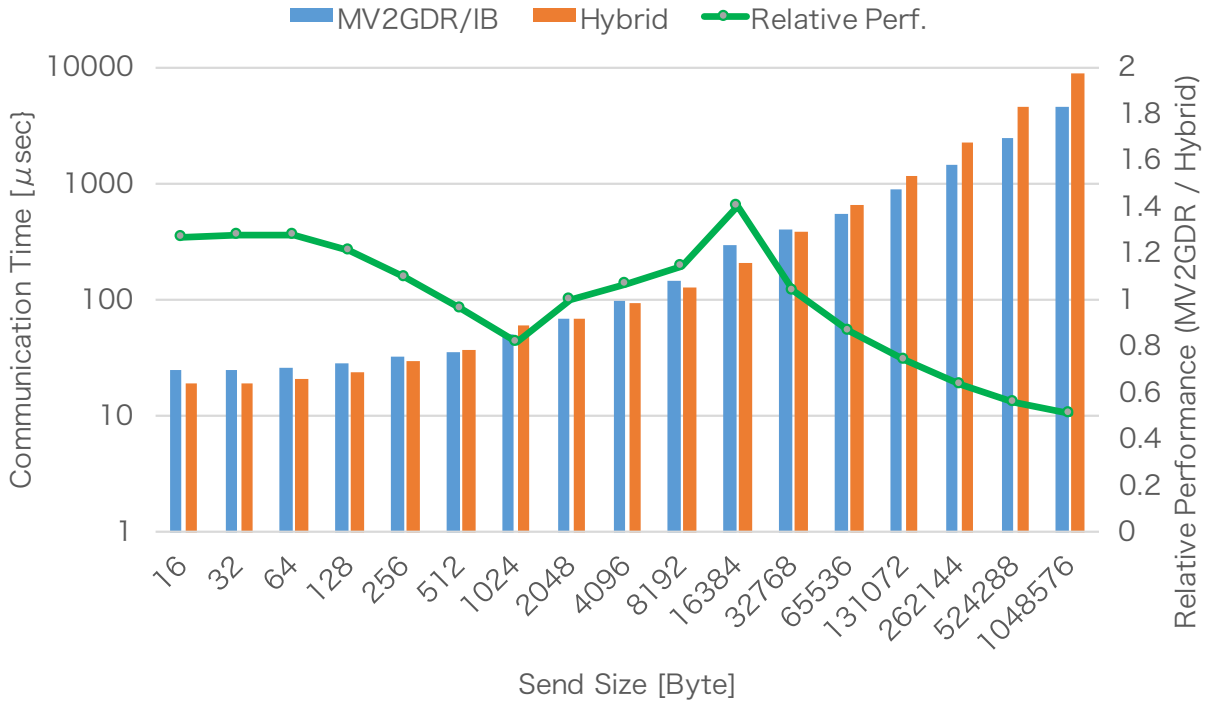


図 6.11　Allgather：16 ノード

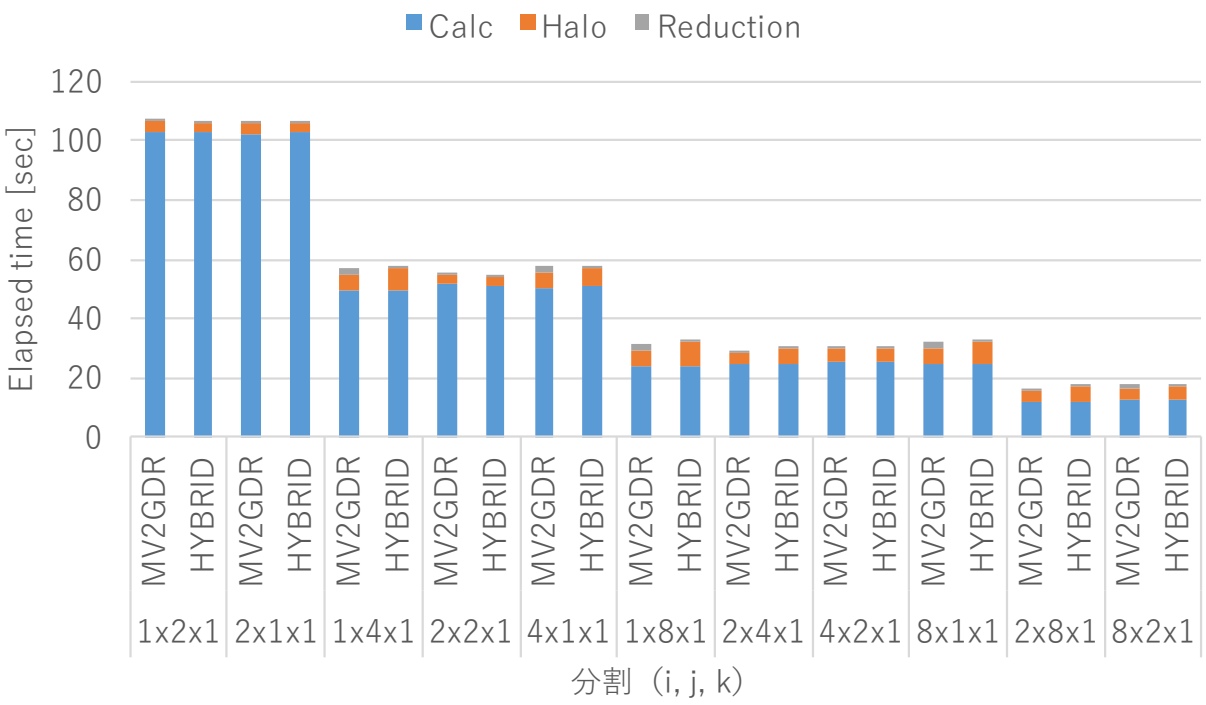


図 6.12　姫野ベンチマーク：データサイズ Large

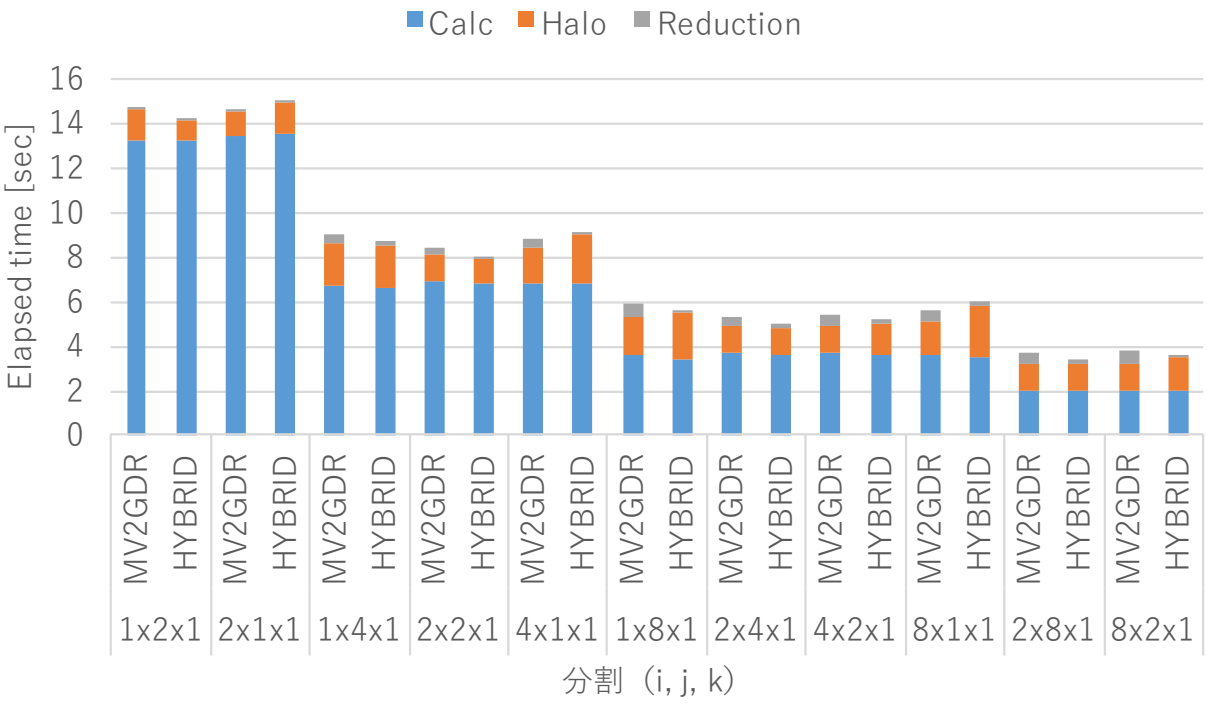


図 6.13　姫野ベンチマーク：データサイズ Middle

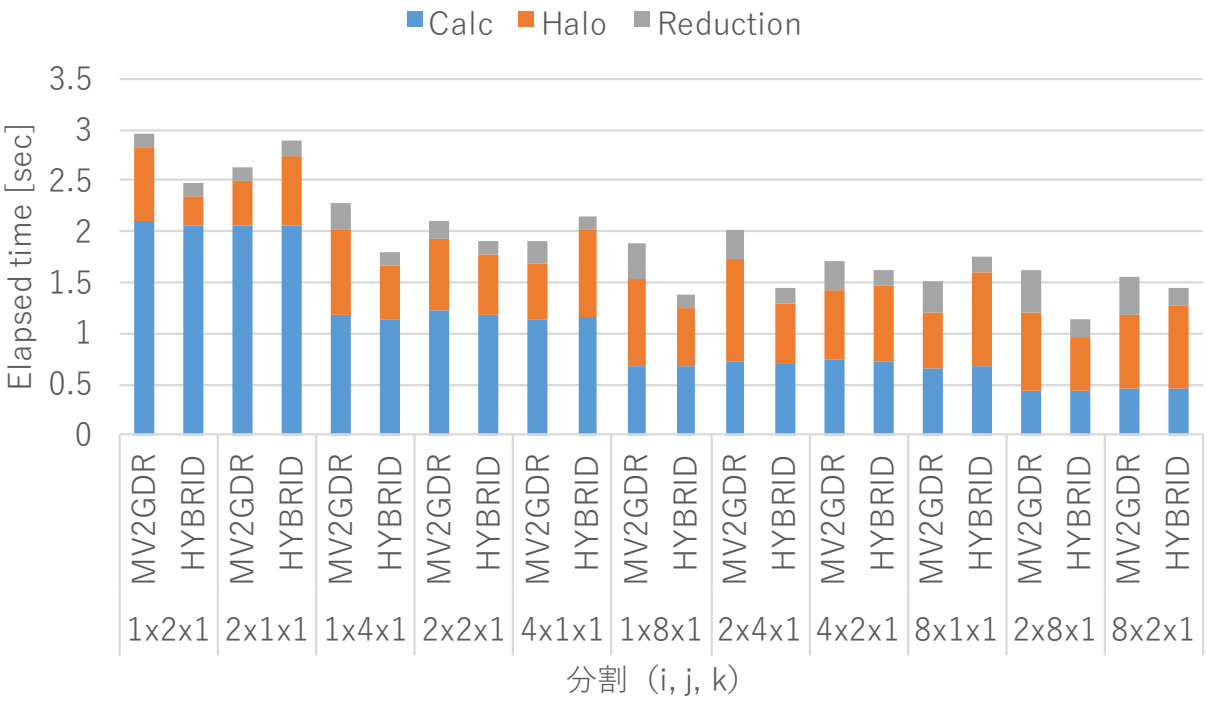


図 6.14　姫野ベンチマーク：データサイズ Small

（1 × 2 × 1）や（2 × 1 × 1）のように，2 ノードによる性能評価では，ハイブリッド通信を利用することができない．そのため，これらの分割では共通して MV2GDR の MPI_Allreduce() を用いる．

図 6.12，図 6.13，図 6.14 にそれぞれ問題サイズ Large，Middle，Small の実行時間を示す．それぞれのグラフの縦軸に 10000 イテレーションの実行時間の総和，横軸に分割方法を示す．3 次元配列の袖領域交換は，2 次元配列のそれと比べ通信データサイズが大きい．図 6.12 の問題サイズ Large では，通信データサイズが 512KB を超えている．通信データサイズ 512KB は，TCA/PEACH2 と MV2GDR の通信性能分岐点であり，これを超えたデータサイズではレイテンシよりもバンド幅が重要になり，TCA/PEACH2 の低レイテンシ通信のメリットを活かせない．そのため，TCA/PEACH2 のみの袖領域交換が行われる分割（1 × 4 × 1）や（1 × 8 × 1）において MV2GDR に対して実行時間が大きくなっている．このように，ハイブリッド通信は問題サイズ Large では最大で 2〜3% 程度の性能向上しか得られなかった．一方で，分割（4 × 1 × 1）や（8 × 1 × 1）におけるハイブリッド通信による袖領域交換は，MV2GDR の実行時間に対して大きくなっている．この原因として，ハイブリッド通信で必然的に発生する QPI 経由による DMA 通信が挙げられる．j 次元のみ分割では，ハイブリッド通信，MV2GDR ともに MPI の通信のみによる袖領域交換が行われる．5.3 節で述べたように，QPI を経由した DMA 通信による性能低下を最小限にするために，ハイブリッド通信ではプロトコルスイッチのタイミングをデフォルトの MV2GDR よりも早く切り替えている．しかし，QPI を経由した通信性能は経由していない通信よりも常に性能が低く，その結果として袖領域交換において性能差が生まれたと考える．

図 6.13 の問題サイズ Middle では，通信データサイズは大きくて 256KB にとどまり，TCA/PEACH2

が有効な範囲である．そのため，ハイブリッド通信では，多くの分割方法において MV2GDR に対して優位な性能を示していることがわかる．問題サイズ Large よりも全体の実行時間に占める計算時間の割合が小さくなったため，分割（$2 \times 8 \times 1$）において，9.5% の性能向上を達成した．

図 6.14 の問題サイズ Small は，通信サイズが最大で 64KB であり，分割（$2 \times 4 \times 1$）および（$2 \times 8 \times 1$）において，MV2GDR に対してハイブリッド通信は 41% の性能向上を達成した．この 2 つの分割方法に共通することは，全体の通信量のなかでブロックストライド通信が 8 ～ 9 割を占めていることである．MV2GDR のみの通信に対して，Pack/Unpack が必要ないこと，ブロックストライド通信が優位なデータサイズであることが影響し，優位な結果が得られたと考えられる．さらに，1 イテレーションの通信時間が小さいため，Reduction の実行時間の割合が大きくなっている．6.2.6 節で示したとおり，ハイブリッド通信による Allreduce の性能は非常に高い．このように，異なる通信に対してハイブリッド通信を適用することで，MPI/InfiniBand だけでは得られない速度向上を達成することが可能になる．

6.3　本章の結論

本章では, コモディティネットワークである InfiniBand と GPU 間直接通信機構である TCA/PEACH2 によるハイブリッド通信を提案した．ハイブリッド通信は，TCA/PEACH2 と InfiniBand で異なる通信特性に着目し，最適な通信を選択することで，MPI/InfiniBand だけでは得られない性能向上を実現する．本論文では，ステンシル計算における袖領域交換通信や，集団通信である Broadcast，Allgather，Allreduce 通信の実装を行った．さらに，ハイブリッド通信における最適な通信を選択することは非常に困難であるため，これを高並列言語である XACC の通信ランタイムに適用することで，生産性と通信性能の向上を目的としている．生産性の評価において，姫野ベンチマークを Hand-coding (MPI+OpenACC+TCA/PEACH2) で記述したプログラムに対して, XACC のプログラムは 48.41% のコード行数で記述することができることを示した．

ハイブリッド通信に関する研究を通じて，コモディティネットワークに加えて GPU 間直接通信機構を併用することで，高い通信性能を得ることができることを示した．一方で，今後，数万ノード以上からなるクラスタシステムを構築するにあたって，すべてのノードを InfiniBand だけで接続することはコストが非常に大きいことが問題となっている．そのため，本研究のようにグローバルとローカルという階層的ネットワークを持つことで，性能面とコスト面から大きなアドバンテージが得られると考える．しかしながら，本論文で示したように，複数の相互結合網によるハイブリッド通信のプログラミングコストは非常に大きい．そこで，高並列言語中にハイブリッド通信を適用することで，プログラミングコストの軽減と GPU 間通信の性能向上を両立することができる．これは，複数の相互結合網を搭載した次世代のクラスタシステムにおいても有効であると考えており，本研究における手法が適用可能である．

第 7 章

結論

本論文では，GPU クラスタにおける演算と通信について，高並列言語を用いて生産性とシステムの効率的利用の両立を実現するフレームワークを提案してきた．本章では，これまでの概要と今後との課題について述べる．

7.1 概略

GPU を代表とするアクセラレータが持つ高い演算性能やメモリバンド幅，電力あたりの性能に注目し，これらを搭載したスーパーコンピュータが HPC の分野で広く利用されている．しかしながら，アクセラレータを搭載したクラスタにおいて，システム中のアクセラレータおよび相互結合網を効率的に利用するためには，非常に複雑なプログラミングが必要であり，アプリケーションの生産性が低下してしまうことが問題とされている．

本論文では，これらの問題を解決するために，アクセラレータと高並列言語 XMP の拡張言語である XMP-dev ならびに XACC に着目し，XMP-dev のフレームワークの中で GPU と CPU によるワークシェアリング，XACC による InfiniBand と TCA/PEACH2 によるハイブリッド通信によって，高い生産性を維持しつつ，各リソースを効率的に利用することができるフレームワークを提供し，その有効性を示してきた．以降，本論文で得られた知見をまとめる．

第 4 章では，GPU クラスタにおける GPU と CPU によるワークシェアリングによって，演算器の最適化を検証した．本研究では，GPU へのデータの分散や並列実行を対象としている XMP-dev のバックエンドスケジューラとして，ランタイムスケジューラである StarPU を適用することで，マルチノードにおけるワークシェアリングを実行する XMP-dev/StarPU の提案と実装を行った．StarPU は，あるデータサイズに対して複数のタスクに分割を行い，それを GPU また CPU の空いている計算リソースに順次スケジュールすることで，各計算リソース間における負荷のバランスを調整する事が可能である．XMP-dev/StarPU のプロトタイプ実装では，各計算リソースの演算性能の違いから，タスクサイズとタスク数が最適でないと，負荷が片方のデバイスに集中してしまった．また，問題の性質やデータサイズ，実行環境によって最適な値は異なり，静的にこの値を決定することは非常に困難である．

そこで，本研究では，アプリケーションの実行中に GPU と CPU に割り当てるタスクサイズを変更するための適応型負荷分散機能を実現するために，XMP-dev の指示文の拡張やランタイムシステムを実装

した．ベンチマークによる評価では，問題サイズやノード数によって最適な値が異なることを確認した．また，この値を用いて，N 体問題および行列積による GPU のみの演算に対する性能を評価した．この結果，N 体問題において最大で 40% の性能向上を達成した．一方行列積では，分割された部分配列のサイズとキャッシュサイズの不整合が原因として考えられる性能低下を確認した．本論文では，非常に単純な収束アルゴリズムを用いていたが，キャッシュヒットなどを考慮できる値に調節するなど，ユーザが選択することも可能である．このように，GPU クラスタにおける GPU と CPU 間の負荷のバランスを適用的に変更する事が可能なフレームワークを構築することが示された．さらに，Hand-coding（MPI + StarPU）によるプログラムに対して，XMP-dev/StarPU のフレームワークによる記述では，33.62% のコード行数で記述可能であり，生産性の高さを示した．

第 6 章では，GPU 間のデータ通信について，コモディティネットワークの InfiniBand と密結合並列演算加速機構 TCA/PEACH2 によるハイブリッド通信の検証を行った．本研究では，提案する TCA/InfiniBand ハイブリッド通信では，TCA/PEACH2 と InfiniBand という異なるネットワークの特性に着目し，これを XACC の通信ランタイムに適用することで，通信の種類に応じて最適なネットワークの選択をランタイムで行う事ができる．

まず，TCA/PEACH2 と InfiniBand の基礎性能から，特に，TCA/PEACH2 は小データおよびブロックストライド通信性能が非常に高い，InfiniBand との性能が逆転するデータサイズが 512KB であることを確認した．これより，TCA/PEACH2 はブロックストライド，ストライド通信，InfiniBand にはブロック通信を割り当てるのが最適であり，最適な通信方向に対して各ネットワークを同時に利用することで，2 次元ラプラス方程式および 3 次元姫野ベンチマークを用いた評価では，最大 41% の性能向上を達成した．一方，集団通信に関して，Broadcast，Allgather および Allreduce 通信に対してハイブリッド通信による実装を行った．集団通信におけるハイブリッド通信では，袖領域交換通信とは異なり，同時にネットワークを使うよりもデータサイズに応じてネットワークを選択したほうが高い性能を得ることができる．本論文における集団通信は，通信を複数のステップに分割して，より近いノードから通信を行い，最終的に必要なデータが全ノードで共有される．Broadcast は各ステップにおける通信サイズは常に一定であるが，Allgather はステップが増えるたびに，通信するデータサイズは 2 倍になる．このため，前半のステップに TCA/PEACH2，後半のステップに InfiniBand を割り当てることで，InfiniBand のみの通信に対して高い性能を示している．対応する MPI の関数に対して，ハイブリッド通信の性能は，Broadcast で 21%，Allgather で 47% と非常に高いことが示された．Allreduce 通信は，他の集団通信と異なり一度ホストメモリにデータをコピーしてから通信を行う．TCA/PEACH2 は，DMA 通信の他に，CPU の store 命令を用いた PIO 通信を提供しており，これはホストメモリ間の通信のみを対象としているが，非常に低レイテンシである通信という特徴がある．他の集団通信と同様に測定したところ，スカラー（4 Byte）のハイブリッド通信による Allreduce は 42% の性能向上を達成している．このように，通信特性に合わせたネットワークの選択は，アプリケーションの性能向上には必要不可欠である．今後，NVLink などより高速なネットワークを搭載したシステムが一般的になると考えられるため，本研究のような通信の使い分けが重要であり，本研究の一部も適用可能であると考えている．さらに，高並列言語にハイブリッド通信を適用することで，3 次元姫野ベンチマークを Hand-coding（MPI + TCA/PEACH2 + OpenACC）で記述したプログラムに対して， 48.41% のコード行数で記述可能であり，生産性の高さが示された．

本研究を通じて，アクセラレータとして代表的な GPU を搭載したクラスタシステムにおける，高並列言語によるリソースの効率的利用について述べてきた．これより，本論文が提供する高並列言語によるプログラミングフレームワークを用いることで，GPU クラスタの演算加速器および相互結合網を効率的に利用でき，かつ，高い生産性を達成することを示した．

7.2　今後の課題

本論文で提案した，XMP-dev/StarPU では，CPU Weight という CPU に割り当てるデータの割合を示すパラメータを用いて，GPU と CPU 間のタスクサイズの調整を行ってきた．しかしながら，タスクを生成するときの分割数については常に固定で実行を行ってきた．そのため，分割数を変更するとデータサイズも変化してしまい，総合的なデータサイズの最適化が困難である．そのため，各デバイスに割り当てる「タスクのサイズ」というパラメータを設定することで，CPU Weight と分割数を計算することが出来るため，より最適化が行い易いと考えている．また，今回の性能評価ではノード間で等分割による分散を行ったが，データ分割を変更すると，ノード間で計算量が変化する可能性もある．しかしながら，ノード内における GPU/CPU の負荷バランスは XMP-dev/StarPU の適応型負荷分散機能を用いて，ノード間のデータ分割を XMP に搭載されている gblock 機能を用いることで，同時にノード間の負荷バランスのチューニングも可能であると考える．

XACC によるハイブリッド通信について，本論文は TCA/PEACH2 と InfiniBand という実験的な環境での GPU 間通信の検証を行った．今後は，NVIDIA 社が提案している NVLink と InfiniBand を組み合わせた環境が一般的に利用されると想定される．そのため，各ネットワークの特徴を把握し，それに応じたネットワークを選択する必要がある．これは，本論文で示してきた手法が一部適用可能であると考えており，実際のシステムにおいての実装および評価を行いたいと考えている．

本論文では，演算および通信について個別に最適化を行ってきたが，それぞれは独立したシステムであり，これらを一つのフレームワークで提供したほうが，より高い生産性が得られると考えている．本研究で用いた XMP-dev は，XACC を開発するにあたっての前身のフレームワークであり，今後 XACC のフレームワーク上で通信だけでなく，ワークシェアリングに基づく演算の最適化行える機構を XACC に導入したいと考えている．ワークシェアリングとハイブリッド通信は独立した技術であるが，対象とする問題によっては，データの依存関係などからワークシェアリング中に GPU 間ハイブリッド通信を実行できないことが考えられる．そのような時には，一度ホストメモリ上にデータを上げ，それをハイブリッド通信の対象とすることで，両立することができると考えている．

参考文献

[1] NVIDIA Tesla. http://www.nvidia.com/object/tesla-supercomputing-solutions.html.

[2] Intel Many Integrated Core Architecture. view-source:http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[3] Top500 supercomputing sites. http://www.top500.org/.

[4] The Green500 List. http://www.green500.org/.

[5] *CUDA C Programming Guide.* https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[6] OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[7] 大島聡史, 吉瀬謙二, 片桐孝洋, 弓場敏嗣. CPU と GPU を用いた並列 GEMM 演算の提案と実装. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 47, No. SIG12(ACS15), pp. 317–328, Sep. 2006.

[8] 遠藤敏夫, 松岡聡, 橋爪信明, 長坂真路. ヘテロ型スーパコンピュータ TSUBAME の Linpack による性能評価. 情報処理学会論文誌コンピューティングシステム, Vol. 48, No. SIG8(ACS18), pp. 62–70, May 2007.

[9] 遠藤敏夫, 額田彰, 松岡聡, 丸山直也, Hideyuki Jitsumoto. 異種アクセラレータを持つヘテロ型スーパーコンピュータ上の Linpack の性能向上手法. 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2009-HPC-121, No. 24, pp. 1–8, Jul. 2009.

[10] NVIDIA Corporation. NVIDA GPUDirect. https://developer.nvidia.com/gpudirect.

[11] InfiniBand Trade Association.

[12] MVAPICH2: A High Performance MPI Library for NVIDIA GPU Clusters with InfiniBand. http://on-demand.gputechconf.com/gtc/2013/presentations/S3316-MVAPICH2-High-Performance-MPI-Library.pdf.

[13] HPCI 技術ロードマップ白書 2012 年 3 月. http://open-supercomputer.org/wp-content/uploads/2012/03/hpci-roadmap.pdf.

[14] 塙敏博, 児玉祐悦, 朴泰祐, 佐藤三久. Tightly Coupled Accelerators アーキテクチャに基づく GPU クラスタの構築と性能予備評価. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 6, No. 4, pp. 14–25, Oct. 2013.

[15] 塙敏博, 児玉祐悦, 藤井久史, 朴泰祐, 佐藤三久. HA-PACS/TCA システムにおけるマルチノード GPU 間通信性能評価. 情報処理学会研究報告計算機アーキテクチャ（ARC）, Vol. 2014-ARC-208, No. 20, pp. 1–8, Jan. 2014.

[16] T. Hanawa, Y. Kodama, T. Boku, and M. Sato. Interconnection Network for Tightly Coupled Accelerators Architecture. In *IEEE 21st Annual Symposium on High-Performance Interconnects (HOT Interconnects 21)*, pp. 79–82, Aug. 2013.

[17] T. Hanawa, Y. Kodama, T. Boku, and M. Sato. Tightly Coupled Accelerators Architecture for Minimizing Communication Latency among Accelerators. In *The Third International Workshop on Accelerators and Hybrid Exascale Systems (AsHES) in conjunction with IEEE 27th International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1030–1039, May 2013.

[18] NVIDIA NVLINK HIGH-SPEED INTERCONNECT. `http://www.nvidia.com/object/nvlink.html`.

[19] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pp. 45–55, 2009.

[20] 荒木拓也, 村井均, 蒲池恒彦, 妹尾義樹. データ並列言語を対象とした動的負荷分散機構の実現と評価. 情報処理学会論文誌ハイパフォーマンスコンピューティングシステム, Vol. 43, No. 5, pp. 66–76, Sep. 2002.

[21] PGI Accelerator Compiler. http://www.softek.co.jp/SPG/Pgi/Accel/index.html.

[22] HMPP Workbench. http://www.caps-entreprise.com/hmpp.html.

[23] 李珍泌, チャン トゥアンミン, 小田嶋哲哉, 朴泰祐, 佐藤三久. PGAS 並列プログラミング言語 XcalableMP における演算加速装置を持つクラスタ向け拡張仕様の提案と試作. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 2011, No. 2, pp. 33–50, Mar. 2012.

[24] M. Nakao, J. Lee, T. Boku, and M. Sato. Productivity and Performance of Global-View Programming with XcalableMP PGAS Language. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, CCGRID '12, pp. 402–409, May 2012.

[25] S. Tomov, R. Nath, P. Du, and J. Dongarra. *MAGMA version 0.2 User Guide*. http://icl.cs.utk.edu/magma/.

[26] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. Faster, Cheaper, Better a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In *GPU Computing Gems*, Vol. 2, Sep. 2010.

[27] R. Ammendola, A. Biagioni, O. Frezza, F. Lo Cicero, A. Lonardo, P. S. Paolucci, D. Rossetti, A. Salamon, G. Salina, F. Simula, L. Tosoratto, and P. Vicini. APEnet+: high bandwidth 3D torus direct network for petaflops scale commodity clusters. *Journal of Physics: Conference Series*, Vol. 331, No. 5, p. 052029, 2011.

[28] R. Ammendola, M. Bernaschi, A. Biagioni, M. Bisson, M. Fatica, O. Frezza, F. Lo Cicero,

A. Lonardo, E. Mastrostefano, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini. GPU Peer-to-Peer Techniques Applied to a Cluster Interconnect. In *3rd Workshop on Communication Architecture for Scalable Systems (CASS2013) in conjunction with IPDPS*, pp. 806–815, May 2013.

[29] Mellanox Technologies: Mellanox OFED GPUDirect. `http://www.mellanox.com/page/products_dyn?product_family=116`.

[30] XcalableMP Specification Working Group. XcalableMP Specification Version 1.2.1. `http://www.xcalablemp.org/download/spec/xmp-spec-1.2.1.pdf`.

[31] 李珍泌, 朴泰祐, 佐藤三久. 分散メモリ向け並列言語 XcalableMP コンパイラの実装と性能評価. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 3, No. 3, pp. 153–165, Sep. 2010.

[32] J. Lee and M. Sato. Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems. In *Third International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, pp. 413–420, Sep. 2010.

[33] Omni OpenMP compiler project. `http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/index.html`.

[34] T. Nomizu, D. Takahashi, J. Lee, T. Boku, and M. Sato. Implementation of XcalableMP Device Acceleration Extention with OpenCL. In *Multicore and GPU Programming Models, Languages and Compilers Workshop (PLC 2012)*, pp. 2394–2403, May 2012.

[35] 田渕晶大, 村井均, 朴泰祐, 佐藤三久. XcalableMP と OpenACC の統合による GPU クラスタ向け並列プログラミングモデル. 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2014-HPC-145, No. 39, pp. 1–7, Jul. 2014.

[36] 中尾昌広, 村井均, 下坂健則, 田渕晶大, 塙敏博, 児玉祐悦, 朴泰祐, 佐藤三久. XcalableACC:OpenACC を用いたアクセラレータクラスタのための PGAS 言語 XcalableMP の拡張. 情報処理学会研究報告 (ハイパフォーマンスコンピューティング), Vol. 2014-HPC-146, No. 7, pp. 1–11, Sep. 2014.

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

[38] Omni Compiler Project. Omni OpenACC Compiler. `http://omni-compiler.org/openacc.html`.

[39] A. Tabuchi, M. Nakao, and M. Sato. A Source-to-Source OpenACC Compiler for CUDA. In *11th The International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar2013) in conjunction with Euro-Par2013*, Vol. 8374, pp. 178–187, 2013.

[40] 小田嶋哲哉, 李珍泌, 朴泰祐, 佐藤三久, 塙敏博, 児玉祐悦, Raymond Namyst, Samuel Thibault, Olivier Aumage. GPU クラスタにおける GPU/CPU ハイブリッド・プログラミング環境. 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, No. 9, pp. 1–8, Jul. 2012.

[41] 小田嶋哲哉, 朴泰祐, 塙敏博, 児玉祐悦, Raymond Namyst, Samuel Thibault, Olivier Aumage, 佐藤三久. GPU クラスタ向け並列言語 XMP-dev における GPU/CPU 動的負荷分散機能. 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（HPCS 2013）論文集, Vol. 2014, pp. 87–96, Dec. 2013.

[42] T. Odajima, T. Boku, M. Sato, T. Hanawa, Y. Kodama, R. Namyst, S. Thibault, and O. Aumage. Adaptive Task Size Control on High Level Programming for GPU/CPU Work Sharing. In *The 2013 International Symposium on Advances of Distributed and Parallel Computing (ADPC 2013)*, pp. 59–68, Dec. 2013.

[43] E. Cesar, M. Alexander, A. Streit, J. Traff, C. Cerin, A. Knupfer, D. Kranzlmuller, and S. Jha. A Unified Runtime System for Heterogeneous Multi-core Architecturesq. In *Euro-Par 2008 Workshops - Parallel Processing*, Vol. 5415, pp. 174–183, 2009.

[44] C. Augonnet, S. Thibault, and R. Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. In *Concurrency Computat.: Pract. Exper*, Mar. 2010.

[45] 筑波大学計算科学研究センター. HA-PACS ベースクラスタ. `http://www.ccs.tsukuba.ac.jp/research/research_promotion/project/ha-pacs/cluster`.

[46] 朴泰祐, 佐藤三久, 塙敏博, 児玉祐悦, 高橋大介, 建部修見, 多田野寛人, 藏増嘉伸, 吉川耕司, 庄司光男. 演算加速装置に基づく超並列クラスタ HA-PACS による大規模計算科学. 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2011-HPC-130, No. 21, pp. 1–7, Jul. 2011.

[47] Texas Advanced Computing Center - GotoBlas2. http://www.tacc.utexas.edu/tacc-projects/gotoblas2.

[48] T. Kuhara, T. Kaneda, T. Hanawa, Y. Kodama, T. Boku, and H. Amano. A Preliminarily Evaluation of PEACH3: A Switching Hub for Tightly Coupled Accelerators. In *Computing and Networking (CANDAR)*, pp. 377–381, Dec 2014.

[49] 天野英晴, 久原拓也, 塙敏博, 児玉祐悦, 朴泰祐. PEACH3 の基本転送性能の予備評価. 電子情報通信学会技術研究報告（コンピュータシステム）, Vol. 114, No. 155, pp. 97–102, Jul. 2014.

[50] Altera Corporation. Stratix IV Device Handbook. `http://www.altera.co.jp/literature/lit-stratix-iv.jsp`.

[51] MVAPICH2-GDR 2.2a Userguide. `http://mvapich.cse.ohio-state.edu/userguide/gdr/2.2a/`.

[52] 小田嶋哲哉, 朴泰祐, 塙敏博, 児玉祐悦, 村井均, 中尾昌広, 田渕晶大, 佐藤三久. アクセラレータ向け並列言語 XcalableACC における TCA/InfiniBand ハイブリッド通信. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 8, No. 4, pp. 61–77, Nov. 2015.

[53] 松本和也, 塙敏博, 児玉祐悦, 藤井久史, 朴泰祐. 密結合並列演算加速機構 TCA による GPU 間直接通信における Collective 通信の実装と性能評価. 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 8, No. 4, pp. 36–49, Nov. 2015.

[54] K. Matsumoto, T. Hanawa, Y. Kodama, H. Fujii, and T. Boku. Implementation of CG Method on GPU Cluster with Proprietary Interconnect TCA for GPU Direct Communication. In *The Fifth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES 2015)*, pp. 647–655, May 2015.

[55] 桑原悠太, 塙敏博, 朴泰祐. GMPI：GPU クラスタにおける GPU セルフ MPI の提案. 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2015-HPC-151, No. 12, pp. 1–8, Sep. 2015.

[56] 藤井久史, 塙敏博, 児玉祐悦, 朴泰祐, 佐藤三久. GPU 向け FFT コードの TCA アーキテクチャによる実装と性能評価. 情報処理学会研究報告（ハイパフォーマンスコンピューティング）, Vol. 2015-HPC-148, No. 12, pp. 1–9, Feb. 2015.

付録 A

サンプルプログラム

リスト A.1　XACC による 2 次元ラプラス方程式のプログラム例

```
1  double u[XSIZE][YSIZE], uu[XSIZE][YSIZE];
2
3  #pragma xmp nodes p(YNODES, XNODES)
4  #pragma xmp template t(0:YSIZE−1, 0:XSIZE−1)
5  #pragma xmp distribute t(block, block) onto p
6  #pragma xmp align u[i][j] with t(j, i)
7  #pragma xmp align uu[i][j] with t(j, i)
8  #pragma xmp shadow u[1:1][1:1]
9  #pragma xmp shadow uu[1:1][1:1]
10 ...
11 int main(int argc, char **argv)
12 {
13   ...
14 #pragma acc data copy(u, uu)
15   {
16 #pragma xmp reflect_init (u) acc
17 #pragma xmp reflect_init (uu) acc
18
19     for (t = 0; t < niter; t++) {
20       if ((k % 2) == 0) {
21 #pragma xmp reflect_do (u) acc
22
23 #pragma xmp loop (y, x) on t(y, x)
24 #pragma acc parallel loop gang vector_length(256) async(1)
25         for (x = 1; x < XSIZE−1; x++) {
26 #pragma acc loop vector
27           for (y = 1; y < YSIZE−1; y++) {
28             uu[x][y] = (u[x−1][y] + u[x+1][y] + u[x][y−1] + u[x][y+1]) * 0.25;
29           }
30         }
31       } else {
```

```
32 #pragma xmp reflect_do (uu) acc
33
34 #pragma xmp loop (y, x) on t(y, x)
35 #pragma acc parallel loop gang vector_length(256) async(1)
36         for (x = 1; x < XSIZE−1; x++) {
37 #pragma acc loop vector
38           for(y = 1; y < YSIZE−1; y++)
39             u[x][y] = (uu[x−1][y] + uu[x+1][y] + uu[x][y−1] + uu[x][y+1]) ∗ 0.25;
40         }
41       }
42     }
43   }
44 }
```

リスト A.2　XACC による姫野ベンチマークのプログラム例

```
1  #pragma xmp template t(0:MKMAX−1, 0:MJMAX−1, 0:MIMAX−1)
2  #pragma xmp nodes n(NDY, NDX)
3  #pragma xmp distribute t(∗, block, block) onto n
4  #pragma xmp align p[k][j][i] with t(i, j, k)
5  #pragma xmp align bnd[k][j][i] with t(i, j, k)
6  #pragma xmp align wrk1[k][j][i] with t(i, j, k)
7  #pragma xmp align wrk2[k][j][i] with t(i, j, k)
8  #pragma xmp align a[∗][k][j][i] with t(i, j, k)
9  #pragma xmp align b[∗][k][j][i] with t(i, j, k)
10 #pragma xmp align c[∗][k][j][i] with t(i, j, k)
11 #pragma xmp shadow p[1][1][0]
12 ...
13 #pragma acc data copy(p, bnd, wrk1, wrk2, a, b, c) create(gosa)
14 {
15 #pragma xmp reflect_init (p) acc
16 ...
17   for (n = 0; n < nn; ++n) {
18     gosa = 0.0;
19 #pragma acc update device(gosa)
20
21 #pragma xmp loop (k,j,i) on t(k,j,i)
22 #pragma acc parallel loop firstprivate(omega) reduction(+:gosa) collapse(2) gang vector_length
       (64) async
23     for (i = 1; i < imax−1; ++i)
24       for (j = 1; j < jmax−1; ++j) {
25 #pragma acc loop vector reduction(+:gosa) private(s0, ss)
26         for (k = 1; k < kmax−1; ++k) {
27           s0 = a[0][i][j][k] ∗ p[i+1][j ][k ] + a[1][i][j][k] ∗ p[i ][j+1][k ] + a[2][i][j][k] ∗ p[i ][j ][k+1] + b
                 [0][i][j][k] ∗ ( p[i+1][j+1][k ] − p[i+1][j−1][k ] − p[i−1][j+1][k ] + p[i−1][j−1][k ] ) +
                 b[1][i][j][k] ∗ ( p[i ][j+1][k+1] − p[i ][j−1][k+1] − p[i ][j+1][k−1] + p[i ][j−1][k−1] ) +
                  b[2][i][j][k] ∗ ( p[i+1][j ][k+1] − p[i−1][j ][k+1] − p[i+1][j ][k−1] + p[i−1][j ][k−1] )
```

```
                + c[0][i][j][k] * p[i-1][j ][k ] + c[1][i][j][k] * p[i ][j-1][k ] + c[2][i][j][k] * p[i ][j ][k-1] +
                wrk1[i][j][k];
28            ss = ( s0 * a[3][i][j][k] - p[i][j][k] ) * bnd[i][j][k];
29            gosa += ss*ss;
30
31            wrk2[i][j][k] = p[i][j][k] + omega * ss;
32          }
33        }
34
35  #pragma xmp loop (k,j,i) on t(k,j,i)
36  #pragma acc parallel loop collapse(2) gang vector_length(64) async
37      for (i = 1; i < imax-1; ++i)
38        for (j = 1; j < jmax-1 ;++j) {
39  #pragma acc loop vector
40          for (k = 1; k < kmax-1; ++k)
41            p[i][j][k] = wrk2[i][j][k];
42        }
43
44  #pragma acc wait
45  #pragma xmp reflect_do (p) acc
46  #pragma xmp reduction(+:gosa) acc
47  #pragma acc update host (gosa)
48    } /* end n loop */
49  } //end of acc data
50  ...
```

付録 B

業績一覧

論文誌

1. 小田嶋 哲哉, 朴 泰祐, 塙 敏博, 児玉 祐悦, 村井 均, 中尾 昌広, 田渕 晶大, 佐藤 三久, “アクセラレータ向け並列言語 XcalableACC における TCA/InfiniBand ハイブリッド通信”, 情報処理学会論文誌コンピューティングシステム（ACS）, Vol. 8, No.4, pp. 61-77, Nov. 2015.

査読付き国際会議（口頭発表）

1. Tetsuya Odajima, Taisuke Boku, Toshihiro Hanawa, Hitoshi Murai, Masahiro Nakao, Akihiro Tabuchi and Mitsuhisa Sato, “Hybrid Communication with TCA and InfiniBand on A Parallel Programming Language XcalableACC for GPU Clusters”, Workshop Series on Heterogeneous and Unconventional Cluster Architectures and Applications (HUCAA 2015), pp. 627-634, Sep. 2015.
2. Tetsuya Odajima, Taisuke Boku, Mitsuhisa Sato, Toshihiro Hanawa, Yuetsu Kodama, Raymond Namyst, Samuel Thibault and Olivier Aumage, “Adaptive Task Size Control on High Level Programming for GPU/CPU Work Sharing”, The 2013 International Symposium on Advances of Distributed and Parallel Computing (ADPC 2013), pp. 59-68, Dec. 2013.
3. Tetsuya Odajima, Taisuke Boku, Toshihiro Hanawa, Jinpil Lee, Mitsuhisa Sato, “GPU/ CPU Work Sharing with Parallel Language XcalableMP-dev for Parallelized Accelerated Computing”, The 41st International Conference on Parallel Processing (ICPP 2012) The 5th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2 2012), pp. 97-106, 10-13 Sep. 2012.

査読付き国内会議（口頭発表）

1. 小田嶋 哲哉, 朴 泰祐, 佐藤 三久, 塙 敏博, 児玉 祐悦, Raymond Namyst, Samuel Thibault, Olivier Aumage, “GPU クラスタ向け並列言語 XMP-dev における GPU/CPU 動的負荷分散機能”, 2014 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（HPCS2014）論文集, Vol. 2014, pp. 87-96, Dec. 2014.

研究会（口頭発表）

1. 小田嶋 哲哉, 朴 泰祐, 塙 敏博, 村井 均, 中尾 昌広, 田渕 晶大, 佐藤 三久, “アクセラレータ向け並列プログラミング言語 XcalableACC における TCA/InfiniBand ハイブリッド通信”, 2015 年並列／分散／協調処理に関する『別府』サマー・ワークショップ（SWoPP2015）, 情報処理学会研究報告 ハイパフォーマンスコンピューティング（HPC）, Vol. 2015-HPC-150, No. 43, pp. 1-12, Jul. 2015.
2. 小田嶋 哲哉, 朴 泰祐, 塙 敏博, 児玉 祐悦, 村井 均, 中尾 昌広, 佐藤 三久, “HA-PACS/TCA における TCA および InfiniBand ハイブリッド通信”, 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2014-HPC-147, No. 32, pp. 1-8, Dec. 2014.
3. 小田嶋 哲哉, 朴 泰祐, 佐藤 三久, 塙 敏博, 児玉 祐悦, Raymond Namyst, Samuel Thibault, OlivierAumage, “並列言語 XMP-dev における GPU/CPU 動的負荷分散機能”, 2013 年並列／分散／協調処理に関する『北九州』サマー・ワークショップ（SWoPP 北九州 2013）, 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2013-HPC-140, No. 40, pp. 1-7, Jul. 2013.
4. 小田嶋 哲哉, 李 珍泌, 朴 泰祐, 佐藤 三久, 塙 敏博, 児玉 祐悦, Raymond Namyst, Samuel Thibault, and Olivier Aumage, “GPU クラスタ向け並列言語 XMP-dev における GPU/CPU 協調計算”, 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2013-HPC-138, No. 25, pp. 1-9, Feb. 2013.
5. 小田嶋 哲哉, 李 珍泌, 朴 泰祐, 佐藤 三久, 塙 敏博, 児玉 祐悦, Raymond Namyst, Samuel Thibault, Olivier Aumage, “GPU クラスタにおける GPU/CPU ハイブリッド・プログラミング環境”, 2012 年並列／分散／協調処理に関する『鳥取』サマー・ワークショップ（SWoPP 鳥取 2012）, 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2012-HPC-135, No. 9, pp. 1-8, Jul. 2012.
6. 小田嶋 哲哉, チャン トゥァン ミン, 李 珍泌, 朴 泰祐, 佐藤 三久, “並列言語 XcalableMP の GPU 向け拡張”, 情報処理学会研究報告ハイパフォーマンスコンピューティング（HPC）, Vol. 2011-HPC-129, No. 12, pp. 1-8, Mar. 2011.

ポスター発表

1. Norihisa Fujita, Tetsuya Odajima, “Application and System Software on Tightly Coupled Accelerators Architecture”, 2014 ATIP Workshop: Japanese Research Toward Next-Generation Extreme Computing in SC14, Nov. 2014.
2. Tetsuya Odajima, Taisuke Boku, Mitsuhisa Sato, Toshihiro Hanawa, Yuetsu Kodama, Raymond Namyst, Samuel Thibault and Olivier Aumage, “Work Sharing with GPU and CPU on PGAS Programming Language XcalableMP-dev”, The Fourth AICS International Symposium, Dec. 2013.
3. Tetsuya Odajima, Taisuke Boku, Mitsuhisa Sato, Toshihiro Hanawa, Yuetsu Kodama, Raymond Namyst, Samuel Thibault and Olivier Aumage, “XMP-dev/StarPU: High Level Parallel Programming for GPU/CPU Work Sharing”, HPC in Asia in ISC’13, Jun. 2013.
4. 小田嶋 哲哉, 李 珍泌, 朴 泰祐, 佐藤 三久, “XMP-dev に基づく CPU/GPU ハイブリッド負荷分散システム”, SACSIS2012 ポスター発表, 2012.