修 士 論 文

マルチコアクラスタ向け
チップレベルハイブリッド並列化に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

中尾 哲也

2009 年 3 月

修 士 論 文

マルチコアクラスタ向け
チップレベルハイブリッド並列化に関する研究

指導教官　井口寧 准教授

審査委員主査　田中清史 准教授
審査委員　松澤照男 教授
審査委員　日比野靖 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

0710052 中尾 哲也

提出年月: 2009 年 2 月

概 要

本稿では，マルチコアプロセッサをノード内に複数搭載したクラスタに対して，ノード間及びマルチコアプロセッサチップ間はメッセージパッシングによる並列化を行い，チップ内では共有メモリによる並列化を行う，チップレベルハイブリッド並列化を提案する．マルチコアプロセッサは共有キャッシュを設けている場合が多く，本研究ではその構造を利用した並列化を行う．従来の並列化手法とは異なるマルチコアプロセッサの構造を利用した新たな並列化を行うことで処理時間を短縮することができる可能性がある．本研究では簡単な数値計算による評価として行列積，実際のアプリケーションによる評価として一般逆行列の数値計算を，メッセージパッシングのみで並列化を行う手法や，ノード間はメッセージパッシングによる並列化を行いノード内で共有メモリ並列化を行う手法と比較し，その有効性を調査する．

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

近年，パーソナルコンピュータにマルチコアプロセッサの普及が進んでいる．マルチコアプロセッサとは，1つのプロセッサパッケージ内に複数のコア (演算回路) を搭載したプロセッサのことである．マルチコアプロセッサが登場した背景には，プロセッサ単体での性能向上に限界が見られるようになった点があげられる．2004年頃までは多くのプロセッサメーカーは1回の命令実行速度を速くしたり，命令レベル並列性を高める方法に多くの努力を費やしてきた．しかし次第にLSIの微細化による消費電力・単位面積あたりの発熱量が問題となった．そこで複数のプロセッサコアを搭載することで全体の性能向上を狙うようになったのである．現在市販されているマルチコアプロセッサは，コア数が2つであるデュアルコアプロセッサまたは4つのクアッドコアプロセッサである．またマルチコアプロセッサはコア間が密に結合した構造になっており，大規模な2次キャッシュもしくは3次キャッシュをコア間で共有する構造となっている場合が多く，マルチコアプロセッサの特徴でもある．

次に並列計算機に目を向けてみると，同様にマルチコアプロセッサ普及の波は押し寄せている．世界の高速な計算機をランク付けしているプロジェクトTOP500[13]によると，2008年11月のランキング発表ではIBM社のRoadrunnerが1位を獲得している．Cellプロセッサとデュアルコアプロセッサを搭載しており，総コア数は12万にも及ぶ．また，TOP500にランキングされている中でも336台がクアッドコアプロセッサを，153台がデュアルコアプロセッサを搭載している．本学でも2009年1月時点でデュアルコアプロセッサを192コア搭載したSGI社のAltix4700が稼働しており，3月にはクアッドコアプロセッサを256ノード，総コア数2048であるCray社のCray XT5が導入される．今後もこのような，従来のシングルコアプロセッサとは異なるアーキテクチャを持つマルチコアプロセッサが次々と並列計算機に搭載されるようになることは確実である．

そこで近年特にマルチコアプロセッサを搭載した並列計算機の性能を最大限引き出すことができる並列化手法が求められるようになってきた．マルチコアプロセッサを複数搭載したクラスタに限ると，マルチコアプロセッサを複数搭載したノードが，複数ネットワークで接続した形態をとる．そのためノード間，マルチコアプロセッサのチップ間，チップ内において通信性能が異なり，またチップ内では共有キャッシュを持つために，ノード・チップ・コアという3層構造を成している．そのため，並列処理においては，アーキテクチャの性能を最大限引き出すために，プログラムの並列化において工夫が必要になる．

1.2 目的

本研究の目的は，マルチコアプロセッサを搭載したクラスタにおいて，共有キャッシュが存在するマルチコアプロセッサの構造を利用した並列化を行い，並列処理全体の処理時間の短縮を目指すものである．従来，マルチコアプロセッサを搭載したクラスタのような共有分散メモリ型並列計算機における並列プログラミングモデルとして 2 種類の方法が考えられてきた．すべてのコア間で通信ライブラリを用いてメッセージパッシングを行い並列処理を進めていく方法と，ノード間はメッセージパッシングを行うがノード内では OpenMP などのスレッド API を利用した共有メモリ並列化を行う方法である．すべてのコア間でメッセージパッシングを行う手法は通信オーバーヘッドが大きい．一方ノード間をメッセージパッシングで，ノード内を共有メモリ並列化で並列処理を行う方法はマルチコアプロセッサの構造を無視しており，またノード内のすべてのコアでの共有メモリ並列化はオーバーヘッドが大きくなる可能性がある．

そこで本研究ではノード間及びマルチコアプロセッサチップ間で MPI ライブラリを用いたメッセージパッシングを行い，チップ内のコア間で OpenMP を用いた共有メモリ並列化を行うチップレベル MPI/OpenMP ハイブリッド並列化手法を提案する．そして基礎的な数値計算による評価として行列積に，また実際のアプリケーションによる評価として一般逆行列の数値計算にチップレベルハイブリッド並列化を適用し，全体の処理時間の短縮を行うことを目的とする．

1.3 本論文の構成

本論文でははじめに，第 1 章で本研究の背景と目的について述べる．次に第 2 章で並列計算機の種類及び現状について，また一般的な並列処理の概要を述べる．第 3 章では並列プログラミングモデルについて，従来考えられてきた手法について述べる．また従来手法を現状のマルチコアプロセッサを搭載した並列計算機で実行するにあたっての問題点，本研究で提案する並列化手法について詳細に述べる．第 4 章では基礎的な数値計算による評価として行列積の並列処理に提案手法を適用しその結果を示し考察を行う．第 5 章では実際のアプリケーションへの適用として一般逆行列の並列処理に提案手法を適用しその結果と考察を行う．最後に第 6 章で結論を述べる．ここでは研究を通してわかったこと，問題点や今後の課題について述べる．

第2章　並列計算機

2.1　はじめに

並列計算機の歴史は古く，これまで様々なタイプの並列計算機が誕生してきた．並列計算機はメモリなどの構成により共有メモリ型や分散メモリ型などに分類することができる．また近年は並列計算機の中に新たなアーキテクチャを持つプロセッサとして，マルチコアプロセッサが搭載されたのもが増えてきた．マルチコアプロセッサは共有キャッシュを設けるものが多く，またコア間が密に結合した構造となっている．したがって並列計算には複数の階層構造が生まれることとなる．

2.2　並列計算機の種類

並列計算機の種類はプロセッサ，メモリ，ネットワーク構成の違いにより大別することができる．本節では並列計算機を共有メモリ型，分散メモリ型，共有分散メモリ型に分類し，以下に詳細を述べる．

2.2.1　共有メモリ型

共有メモリ型とは，プロセッサ間で論理的にメモリを共有する並列計算機である (図2.1)．複数のプロセッサが通常のアドレス指定によりデータを読み書きできる共有メモリが存在する．物理的にはメモリが分散しているが、論理的にはメモリを共有している並列計算機を分散共有メモリ型並列計算機という．共有メモリ型には次のような特徴がある．

- 利点
  - プログラミングが比較的容易である
  - データの分割を考慮しなくても並列処理が可能
- 欠点
  - メモリへのアクセスで競合が生じる
  - キャッシュの一貫性などで同期コストがかかる
  - 高価である

UMA

共有メモリ型の中でも，共有メモリへのアクセスコストが，メモリ領域とプロセッサに依存せず均一であるアーキテクチャのことを UMA(Unifirm Memory Access) という．

NUMA

共有メモリ型の中でも，共有メモリへのアクセスコストが，メモリ領域とプロセッサに依存して均一でないアーキテクチャのことを NUMA(Non-Uniform Memory Access) という．更にキャッシュの一貫性を保つものを ccNUMA(cache-coherent Non-Uniform Memory Access) という．

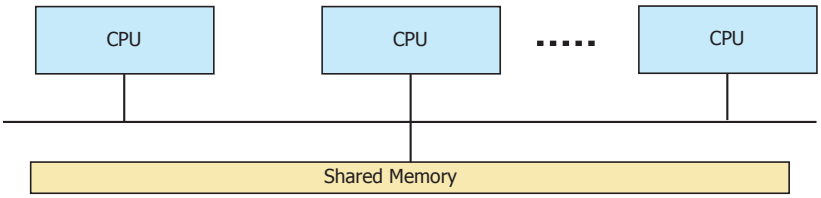

図 2.1: 共有メモリ型

2.2.2 分散メモリ型

分散メモリ型とは，プロセッサとメモリで構成される計算機 (ノード) が複数ネットワークで接続された並列計算機である (図 2.2)．メモリが分散しているため，各計算機がメッセージを送受信して並列動作する．分散メモリ型には次のような特徴がある．

- 利点
  - 拡張性があり，大規模なシステム構築が可能である
  - コストパフォーマンスが高い
  - 通信ライブラリが多数存在する
- 欠点
  - データ送受信をプログラム中に明示的に書く必要がある
  - プログラミングが複雑になる
  - データ送受信に通信コストがかかる

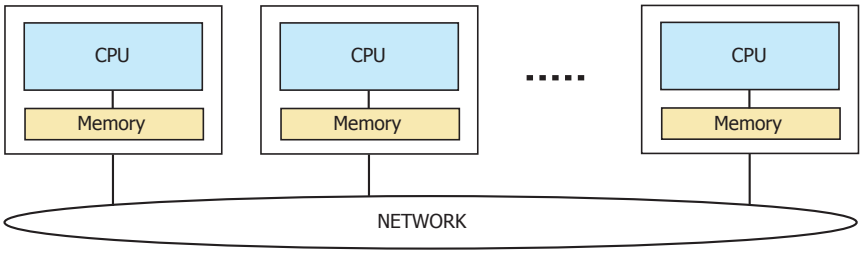

図 2.2: 分散メモリ型

2.2.3 共有分散メモリ型

共有分散メモリ型とは，複数のプロセッサ間でメモリを共有した計算機 (ノード) が複数ネットワークで接続された並列計算機である (図 2.3)．つまり，共有メモリ型と分散メモリ型を合わせた並列計算機である．現在の大規模な並列計算機は，このアーキテクチャが主流である．ノード内とノード間という階層構造を持つ．共有分散メモリ型には次のような特徴がある．

- 利点
  - 大規模なシステム構築が可能である
  - 大規模な並列処理が可能になる
- 欠点
  - 階層構造になるため，良い性能を得るためには並列プログラミングが複雑になる

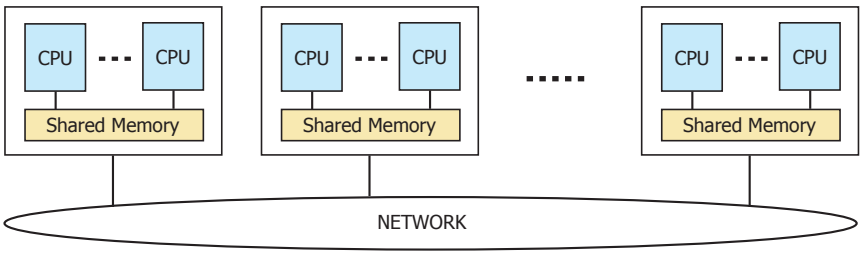

図 2.3: 共有分散メモリ型

2.3 並列計算機の現状

1990 年代に入ると並列計算機は増加し，多種多様なシステムが現れた．そこで並列計算機の有意な統計を得るために，TOP500[13] というプロジェクトが発足した．これは世界で最も高速な計算機システムのうち，上位 500 位までを定期的にランク付けし評価するプロジェクトである．1993 年に発足し，年に 2 回リストの更新を行っている．ハイパフォーマンスコンピューティングにおけるトレンドを追跡及び分析するための基準を提供することを目的としており，LINPACK ベンチマークによるランク付けを行っている．表 2.1 に 2008 年 11 月に発表された TOP500 ランキングを示す．ランキング 1 位は米エネルギー省のロスアラモス国立研究所の Roadrunner で，1.105PFLOPS のベンチマークスコアを記録した．日本の計算機は，27 位の東京大学の T2K クラスタ，29 位の東京工業大学の TSUBAME などがランクインしている．更に現在，日本の文部科学省主導のもとで次世代スーパーコンピュータ開発の国家プロジェクトが進められている．これは 2010 年度を目処に完成及び運用を目指しているプロジェクトであり，理論演算性能 10PFLOPS を目標としている．

現在一般的に販売されているプロセッサのほとんどが，後述するマルチコアプロセッサになっているが，並列計算機においてもマルチコアプロセッサを搭載したものが増えており，TOP500 の中でも 336 台がクアッドコアプロセッサを，153 台がデュアルコアプロセッサを搭載している．

表 2.1: 2008 年 11 月 TOP500 ランキング

順位	システム名 (開発元)	納入先 (国)	コア数	TFLOPS
1	Roadrunner (IBM)	ロスアラモス国立研究所 (米国)	129600	1105.00
2	Jaguar Cray XT5 (Cray)	オークリッジ国立研究所 (米国)	150152	1059.00
3	Pleiades Altix ICE (SGI)	NASA エイムズ研究所 (米国)	51200	487.01
4	BlueGene/L (IBM)	ローレンス・リバモア国立研究所 (米国)	212992	478.20
5	BlueGene/P (IBM)	アルゴンヌ国立研究所 (米国)	163840	450.30
6	Ranger SunBlade x6420 (Sun)	テキサス大学 高度コンピューティングセンター (米国)	62976	433.20
7	Franklin Cray XT4 (Cray)	ローレンス・バークレイ国立研究所 (米国)	38642	266.30
8	Jaguar Cray XT4 (Cray)	オークリッジ国立研究所 (米国)	30976	205.00
9	Red Storm (Cray)	サンディア国立研究所 (米国)	38208	204.20
10	Dawning 5000A (Dawning)	上海スーパーコンピューティングセンター (中国)	30720	180.60

2.3.1 マルチコアプロセッサ

マルチコアプロセッサとは，1つのプロセッサパッケージ内に複数のコア (プロセッサ回路の中核部分) を搭載したプロセッサのことである．プロセッサパッケージ内のコアが2つであればデュアルコアプロセッサ，4つであればクアッドコアプロセッサと呼ばれる．また，現在のマルチコアプロセッサはコア間で2次キャッシュもしくは3次キャッシュを共有する構造となっている場合が多い．

2004年頃までは多くのプロセッサメーカーは1回の命令実行速度を速くしたり，命令レベル並列性を高める方法に多くの努力を費やしてきた．しかし次第にLSIの微細化による消費電力・単位面積あたりの発熱量の増加が問題となり，性能向上に限界が見られるようになった．そこで単一のプロセッサコアの性能向上ではなく，複数のプロセッサコアを搭載することで全体の性能向上を狙うようになった．また近年のプロセッサ製造技術の向上により少ない面積に多くの回路を搭載することが可能となったため，1つのプロセッサパッケージ内に複数のコアを搭載することができるようになった．こうした理由により最近になって急速にマルチコアプロセッサが普及してきた．

大型コンピュータやスーパーコンピュータでは，早くから複数のプロセッサを搭載した対称型マルチプロセッサ (Symmetric Multiple Processor) が登場していたが，これはシングルコアプロセッサ (従来のプロセッサ) をマザーボード上に複数搭載したものでありマルチコアプロセッサとは異なる．

ホモジニアス・マルチコアプロセッサ

同種のコアだけを複数搭載するマルチコアプロセッサをホモジニアス・マルチコアプロセッサと呼ぶ．2008年現在市場に出回っているほとんどのマルチコアプロセッサはホモジニアス・マルチコアプロセッサである．各々のコアは全く同じ性能・構造・命令体系を持つので，あるコアで動作するプログラムは他のコアでも同様に動作する．

ヘテロジニアス・マルチコアプロセッサ

異種のコアを複数搭載するマルチコアプロセッサをヘテロジニアス・マルチコアプロセッサと呼ぶ．1個のPPE(Power Processor Element) と呼ばれる汎用コアと，8個のSPU(Synergistic Processor Unit) と呼ばれるマルチメディア処理に適したコアを搭載したCellプロセッサが有名である．ホモジニアス・マルチコアプロセッサとは異なり，異種のコアが存在するため，あるコアで動作するプログラムがそのまま他のコアでも動作するとは限らない．

2.3.2 マルチコアクラスタと諸問題

本研究で述べるマルチコアクラスタとは，マルチコアプロセッサを搭載した計算機 (ノード) が，複数ネットワークで接続された並列計算機のことである (図 2.4)．つまり，従来のクラスタ型並列計算機のプロセッサがマルチコアプロセッサに置き換わったものである．現在販売されているプロセッサはほとんどがマルチコアプロセッサであり，現在では普通にクラスタを構築しただけでマルチコアクラスタになる．ノード内に複数のマルチコアプロセッサを搭載すると，コア・マルチコアプロセッサチップ・ノードの 3 層構造が生まれる．コア間・チップ間・ノード間は階層毎に通信速度が異なる．また，マルチコアプロセッサはコア間で共有キャッシュを持つ形態のものが多い．よって，ノード内に複数の共有キャッシュ単位の構造 (チップ単位の構造) が生まれる．

ここでマルチコアクラスタによる並列処理を考えると，このような従来とは異なるアーキテクチャの性能を最大限引き出すためには，プログラムの並列化において工夫が必要である．プログラムの並列化におけるプログラミングモデルは，メッセージパッシングによるものや共有メモリを利用した並列化がある．しかしこのような構造の違いを利用した並列プログラミングモデルの研究はまだあまり多くない．

例えば最近のプログラミングモデルに関する研究では，中島の研究 [2] や馬場らの研究 [3] がある．中島の研究では並列有限要素法アプリケーションにメッセージパッシングモデルとハイブリッドモデルの並列化を適用し，TSUBAME Grid Cluster(dual-core) 上で評価実験を行っている．しかし，dual-core を考慮した並列化や実験は行われていない．また馬場らの研究では連立一次方程式 IDR(s) 法にメッセージパッシングモデルとハイブリッドを適用し，プロセス数やスレッド数を変化させて実験を行っている．こちらも日立 SR11000(dual-core) などの環境で実験を行っているが，dual-core を考慮した実験は行われていない．

そこで本研究ではチップ (共有キャッシュ) 構造に着目し，チップ内・チップ間・ノード間という構造を利用した並列プログラミングモデルの提案を行う．

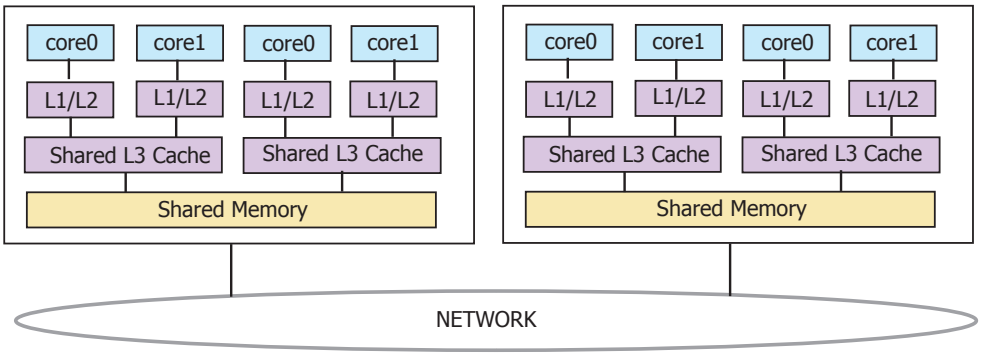

図 2.4: マルチコアプロセッサを搭載したクラスタ

2.4 使用計算機

本研究で使用したマルチコアクラスタの外観・仕様及び評価実験で使用したツールについて述べる．

2.4.1 仕様

本研究で使用した計算機は，クアッドコア AMD Opteron プロセッサを 2 基搭載したノードが Gigabit Ethernet で 4 ノード接続されたクラスタである．計算機の外観を図 2.5 に示す．また仕様の詳細について表 2.2 に示す．

図 2.5: 本研究で使用したマルチコアクラスタの外観

表 2.2: 本研究で使用したクラスタの仕様

CPU	Quad-Core AMD Opteron 2.1GHz 2-sockets (4core × 2)
Cache	L1 Data Cache 64KB, L2 Cache 512KB, Shared-L3 Cache 2MB
Memory	4GB / Node
Network	Gigabit Ethernet
Number of Nodes	4
OS	CentOS 4.6 kernel 2.6.9
Compiler	PGI 7.1
MPI Library	mpich1.2.7p1
Math Library	ScaLAPACK, AMD Core Math Library
Analysis	TAU(Tuning and Analysis Utilities), AMD CodeAnalyst

2.4.2 解析ツール

本研究では，MPI による通信時間の測定に TAU(Tuning and Analysis Utilities)[18] を使用した．また，プログラムのキャッシュヒット率の測定に AMD CodeAnalyst[19] を使用した．

TAU(Tuning and Analysis Utilities)

TAU(Tuning and Analysis Utilities)[18] とは，C,C++,Java,Python 言語で記述された並列プログラムの解析を行うツールセットである．オレゴン大学，ロスアラモス研究所，ユーリッヒ研究所が共同で開発するプロジェクトである．専用コンパイラによりプログラムをコンパイルする．そしてプログラム実行時に解析結果ファイルが出力される．解析結果ファイルは専用ツールにより可視化することができる．本研究では MPI による通信時間の測定に用いた．

AMD CodeAnalyst

AMD CodeAnalyst[19] とは，AMD 社が提供するソフトウェア性能の解析やプロセッサイベントのサンプリングを行うことができるツールである．時間ベース・イベントベース・命令ベース・パイプラインシミュレーションなど，様々な解析を行うことができる．本研究ではプロセッサイベントのサンプリングを行いキャッシュヒット率の測定に用いた．

2.5 並列処理について

並列処理とは，処理する問題を複数の小部分に分割し，それらを複数のプロセッサで同時に処理することで，全体の処理時間を短縮しようとするものである．並列処理にはさまざまな問題点も存在し，適切に行わなければ良い性能が出ない．また，並列処理の性能や効率を知るための指標もある．それらについて以下に詳細を述べる．

2.5.1 オーバーヘッド

並列処理は，プロセッサ同士が独立して同時に処理を行うため，理想的な状況下ではプロセッサの数に比例した性能が得られると考えられる．しかし，実際にはいくつかの要因により図 2.6 に示すようなオーバーヘッドが発生する．以下に並列処理における主なオーバーヘッドについて記述する．

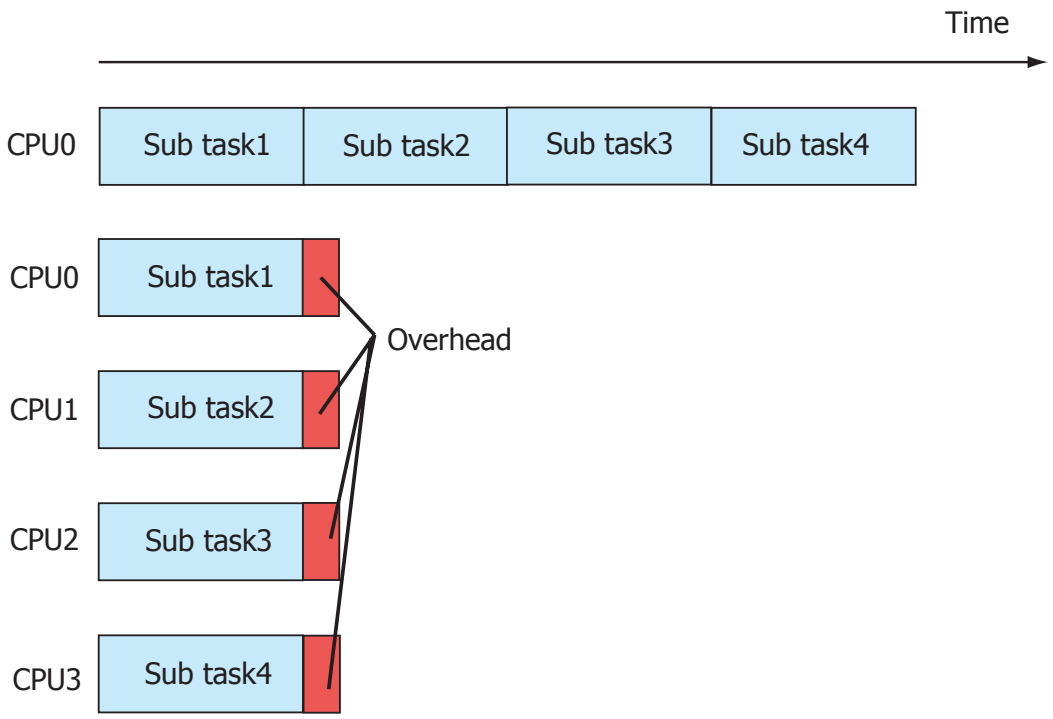

図 2.6: 並列処理におけるオーバーヘッド

通信時間

通信時間が存在することは逐次処理と並列処理の大きな違いの一つである．各々のプロセッサは処理する問題のすべてのデータを保持しているわけではなく，問題の一部分のデータを保持している．つまり，１つのプロセッサのみで処理を行う逐次処理は，必要なデータすべてを保持しているが，並列処理では各プロセッサが分割してデータを保持している場合が多いため，処理の途中でデータを交換する必要がある．このデータの交換を行っている時間が通信時間である．またこの通信は，ネットワークを通して異なるノードのコンピュータにアクセスする場合，同じプロセッサ内や同じノード内，つまり共有メモリ内と比較すると大幅に時間がかかるため，並列処理の効率は著しく悪くなる．

負荷の不均衡

負荷の不均衡とは，各プロセッサに与えられる計算負荷が均衡していない状態のことである．負荷の不均衡が生じると負荷の高いプロセッサに対し，負荷の低いプロセッサは計算量が少なくなり，同期待ち時間が多くなることで並列処理の効率が悪くなる．同期をとるために負荷が高いプロセッサ，つまり計算時間が長くかかるプロセッサを待つことになり，いくら他のすべてのプロセッサが早く処理を済ませていても，一番遅いプロセッサに合わせることになる．このため各プロセッサ間での負荷が均等になるように，処理を分散して割り当てる負荷分散を行う必要がある．この負荷分散には，並列処理をはじめる最初に負荷のバランスを考えて処理を各プロセッサに割り当てる静的負荷分散や，並列処理の途中で各プロセッサの負荷を測定し，動的に処理を割り当てる動的負荷分散などがある．

スレッド並列化

通信時間や負荷の不均衡は主に分散メモリ上でのプロセス並列化におけるオーバーヘッドである．しかし，共有メモリを利用したスレッド並列化を行うにあたってもオーバーヘッドは存在する．一つは，スレッドの生成や消滅によるオーバーヘッドである．スレッドによる並列処理を行うにあたり，スレッドの生成と消滅を頻繁に繰り返しているとそのオーバーヘッドは大きくなる．しかしこのオーバーヘッドは最初にスレッドを生成し，なるべく消滅させないよう適切に実装を行うことでかなり軽減できる．また，メモリアクセスの衝突によるオーバーヘッドも考えられる．これは複数のプロセッサが同時に共有メモリにアクセスするとプロセッサとメモリを繋ぐバスの奪い合いが生じる．このためメモリアクセスが性能を決めているような並列処理プログラムではプロセッサの数を増やした分だけ性能が低下する場合がある．この問題を解決するにはキャッシュの有効利用などの工夫が必要になる．この他にも様々なオーバーヘッドが存在し，共有メモリを利用したスレッド並列化の性能評価は複雑で難しいところがある．

2.5.2 性能の解析

スピードアップ (Speedup) とは逐次処理による実行時間と並列処理による実行時間の比により求められる．$T_\sigma(n)$ を逐次処理による実行時間，$T_\pi(n,p)$ を p 個のプロセスからなる並列処理による実行時間とするとき，並列プログラムのスピードアップは

$$S(n,p) = \frac{T_\sigma(n)}{T_\pi(n,p)} \tag{2.1}$$

である．固定した p の値に対して，普通は $0 < S(n,p) \leq p$ である．もし，$S(n,p) = p$ ならばプログラムは線形スピードアップ (Linear speedup) をもつという．また，ごく希に，線形スピードアップを超えたスーパー線形スピードアップ (Super-linear speedup) をもつことがある．これは，プロセッサ 1 台ではデータセットが大きくキャッシュヒット率が低かった問題が，並列化を行ったことによりプロセッサ 1 台あたりのデータセットが小さくなり，キャッシュヒット率が飛躍的に向上し，メモリアーキテクチャに起因するボトルネックが解消したことで起こる現象である．しかしほとんどの並列プログラムではオーバーヘッドが加わるので，線形スピードアップやスーパー線形スピードアップになることはまずない．

もう一つの指標として効率 (efficiency) がある．効率は逐次プログラムと比較したときの並列プログラムのプロセス利用の計量である．これは次のように定義される．

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_\sigma(n)}{pT_\pi(n,p)} \tag{2.2}$$

$0 < S(n,p) \leq p$ なので $0 < E(n,p) \leq 1$ となる．もし $E(n,p) = 1$ であるならば，プログラムは線形スピードアップを示している．

2.5.3 Amdahl の法則

Amdahl（アムダール）の法則とは，Gene Amdahl（ジーン・アムダール）氏が提唱した，システムの処理性能改善に関する法則である．並列処理の分野でよく使われ，複数のプロセッサを使用したときの理論上の性能向上を予測する．

図 2.7 に 1 個のプロセッサを用いた逐次処理時間と p 個のプロセッサを用いた並列処理時間の関係を示す．

$$T_p = (1-q)T_1 + \frac{qT_1}{p} = \left((1-q) + \frac{q}{p}\right)T_1 \tag{2.3}$$

このときのスピードアップは，式 2.3 を代入し

$$S_p = \frac{T_1}{T_p} = \frac{1}{(1-q) + \dfrac{q}{p}} \tag{2.4}$$

と表され，プロセッサ数 p が無限大に近づく極限では，

$$\lim_{p\to\infty} S_p = \frac{1}{1-q} \tag{2.5}$$

となる．つまり Amdahl の法則は,「並列処理プログラムの速度向上は，対象となるプログラム中の並列化できない部分の割合に大きく左右される」ということを示している．

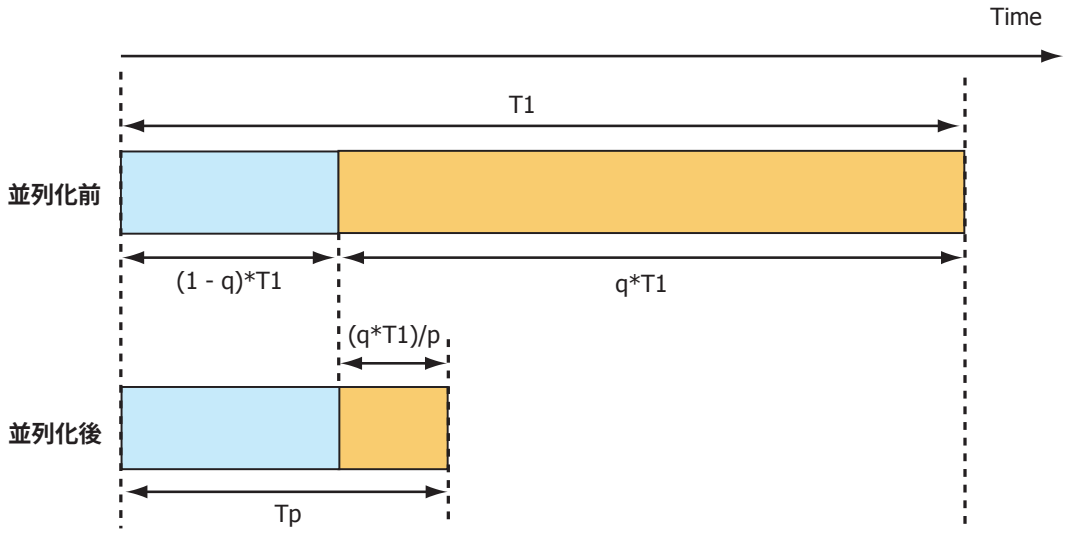

図 2.7: 並列化可能部分と不可能部分

2.6 まとめ

本章では並列計算機の分類とその詳細について述べた．また並列計算機の現状について説明を行い，近年普及しているマルチコアプロセッサについて詳細を述べ，並列計算機に搭載された場合の諸問題についても述べた．また，並列処理の一般論について述べた．

第3章　並列プログラミングモデル

3.1　はじめに

並列計算機を用いて並列処理を行うためには，プログラムをどのようにして並列化を行うかについて，適用する並列プログラミングモデルの検討を行う必要がある．並列プログラミングモデルにはそれぞれに特徴があり使用する並列計算機の種類によっても変わる．どのモデルを選択するのかは非常に重要な問題であり，最適なモデルを選択することで良いパフォーマンスを得ることができる．

3.2　メッセージパッシング並列化

メッセージパッシング並列とは，分散メモリ型並列計算機で広く使われている並列化手法である．各プロセッサ上でそれぞれ独立したプログラム，つまりプロセスが，それぞれ独立したメモリアドレス空間を持って実行される．共有メモリ上であってもメモリアドレス空間が独立したプロセスによって実行される．そして各プロセスは互いに通信を行いながらプログラムの実行を行う．通信メッセージは，データと制御情報から成り，送信側プロセスは送信データを含むメッセージを作り，受信側プロセスに渡す．受信側プロセスは受け取ったメッセージからデータを取り出してメモリに書き込む．つまり，メッセージを送受信するときには，送信側または受信側のメモリを直接アクセスすることはない．

このように各プロセスは通信により互いに協調して並列処理を実行していく．以下にメッセージパッシング並列の特徴を示す．

- 特徴
  - 各プロセッサ上で独立したプログラム (プロセス) を実行する
  - メッセージパッシング・ライブラリ (mpich, OpenMPI) を使用する
  - 通信を明示的にコントロールして並列処理を行う
  - プログラミングは共有メモリ並列と比較して複雑である

3.2.1 MPI(Message Passing Interface)

MPI[9] は Message Passing Interface を意味し，メッセージ通信のプログラムを記述するためのインターフェイスの標準化を目指して作られた，メッセージ通信の API 仕様である．1990 年代の初頭に，それまでベンダーごとに独自に作成されていたプロセス間のメッセージ通信の仕組み・インターフェイスを共通化することを目的に MPI の規格作成が開始された．1994 年に MPI-1(version1) が，翌 1995 年には修正を加えた MPI-1.1(version1.1) がリリースされた．同年，新規機能拡張を考慮した MPI-2(version2) の検討が始まり，1997 年には MPI-2 がリリースされた．

MPI では 100 個以上の関数が定義されているが，実際には 20 個程度の関数で十分プログラミングが可能である．

MPI では主に以下の機能を有している．

1 対 1 通信

MPI における基本的な通信機能である．あるプロセスと別のプロセス間で 1 対 1 で通信を行う．例えば，`MPI_Send` や `MPI_Recv` などである．

集団通信

複数のプロセス間で通信を行う．例えば，`MPI_Bcast` などである．これはあるプロセスからプロセスグループ内のすべてのプロセスへデータをブロードキャストする．

3.2.2 MPI ライブラリ

MPI ライブラリとは，MPI を実装したライブラリのことである．MPI そのものはインターフェイスの規格でしかない．つまり MPI 規格に沿っていれば，ライブラリの中身をどのように実装するかはライブラリ作成者によって異なる．有名なフリーの MPI 実装としては，MPICH[14] がある．これは，アルゴンヌ国立研究所が模範実装として開発し，無償でソースコードを配布したライブラリである．移植性を重視した作りで，盛んに移植が行われ，世界中のほとんどのベンダーの並列計算機上で利用することができる．その他にもフリーの実装として，LAM やその後継である OpenMPI などがある．また SGI 社の MPT や Intel 社の Intel MPI Library など，商用ベンダー各社が独自の実装を行い提供するライブラリも存在する．本研究では，MPICH 1.2.7p1 を使用した．

3.3 共有メモリ並列化

共有メモリ並列化とは，共有メモリ型並列計算機で広く使われている並列化手法である．各プロセッサ上では，スレッドが実行される．スレッドは生成元のプロセスが持つメ

モリ空間を共有している．つまり，各スレッドは共有したメモリ空間を通じて協調して並列処理を行う．共有メモリを利用しているため，分散メモリ型並列計算機では実行することができない．

- 特徴
  - 各プロセッサ上でメモリ空間を共有したスレッドを実行する
  - あるプロセスから生成された複数のスレッドはメモリ空間を共有している
  - スレッドを生成したり制御するためのスレッドシステムやスレッド API(OpenMP, pthread) を利用する
  - プログラミングはメッセージパッシング並列化と比較して簡単である

3.3.1 自動並列化

自動並列化は，共有メモリ並列化を行うにあたって最も簡単な並列化の方法である．自動並列化コンパイラを使用して，プログラムのソースコードを解析し並列実行可能な領域を見つけ，マルチスレッド化する．データの共有設定やスレッドのスケジューリング，同期といった並列化の作業をプログラマーが行わなくても並列化を行うことができる．この自動並列化で主に焦点が当てられるソースコード内の構造はループである．これは，一般的にプログラムの実行時間のほとんどは何らかのループの中で消費されるからである．また，自動並列化の性能はコンパイラの性能に依存するところが大きく，コンパイラによってはループ内に外部関数の呼び出しがある場合には並列化できないなどの条件がある場合があり，スレッド API を利用して手動で並列化を行った場合と比較して良い性能が出ない場合が多い．自動並列化コンパイラには Intel コンパイラや PGI コンパイラなどがあり，コンパイル時に自動並列化を行うオプションを付けることで利用することができる．

3.3.2 OpenMP

OpenMP[15] とは，マルチスレッド並列プログラミングを行うための標準 API である．OpenMP は 1997 年に発表された標準規格であり，多くのハードウェアやソフトウェア・ベンダーが参加する非営利団体「OpenMP Architecture Review Board」によって管理されている．OpenMP は C/C++や Fortran のようなコンパイラ言語ではなく，コンパイラに対する並列処理の機能拡張を規定したものである．したがって，OpenMP を利用するには OpenMP をサポートするコンパイラが必要になる．最近ではスレッド機構が使える多くのコンパイラがサポートしており，ほとんどの UNIX 系 OS に移植されている gcc(GNU Compiler Collection) でもバージョン 4.2 から正式サポートしている．

OpenMP API は，プログラムの並列化領域や，データ属性などの宣言子と並列処理を補助するための OpenMP ライブラリ，また並列処理を行う場合の実行環境を指定する環

境変数から構成される．OpenMP API でプログラムの並列化を行うと，将来のシステムスケールアップに合わせてプログラムを書き直す必要はなく，OpenMP API をサポートしているプラットフォーム間であれば，プログラムの移行は非常に容易である．

OpenMP では主に以下の機能を有している [10].

並列化領域の指示

OpenMP では，プログラムの並列実行領域を，並列実行指示文によってプログラマが明示的に指定する．並列化領域が開始すると，プログラムはスレッドに分かれて並列処理を行う．並列化領域が終了すると分かれたスレッドは消滅し，1 つのマスタースレッドに戻る．

変数属性の指示

並列化領域内の変数 (データ) に対し，その変数をスレッド間で共有するか (Shared)，スレッド固有の変数にするか (Private) を指示する．Private と指示された変数は，各スレッドごとに個別の変数領域が確保される．こうすることで，各スレッドが同一の変数に同時多発的に書き込みを行うことによって起こるデータの不整合を回避することができる．

3.4 ハイブリッド並列化

ハイブリッド並列化とは，メッセージパッシング並列化と共有メモリ並列化を組み合わせて並列処理を行う方法のことである．並列化手法を組み合わせることからハイブリッド並列化と呼ばれる．共有分散メモリ型並列計算機のように共有メモリと分散メモリが混在する環境で適用される場合が多い．メッセージパッシングによる通信処理は元々オーバーヘッドが大きく，それならばメモリを共有している部分は共有メモリ並列化を行ったほうが性能向上が見込まれるのではないかという考えから来ている．しかしこれまでの研究から，どのようなアプリケーションでも性能が向上するわけではなく，すべてメッセージパッシングのみで並列化した方が高速な場合もある．

- 特徴
  - メッセージパッシング並列と共有メモリ並列を組み合わせる
  - メッセージパッシングによるオーバーヘッドを減らせる可能性がある
  - 負荷の不均衡による性能低下を軽減できる可能性がある
  - 2 種類の並列化を考える必要がありプログラミングが複雑になる

3.4.1 MPI/OpenMP ハイブリッド

MPI/OpenMP ハイブリッドとは，メッセージパッシング並列化に MPI を，共有メモリ並列化に OpenMP API を使用して並列化を行う方法である．ハイブリッド並列化を行う方法としては最もよく使用される方法であり，過去のハイブリッド並列処理の研究の多くがこの方法を採用している．MPI はメッセージパッシング並列化のスタンダードでありこれを用いることに疑いの余地はない．OpenMP はコンパイラの能力に依るところもあり，最も性能が良い共有メモリ並列化の方法であるとは言い切れないが，プログラミングを行う際の API の使い勝手が非常に良く，プログラミングが容易でありかつ，細かな調整も可能である点があげられる．このような理由により MPI と OpenMP を組み合わせたハイブリッド並列処理が最も使用されるのではないかと考えられる．その他のハイブリッド並列化の方法としては MPI/pthread ハイブリッドや MPI/自動並列化ハイブリッドなどが考えられる．

ハイブリッド並列化の研究においては，山田らの研究 [4] で通信と演算のオーバーラップによる研究が行われている．これはハイブリッド並列化を行い，ノード内の 1CPU が MPI 通信を行っている間に他の CPU で可能な演算を先に行い，通信と演算をオーバーラップさせる手法である．この研究では通信と演算の比率が小さい場合にはパフォーマンスが落ちるが，その比率が適切な場合には 1,7 倍以上の高速化を達成している．

次に TA QUOC VIET らの研究 [1] を取り上げる．これは SMP Cluster においてメッセージパッシングのみによる並列化とプロセス間 MPI 通信によるハイブリッド並列化，スレッド間 MPI 通信によるハイブリッド並列化をモデル化し比較実験を行っている．NAS-CG ベンチマークと High Performance Linpack に実装を行い，すべての結果において，スレッド間 MPI 通信によるハイブリッド並列化がメッセージパッシングのみによる並列化よりも優れた結果を出している．

3.5 従来手法の問題点

本研究が対象とするようなクラスタ型の並列計算機，つまり共有分散メモリ型並列計算機において並列処理を行うには従来 2 つの方法が考えられてきた．一つは MPI などの通信ライブラリを用いてすべてのコア (プロセッサ) 間でメッセージパッシングのみによる並列処理を行う方法 (以降 PureMPI と呼ぶ, 図 3.1) と，ノード間はメッセージパッシングによる並列処理を行い，ノード内では共有メモリ並列処理を行うハイブリッド並列処理 (以降 NodeHybrid と呼ぶ, 図 3.2) である．

次に，マルチコアプロセッサがノード内に複数搭載された環境での従来手法の問題点について以下に示す．

- PureMPI
  - 通信・同期処理の増加による性能低下．したがって台数効果が得られない

- マルチコアプロセッサの共有キャッシュが有効利用できない (図 3.3)

- NodeHybrid
  - OpenMP によるノード内すべてのコアを使用した共有メモリ並列化によるオーバーヘッド
  - ノード間で大きなデータサイズの通信を行うことによるネットワークバンド幅の飽和

まず PureMPI については，通信・同期処理の増加による性能低下が考えられる．マルチコアプロセッサは 1 チップに複数のコアを搭載している．よって従来の並列計算機よりもノード内に搭載されたコア数が多い．PureMPI はノード内外すべてのコア同士が通信を行う並列モデルである．したがって通信や同期による処理の増加がボトルネックとなり並列処理全体の性能が低下する可能性がある．さらにマルチコアプロセッサの共有キャッシュを有効利用ができない点があげられる (図 3.3)．PureMPI はプロセス並列のモデルである．つまり各コアは独自のメモリ空間を持つプロセスを実行しているため，共有キャッシュのデータも各コア独自のデータがそれぞれ入っていることになる．したがって共有キャッシュは各コアがコア数分で割った大きさのキャッシュを使っていることになる．

一方の NodeHybrid では，OpenMP による共有メモリ並列化のオーバーヘッドが考えられる．これはアプリケーションの実装方法によって異なるが，大抵はスレッド生成やスレッド間で同期などの処理によるオーバーヘッドがある．したがってノード内のすべてのコアで OpenMP を用いた共有メモリ並列化を行えばオーバーヘッドは大きくなる．また，通信に関しても問題点が考えられる．これはマルチコアプロセッサに限った問題ではないが，NodeHybrid はノード間でのみ通信を行うため，通信回数は少ない．しかし，1 度に送るデータ量が大きい．したがって一斉にデータを送受信する場合，ノード間のネットワークバンド幅が飽和し，通信性能が低下する可能性がある．

そこで本研究ではこれらの問題点を解決するために，ノード内のチップ単位の構造，つまり共有キャッシュ単位の構造を利用し，チップレベルでのハイブリッド並列化を提案する．

3.6 チップレベルハイブリッド並列化の提案

マルチコアプロセッサを搭載したクラスタ環境での従来手法の問題点を解決するために，本研究ではチップレベルでの MPI/OpenMP ハイブリッド並列化を提案する．これは，マルチコアプロセッサを搭載することでノード内にできた共有キャッシュ構造，つまりマルチコアプロセッサのチップ構造を利用し，ノード間及びチップ間で MPI によるメッセージパッシング並列化を，共有キャッシュが存在するチップ内では OpenMP による共有メモリ並列化を行う並列化手法である (図 3.4)．チップ内で共有メモリ並列化を行うため共

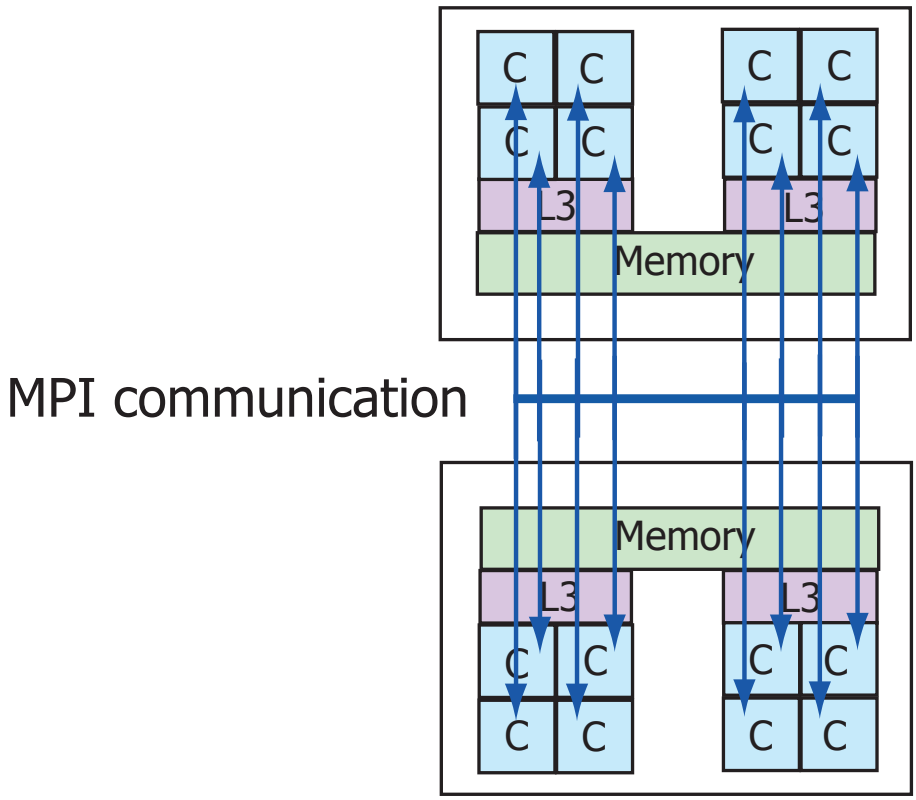

図 3.1: PureMPI

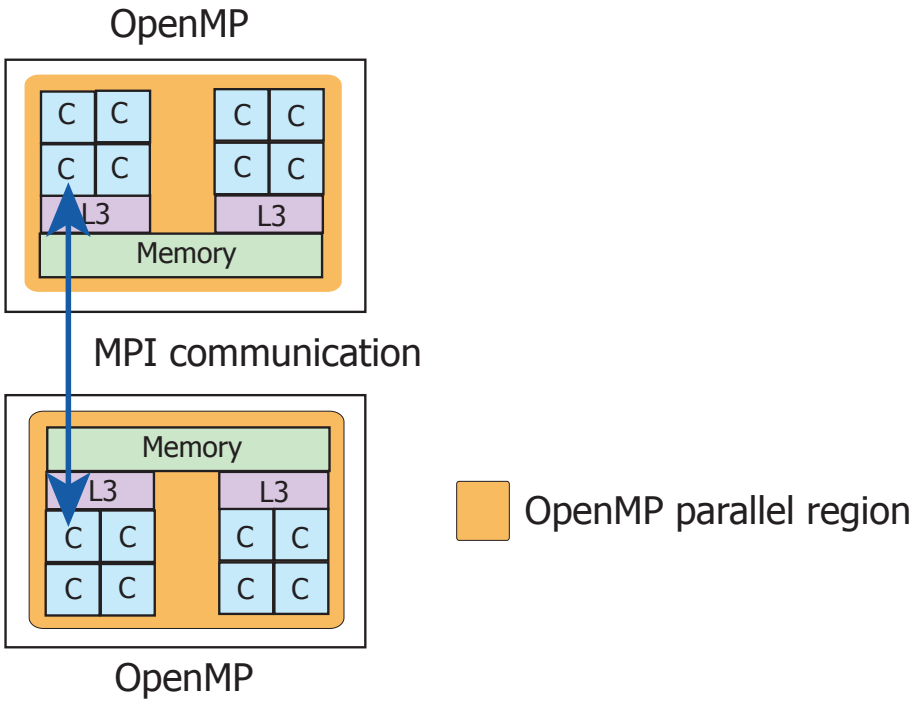

図 3.2: NodeHybrid

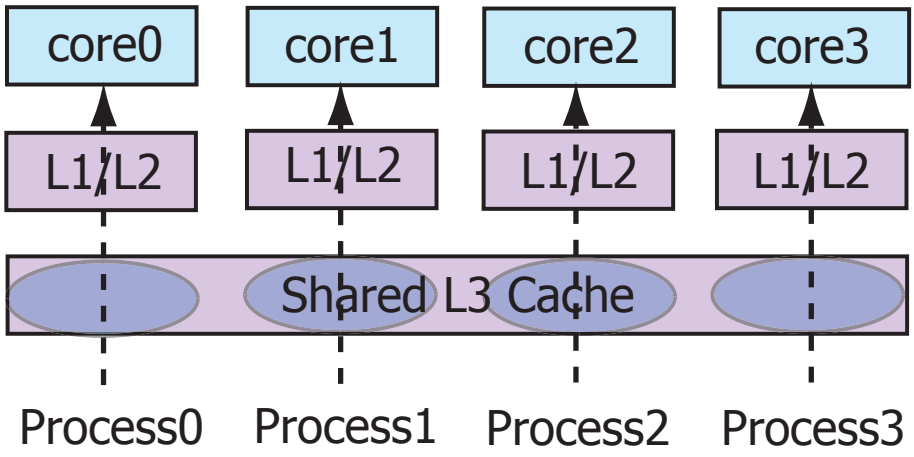

図 3.3: PureMPI の問題点:共有キャッシュが有効利用できない

有キャッシュを活用することができ，またノード内全体で共有メモリ並列化を行うわけではないのでオーバーヘッドも小さいと考えられる．

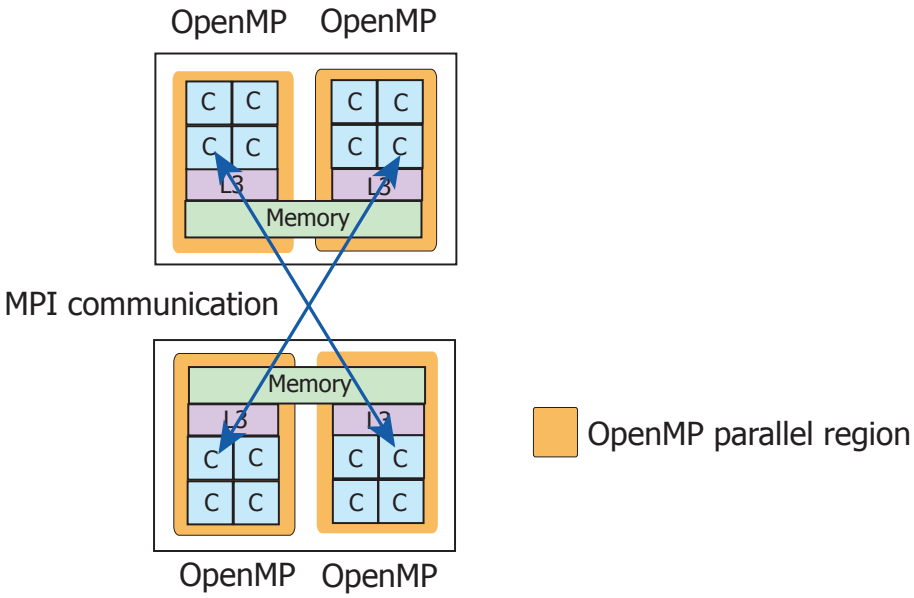

図 3.4: チップレベル MPI/OpenMP ハイブリッド並列化

3.6.1 プロセス・スレッドのコアへの割り当て

通常，プロセスやスレッドをどのコアで実行するかはオペレーティングシステム (OS) が管理している．しかし本提案手法を実現するには，ノード内に生成された MPI プロセスを各マルチコアプロセッサチップ内の 1 コアに割り当て，そのプロセスから生成されるスレッドをマルチコアプロセッサチップ内の全コアに割り当てる必要がある．したがって，プログラマが明示的にプロセスやスレッドの割り当てを指定する必要がある．また，プロセスやスレッドをコアに静的に割り当てることで，マイグレーションを止めることができ，それによって起こるキャッシュの無効化によるアプリケーションの性能低下を防ぐことができる．

本研究ではプロセッサアフィニティ機構 (Processor Affinity Mechanism)[8] を利用してプロセスやスレッドを任意のコアに割り当てる．以下では文献 [8] を参考にしてプロセッサアフィニティ機構の説明を行う．

プロセッサアフィニティ機構とは Linux Kernel 2.6 以降で利用できる，プロセスやスレッドが動作するプロセッサもしくはコアをアプリケーションが明示的に指定できるメカニズムである．アフィニティ機構では，あるスレッドがどのプロセッサ上で動作できるかという許可情報をアフィニティマスク (Affinity Mask) と呼ばれるマスクデータで表す．アフィニティマスクは「そのコンピュータ上に搭載されるプロセッサもしくはコアの集合」

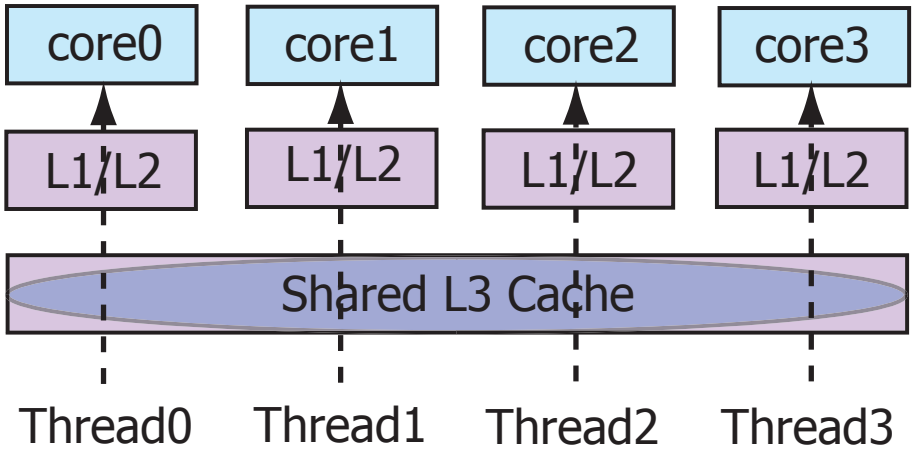

図 3.5: 提案手法による共有キャッシュの有効利用

を表すビットデータ cpu_set_t 構造体で表され，このビットデータの各ビットがそれぞれ 1 つのプロセッサもしくはコアに対応する．スレッドやプロセスは，自分のアフィニティマスクのうち 1 がセットされているコア上でのみ動作が許可される．もし全ビットが 1 である場合には，コアへの割り当ては完全に OS に任される．この機構を利用して，1 つのビットだけを 1 にして残りのビットをすべて 0 にすれば，ある特定のコア上でだけそのプロセス・スレッドの実行を許可することができる．なお，アフィニティマスクはプロセス構造体中に保持されるデータであるが，マスクの値はスレッド毎に管理される．

アフィニティマスクをプロセスやスレッドに設定するには，sched_setaffinity システムコールを，現在のアフィニティマスクを取得するには sched_getaffinity システムコールを使用する．アフィニティマスク関連のシステムコールとマクロを表 3.1 に示す．

これらシステムコールやマクロはヘッダファイル”sched.h” をインクルードすることで利用可能である． システムコールの具体的な使い方についてはいくつかの方法が考えられる．プロセス・スレッドをコアへ割り当てるだけのプログラムを記述し，共有オブジェクトとしてコンパイルし LD_PRELOAD を利用して本プログラムと共に実行する方法や，本プログラムの中に直接記述する方法などである．本研究では本プログラムの中に直接記述する方法を採用した．MPI プログラムにおいては，MPI の初期化関数の直後などにシステムコールを使用して MPI プロセスを各コアに割り当てる処理を記述する．OpenMP プログラムでは，OpenMP の並列化領域が開始される前もしくは直後にスレッドを各コアに割り当てる処理を記述することで利用可能である．図 3.8 に MPI+OpenMP ハイブリッド並列プログラムにおけるプロセスとスレッドのコアへの割り当て動作させる場合の例を示す．

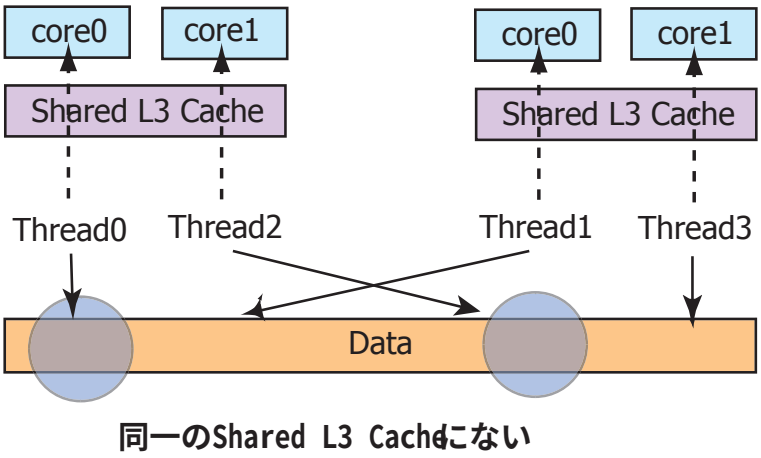

図 3.6: L3 共有キャッシュが有効利用できない場合

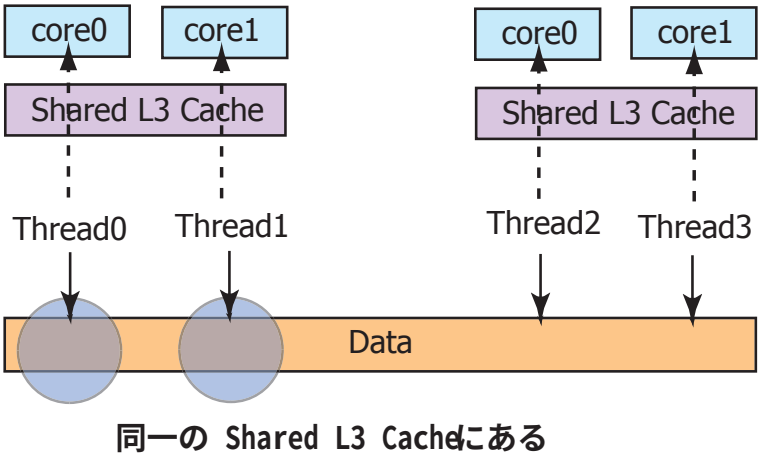

図 3.7: スレッドを適切に割り当てることによる L3 共有キャッシュの有効利用

```
#include <sched.h>
#include "mpi.h"
#include "omp.h"
...
cpu_set_t mask;      /* アフィニティマスク */
int tn;              /* スレッド番号を格納する変数 */
int num_threads = 4; /* スレッド数(コア数)を格納する変数 */

MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
...
/* MPI プロセスのコアへの割り当てを行う */
if( my_rank % 2 == 0 ){
  __CPU_ZERO( &mask );
  __CPU_SET( 0, &mask );
}else{
  __CPU_ZERO( &mask );
  __CPU_SET( num_threads, &mask );
if( sched_setaffinity( 0, sizeof( mask ), &mask ) == -1 )
  printf("WARNING: faild to set CPU affinity");
...

/* スレッドのコアへの割り当てを行う */
#pragma omp parallel shared( num_threads ) private( mask, tn )
{
  tn = omp_get_thread_num();
  if( my_rank % 2 == 0 ){
    __CPU_ZERO( &mask );
    __CPU_SET( tn, &mask );
  }else{
    __CPU_ZERO( &mask );
    __CPU_SET( tn + num_threads );
  }
  if( sched_setaffinity( 0, sizeof( mask ), &mask ) == -1 )
    printf("WARNING: faild to set CPU affinity");
}
...
```

図 3.8: MPI プロセス・OpenMP スレッドのコアへの割り当て (QuadCore-2sockets の場合)

表 3.1: Linux プロセッサアフィニティ関数・マクロ

関数・マクロ	説明
sched_setaffinity	第 1 引数で指定されるプロセス ID を持つプロセスのアフィニティマスクを設定する．プロセス ID に 0 を指定した場合，自プロセスの実行中スレッドに対して指定したことになる．
sched_getaffinity	第 1 引数で指定されるプロセス ID を持つプロセスのアフィニティマスクを取得する．プロセス ID に 0 を指定した場合，自プロセスの実行中スレッドのマスクを取得する．
__CPU_CLR	第 1 引数で指定されたプロセッサを第 2 引数で指定されたアフィニティマスクからクリアする．
__CPU_ISSET	第 1 引数で指定されたプロセッサが第 2 引数で指定されたアフィニティマスクにおいてセットされているかチェックする
__CPU_SET	第 1 引数で指定されたプロセッサを第 2 引数で指定されたアフィニティマスクにセットする
__CPU_ZERO	第 1 引数で指定されたアフィニティマスクの全ビットをクリアする

3.6.2 通信時間

通信時間は並列処理において，また各並列化手法を比較するにあたり最も重要な要素の一つである．一般的に並列処理における全通信時間 T_{COMM} は，データ転送の立ち上がり時間 (セットアップ時間,$Setup\_Time$) とデータサイズ ($DataSize$)，データ転送速度 (バンド幅.$Bandwidth$)，通信回数 (nc) を用いて次式で表すことができる．

$$T_{COMM} = \left(Setup\_Time + \frac{DataSize}{Bandwidth} \right) \times nc \tag{3.1}$$

データ転送の立ち上がり時間とは，通信関数を呼び出した際にかかる時間，データ転送を開始する前処理にかかる時間のことである．また，各プロセッサ間の負荷が不均衡な場合にはこれに同期時間 (待ち時間) が加わることになるがここでは無視することとする．

ここで，PureMPI・NodeHybrid・提案手法の比較を行う．ソフトウェアの実装方法や，通信が 1 対 1 通信であるか集団通信であるかによって細かくは違うのであるが，ここでは一般論を述べる．まず各手法において違いが現れるのはデータサイズ $DataSize$ と通信回数 nc である．前提として，コア数・マルチコアチップ数・クラスタノード数は

$$\text{コア数} \geq \text{マルチコアチップ数} \geq \text{クラスタノード数} \tag{3.2}$$

である．

データサイズ $DataSize$ は，処理する問題をメッセージパッシングで分割する粒度で比較することができる．PureMPI はすべてのコア間でメッセージパッシングを行う手法であるため，問題をコア数で分割する．つまり最も細粒度になる．提案手法はすべてのマル

チコアチップ間でメッセージパッシングを行うため，問題をマルチコアチップ数で分割する．よって中粒度となる．NodeHybrid はすべてのクラスタノード間で行うため粗粒度になる．したがって $DataSize$ を比較すると

$$NodeHybrid \geq \text{提案手法} \geq PureMPI \tag{3.3}$$

となる．

通信回数 nc は単純にコア数･マルチコアチップ数･クラスタノード数に比例する．PureMPI はすべてのコア間で，提案手法はすべてのマルチコアチップ間で，NodeHybrid はすべてのクラスタノード間で通信を行うからである．したがって通信回数 nc を比較すると

$$PureMPI \geq \text{提案手法} \geq NodeHybrid \tag{3.4}$$

となる．

以上の点から，NodeHybrid は通信データのサイズが最も大きくなるためノード間のネットワーク帯域が飽和し，ボトルネックとなる可能性がある．一方，PureMPI は通信回数が最も多くなるためデータ転送の立ち上がり時間がボトルネックとなる可能性がある．よって提案手法が最も短い通信時間になる可能性がある．

3.6.3 共有キャッシュ

共有キャッシュの存在はマルチコアプロセッサの大きな特徴である．本提案手法はチップ内でスレッドによる共有メモリ並列化を行っているため，共有キャッシュを有効に利用できると考えられる．NodeHybrid も同様にノード内のすべてで共有メモリ並列化を行っているため，共有キャッシュを利用できると考えられる．しかし，スレッドがノード内の各コアにデータ配置を考慮せず各々実行されていたとすれば，提案手法よりも共有キャッシュのヒット率は下がる可能性がある．PureMPI はプロセスによる並列化であるため共有キャッシュを有効利用できない．したがって，各並列化手法の共有キャッシュのヒット率は

$$\text{提案手法} \geq NodeHybrid > PureMPI$$

になるのではないかと予想される．またこの違いは演算時間に影響すると考えられる．

3.6.4 処理時間

PureMPI の実行時間 T_{MPI} は，演算時間を $TCalc_{MPI}$，通信時間を $TComm_{MPI}$ とおくと

$$T_{MPI} = TCalc_{MPI} + TComm_{MPI} \tag{3.5}$$

と表すことができる．

一方，NodeHybrid の実行時間 $T_{NodeHyb}$ は，演算時間を $TCalc_{NodeHyb}$，通信時間を $TComm_{NodeHyb}$，OpenMP スレッドによるオーバーヘッドを $OMP_{NodeHyb}$ とおくと

$$T_{NodeHyb} = TCalc_{NodeHyb} + TComm_{NodeHyb} + OMP_{NodeHyb} \tag{3.6}$$

と表すことができる．

さらに提案手法の実行時間 $T_{Proposal}$ は，演算時間を $TCalc_{Proposal}$，通信時間を $TComm_{Proposal}$，OpenMP スレッドによるオーバーヘッドを $OMP_{Proposal}$ とおくと

$$T_{Proposal} = TCalc_{Proposal} + TComm_{Proposal} + OMP_{Proposal} \tag{3.7}$$

と表すことができる．

ここで，使用するノード数が増えると PureMPI の通信オーバーヘッドが大きくなるために

$$TComm_{MPI} > TComm_{Proposal} \tag{3.8}$$

$$TComm_{MPI} > TComm_{NodeHyb} \tag{3.9}$$

になると考えられる．ただし，$TComm_{Proposal}$ と $Tcomm_{NodeHyb}$ の関係は，並列計算機の構成やアプリケーションによって変化すると思われるので，どのようになるか推定することは困難である．

また OpenMP スレッドによるオーバーヘッドは

$$OMP_{NodeHyb} > OMP_{Proposal} \tag{3.10}$$

が成り立つと考えられる．これは提案手法はチップ内でのみ OpenMP による共有メモリ並列化であるが，NodeHybrid はノード内全体で共有メモリ並列化を行うからである．

これらの条件を踏まえると，使用ノード数が増え通信オーバーヘッドが大きくなり，またノード内のコア数が多く共有メモリ並列化のオーバーヘッドが大きい，上の条件を満たす場合には

$$T_{MPI} > T_{NodeHyb} > T_{proposal} \tag{3.11}$$

となると考えられる．

3.6.5 台数効果

本提案手法は PureMPI と比較して台数効果が得られると考えられる．PureMPI はすべてのコアと通信を行っているため，ノード数が増えることでコア数が増加し通信のオーバーヘッドが増加する．したがってノード数が増えることで本提案手法の方が良い台数効果を示すと思われる．

3.7　まとめ

本章ではまずはじめに従来から考えられてきた並列プログラミングモデルについて詳細を述べた．またマルチコアプロセッサを搭載することによって生まれる共有キャッシュの構造を利用した，本研究での提案手法について詳細を述べた．提案手法を実現するためにはスレッドやプロセスを明示的にコアに割り当てる必要があり，本研究ではプロセッサアフィニティ機構を利用した．また従来手法と提案手法の通信時間や演算時間についてその比較を行った．

第4章　基礎的な数値計算による評価

4.1　はじめに

並列プログラミングモデル及び従来手法のマルチコアクラスタ環境での問題点，またその解決法としてチップレベルでハイブリッド並列処理を行う手法を第3で述べた．本研究では，並列処理における通信時間の短縮やキャッシュヒット率の向上による演算時間の短縮により，全体の処理時間の短縮を目指すものである．さらに通信時間が短縮することにより台数効果の向上も期待される．

そこで，本章では基礎的な数値計算として行列積による評価実験を行う．行列積は科学技術計算においてもっとも基礎的な数値計算の一つであり，提案手法を適用することでその有効性を調査することができる．まず通信時間の調査実験を行い，従来手法と提案手法における通信時間と演算時間の比較を行う．また，行列サイズを増加させた場合や使用ノード数を変化させた場合の比較を行う．次にキャッシュヒット率の調査実験を行い，従来手法と提案手法におけるキャッシュヒット率の比較を行う．最後に全体の処理時間の調査実験を行い，行列サイズを変化させた場合及び使用ノード数を変化させた場合の比較を行う．そして台数効果を導出し評価する．

4.2　行列積の並列化

行列積は科学技術計算で頻繁に行われる，もっとも基礎的な数値計算の一つである．図4.1に行列積 $A \times B = C$ をどのように並列化したのかを示す．行列 A に関しては行方向にプロセス数分を均等に分割して各プロセスに送信し部分行列 A を保持する．行列 B に関しては全データを各プロセスが保持し，部分行列 C の計算を行う．そしてデータを集めて処理が完了する．PureMPI はすべてのコア数分のプロセスを生成するので行分割の粒度は小さい．提案手法はすべてのマルチコアプロセッサチップ数分のプロセスを生成するので中粒度，NodeHybrid はノード数分の分割なので粗粒度になる．また共有キャッシュを有効活用するために参考文献[6]に示されている OpenMP における reduction 節の使用を行列積の演算部分に取り入れた．演算量は $O(n^3)$，通信量は $O(n^2)$ である．

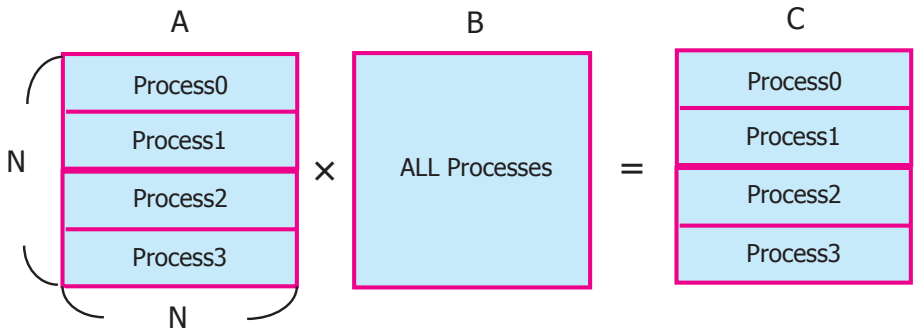

図 4.1: 行列積の並列化

4.3 評価実験

行列積の並列処理について評価を行うために，通信時間の調査，キャッシュヒット率の調査，全体の処理時間の調査及び台数効果の調査を行った．

4.3.1 通信時間の調査

行列積に関して，通信時間の調査実験を行った．この実験により，従来手法と提案手法の通信時間と演算時間の違いを明らかにする．

図 4.2 にノード数 2(コア数 16) の実験結果を，図 4.3 にノード数 4(コア数 32) の実験結果を示す．

まず通信時間に着目すると，行列サイズ N=3840 の提案手法だけ著しく時間がかかっているのがわかる．しかし行列サイズ N=1920 や 2880 の場合には，提案手法と従来手法の通信時間に大きな違いは見られない．そこでツールによる解析結果データから原因を調査してみると，各プロセスが部分行列積を計算し最後に行列を集めるときの通信にほとんどの時間が消費されているのがわかった．つまり N=3840 の提案手法だけ，各コアのキャッシュヒット率の違いなどによるコア間のロードバランスの不均一が起こり，通信待ち時間 (同期時間) が非常に大きくなってしまったからであると思われる．これは手法による違いではなく，行列の分割が偶然キャッシュヒット率に影響したために通信時間が増加したので，特に考察する必要はないと考えられる．

次に演算時間に着目すると，行列サイズ N=1920 及び N=2880 のときには提案手法がもっとも短くなっている．しかし，N=3840 のときには PureMPI がもっとも短い値を示している．これらの原因を調べるために，次にキャッシュヒット率の調査を行う．

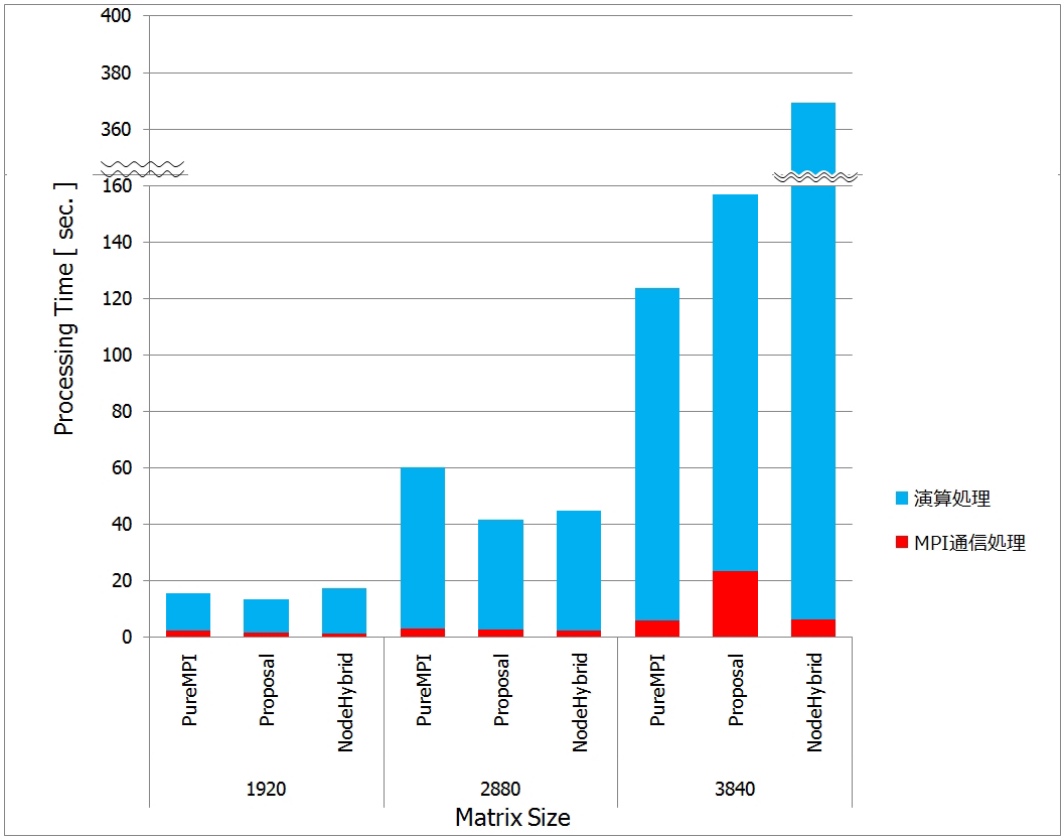

図 4.2: 行列積 - ノード数 2(コア数 16) の解析結果

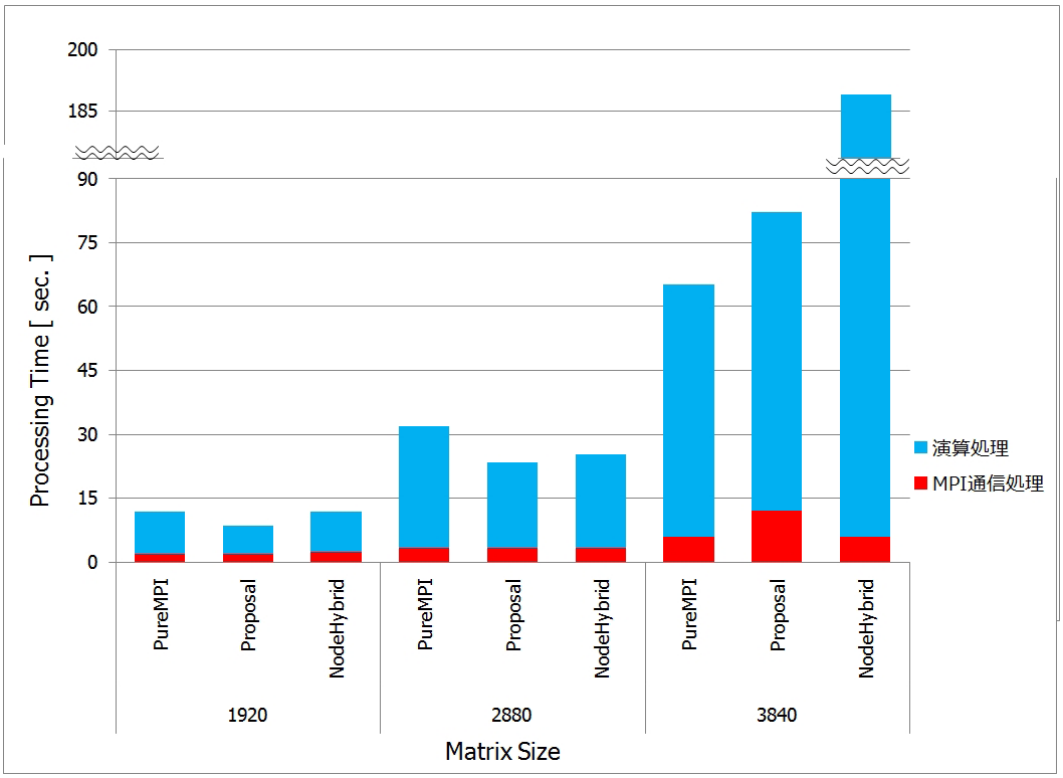

図 4.3: 行列積 - ノード数 4(コア数 32) の解析結果

4.3.2 キャッシュヒット率の調査

キャッシュヒット率の調査実験を行う．この実験により，従来手法及び提案手法のキャッシュヒット率の違いを明らかにする．

実験にはノード数 4(32 コア) を使用した．表 4.1 に行列一辺サイズ N=2880 のときのキャッシュヒット率を，表 4.2 に行列一辺サイズ N=3840 のときのキャッシュヒット率を示す．また，ここでは Average memory access time(平均メモリアクセス時間) の算出を行った．本実験の環境はプロセッサが Opteron プロセッサである．Opteron は Exclusive Cache Architecture を採用しており，各キャッシュレベルで同じデータを持たない仕組みになっている．よってこれらの影響により単純に平均メモリアクセス時間の比較だけではキャッシュ性能を推定することはできない．また平均メモリアクセス時間の導出に使用した L3 共有キャッシュのアクセスレイテンシは各コアの負荷などで変化することがわかっている．したがって以下に示す平均メモリアクセス時間はあくまでも目安であるという前置きをしておくことにする．導出に使用したキャッシュやメモリのレイテンシは参考文献 [7] の値を利用した．

表 4.1: 行列積 - 行列一辺サイズ N=2880 のキャッシュヒット率

N=2880	PureMPI	Proposal	NodeHybrod
L1 data cache hit (%)	50.3	62.7	62.7
L2 cache hit (%)	83.9	79.4	78.9
L3 cache hit (%)	56.4	79.2	83.9
Average memory access time (ns)	7.80	5.91	5.75

表 4.2: 行列積 - 行列一辺サイズ N=3840 のキャッシュヒット率

N=3840	PureMPI	Proposal	NodeHybrod
L1 data cache hit (%)	50.5	63.2	63.3
L2 cache hit (%)	89.4	89.0	88.3
L3 cache hit (%)	1.22	1.04	1.05
Average memory access time (ns)	8.31	6.66	6.84

行列一辺サイズ N=2880 の実験結果の表 4.1 に着目すると，PureMPI の L1，L3 共有キャッシュヒット率が低い．一方，共有メモリ並列化を行っている提案手法及び NodeHybrid は L3 共有キャッシュのヒット率が高くなっていることがわかる．このヒット率の違いは通信時間の調査実験におけるノード数 4(コア数 32) の解析結果の図 4.3 の行列サイズ 2880 に表れている．キャッシュヒット率が低い PureMPI の演算時間が最も長くなっている．平均メ

モリアクセス時間も PureMPI が最も長くなっている．提案手法と NodeHybrid のヒット率を比較すると L2 キャッシュは提案手法が高いヒット率を示しているが，L3 共有キャッシュは NodeHybrid が高い．そこで平均メモリアクセス時間を見ると NodeHybrid の方が若干時間が短いようである．そこで図 4.3 の行列サイズ 2880 を見ると，演算時間は提案手法が短いという逆の結果になっている．したがって，NodeHybrid の演算時間増加はキャッシュヒット率の低下ではなく，OpenMP によるオーバーヘッドの可能性が最も高いと推定される．

一方，行列サイズ N=3840 の実験結果の表 4.2 に着目すると，PureMPI の L3 共有キャッシュヒット率が最も高くなっている．しかし各手法を見るとすべて L3 共有キャッシュは 1%程度のヒット率と非常に低い．L1 キャッシュは PureMPI が最も低い値となっている．にもかかわらず，通信時間の調査実験におけるノード数 4(コア数 32) の解析結果の図 4.3 の行列サイズ 3840 を見ると，PureMPI の演算時間がもっとも短く，NodeHybrid の演算時間がもっとも長くなっている．つまりここでも，キャッシュヒット率の違いよりも OpenMP による共有メモリ並列化のオーバーヘッドが大きく影響したために，NodeHybrid の演算時間が長く，次に提案手法の演算時間が長いという結果になっていると考えられる．

4.3.3 処理時間の調査

行列積による全体の処理時間の調査実験を行う．図 4.4 から図 4.7 に，ノード数 1(8 コア) からノード数 4(32 コア) までの行列サイズを変化させたときの処理時間を示す．

全体的に，処理時間の増加曲線は緩やかにはならず，大きな振れ幅がある．これは行列サイズと使用するコア数・行列分割サイズによってキャッシュのヒット率に大きな違いがあるために，このような振れ幅のある結果となっている．しかし全体を見ると，ほとんどの場合で NodeHybrid がもっとも長い処理時間を示している．つまり共有メモリによる並列化のオーバーヘッドの影響が大きいことがわかる．一方，PureMPI と提案手法では行列サイズによって処理時間が長くなったり短くなったりする．キャッシュのヒット率の影響が大きいために，正確に性能を比較するのは難しいが，全体的に処理時間の曲線を見ると，ほぼ同じ性能と考えてよいと思われる．

4.3.4 台数効果の調査

行列積による台数効果 (Speed-up) の調査を行う．行列サイズ N=960 のときから N=3840 の Speedup を図 4.8 から図 4.11 に示す．

行列サイズ N=960 では PureMPI がもっとも良いスピードアップを示した．これは行列サイズが非常に小さいので通信オーバーヘッドが比較的小さかったためであると考えられる．一方，N=1920,2880 では提案手法がもっとも良いスピードアップを示した．しかし，N=3840 では PureMPI が良いスピードアップを示している．つまり，キャッシュのヒット

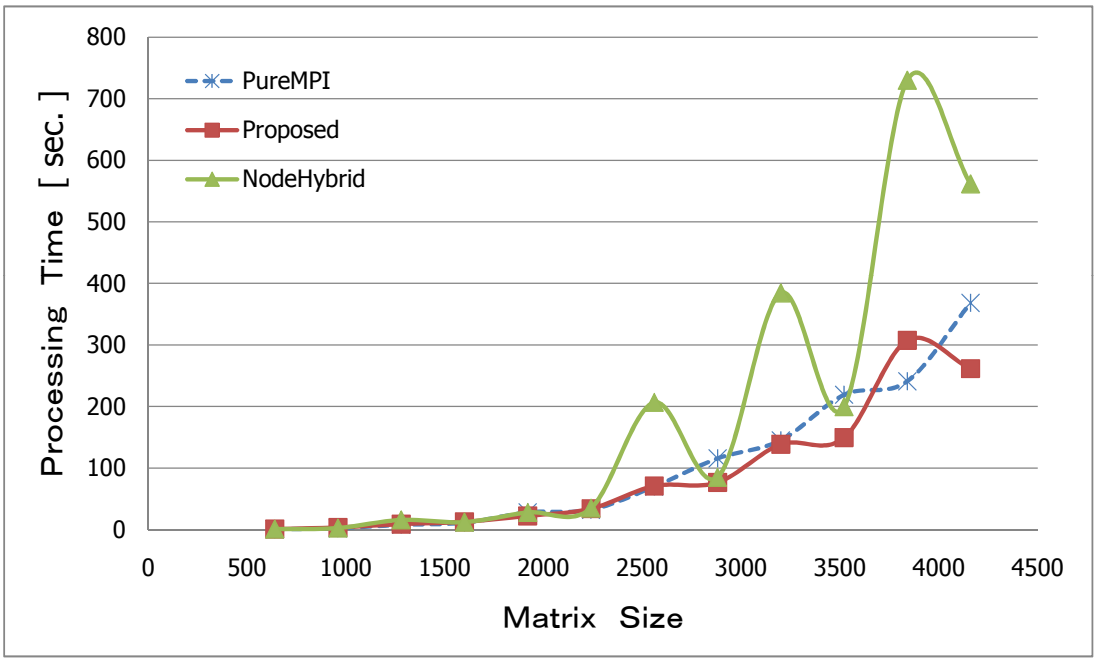

図 4.4: 行列積-ノード数 1(コア数 8) の処理時間

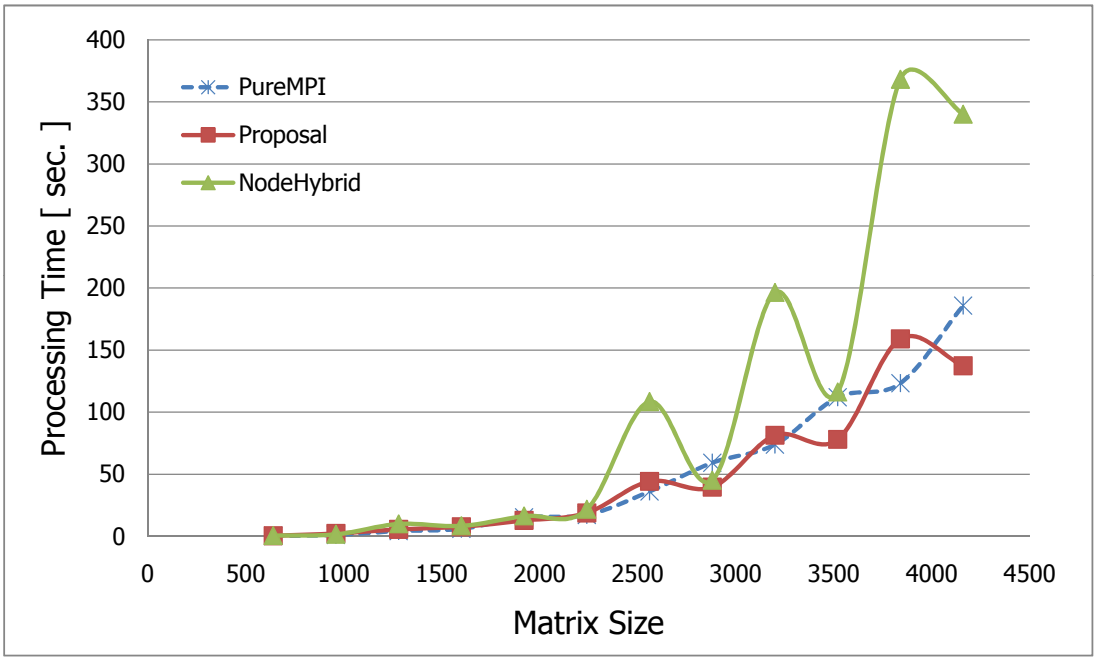

図 4.5: 行列積-ノード数 2(コア数 16) の処理時間

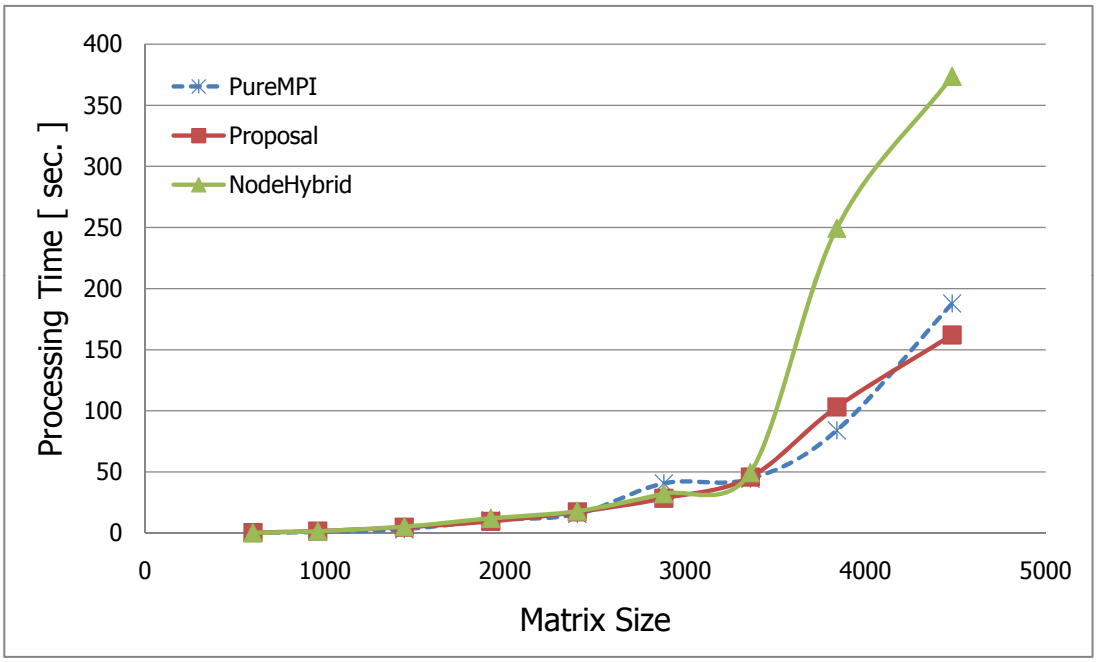

図 4.6: 行列積-ノード数 3(コア数 24) の処理時間

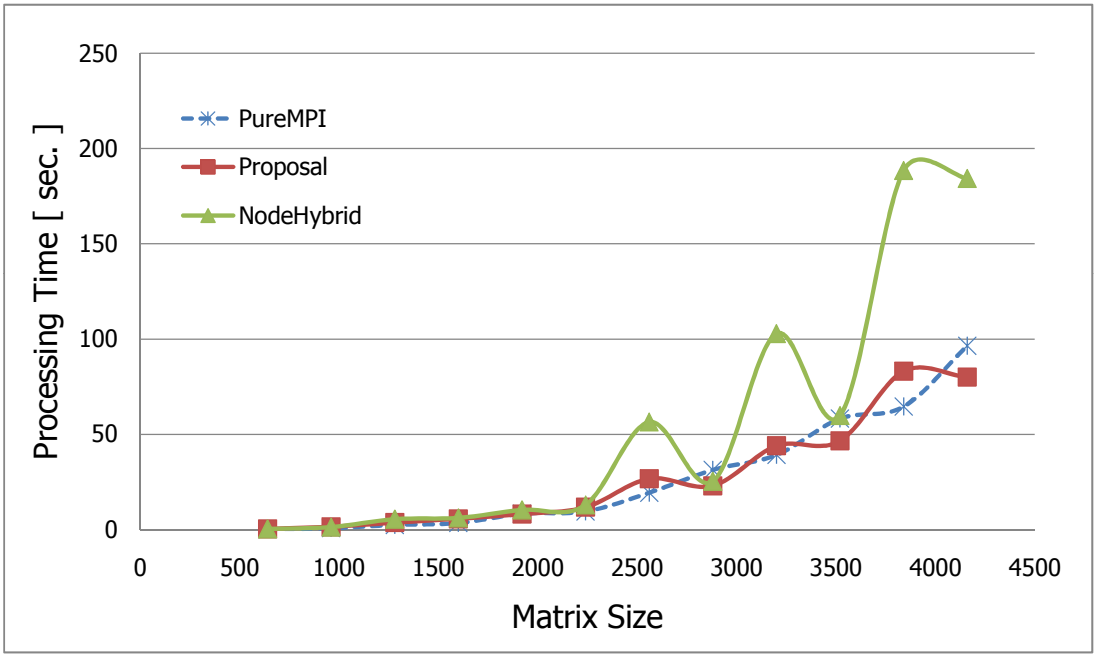

図 4.7: 行列積-ノード数 4(コア数 32) の処理時間

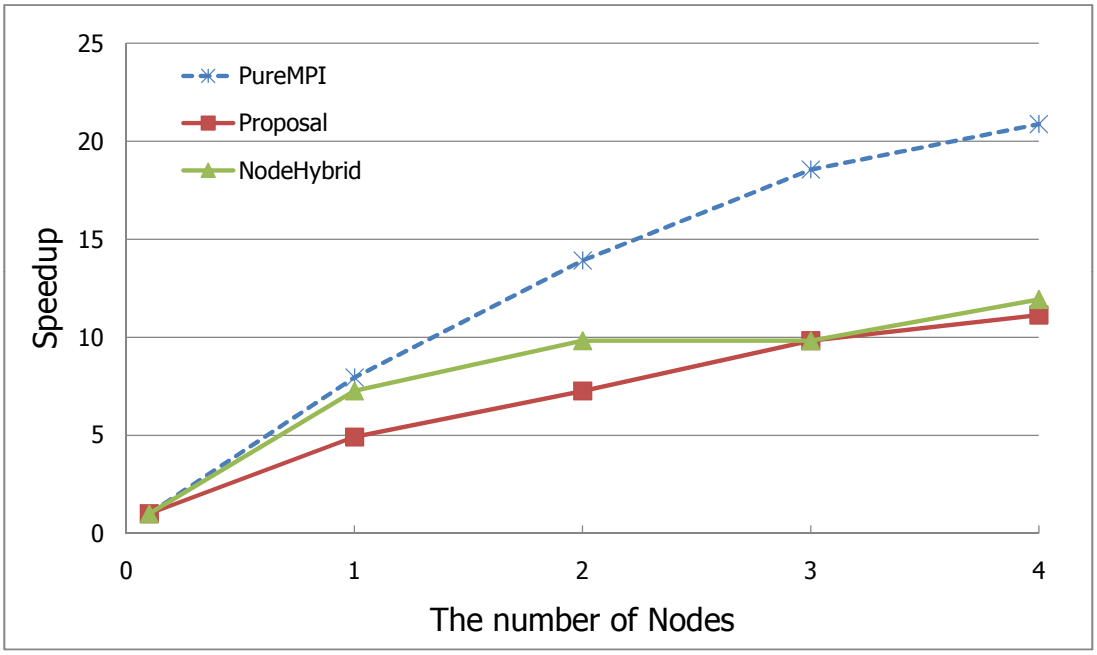

図 4.8: 行列積-行列一辺のサイズ N=960 の Speedup

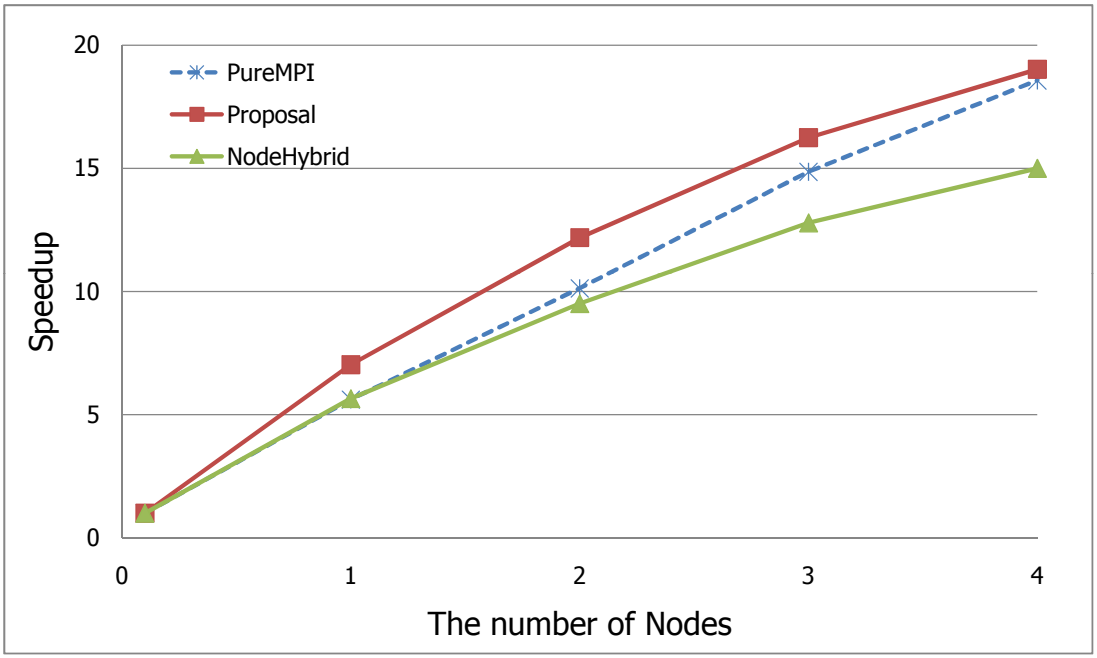

図 4.9: 行列積-行列一辺のサイズ N=1920 の Speedup

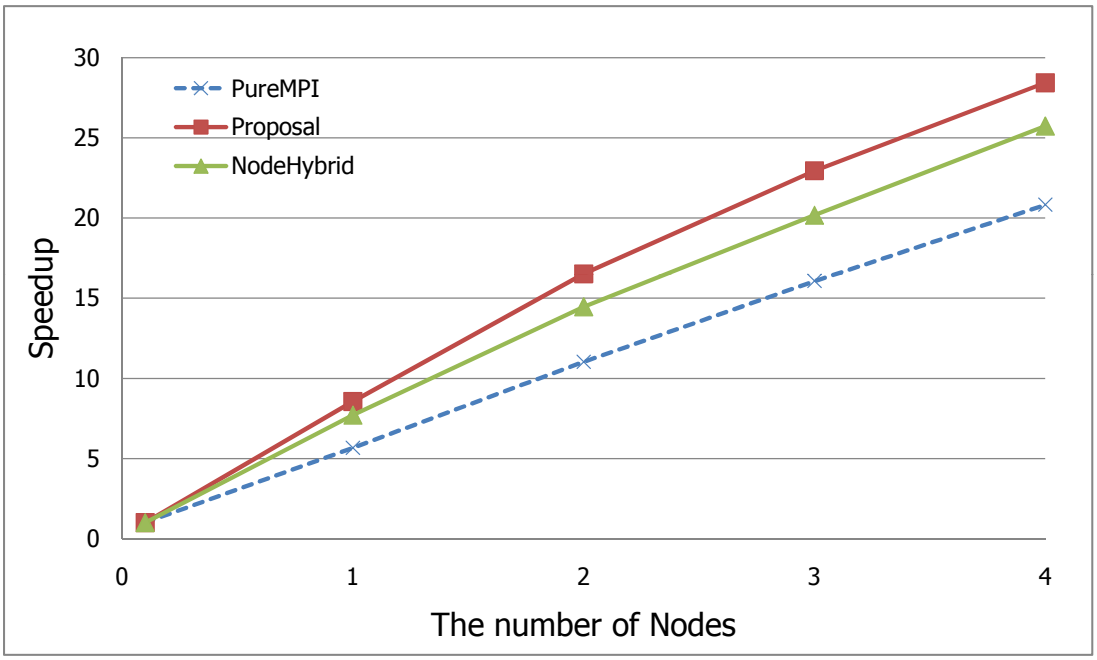

図 4.10: 行列積-行列一辺のサイズ N=2880 の Speedup

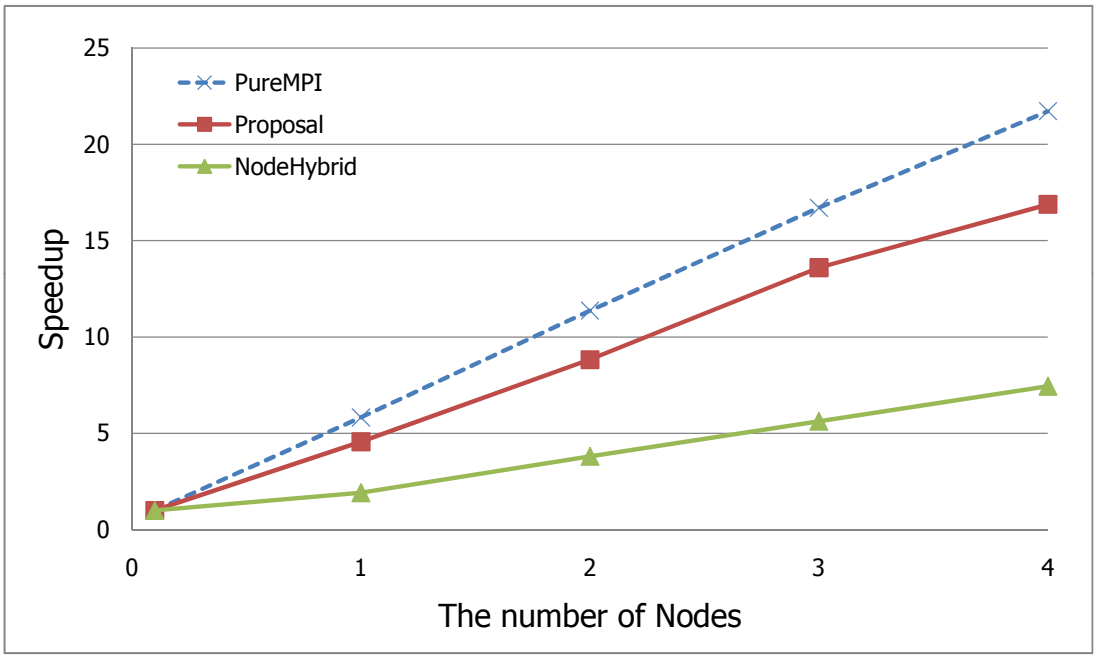

図 4.11: 行列積-行列一辺のサイズ N=3840 の Speedup

率の影響が大きく，手法の違いによる比較が難しくなっている．しかし NodeHybrid が最も良いスピードアップを示している場合はない．

4.4 まとめ

本章では，基礎的な数値計算による評価として行列積の数値計算をとりあげ，実験を行った．行列積は科学技術計算においてもっとも基礎的なそして重要な数値計算の一つである．

はじめに通信時間の調査を行った．使用ノード数の変化による各並列化手法の通信時間の変化はほとんどなく，通信時間に大きな違いは見られなかった．これは行列積の通信処理が，はじめに各プロセスにデータを送信し，最後に結果のデータを集めるという非常に簡単なモデルであるからだと考えられる．PureMPI であってもほとんど通信によるオーバーヘッドは起きない．またサイズ 3840 の提案手法の場合だけ極端に通信時間が長くなったが，これは各コアのキャッシュヒット率が著しく異なるために負荷の不均一が起こり，最後の結果データを集める際に待ち時間が長くなったためである．キャッシュヒット率の調査では，共有メモリ並列化を行う提案手法と NodeHybrid において L3 共有キャッシュのヒット率向上が見られた．また平均メモリアクセス時間を見ても PureMPI より小さい時間であった．にもかかわらず演算時間の増加が確認された理由は，共有メモリ並列化によるオーバーヘッドの方が大きいからであると推定できる．処理時間やスピードアップの調査では，キャッシュヒット率の違いによる影響が大きく，純粋に各手法の性能比較が難しいが，全体を見ると NodeHybrid が最も悪い性能を示しており，提案手法と PureMPI はほぼ同じ性能であると結論付けた．

第5章　実際のアプリケーションによる評価

5.1　はじめに

本章では，実際のアプリケーションによる評価として一般逆行列の並列処理による評価実験を行う．一般逆行列の数値計算は蛍光トモグラフィーと呼ばれる生体内を可視化する，現在盛んに研究が行われている分野で使われている数値計算である．計算時間に時間がかかることから並列処理が求められており，提案手法を適用することでその有効性を調査する．評価実験は第 4 章で行ったものと同じ内容を行う．まず通信時間の調査を行い，従来手法と提案手法における通信時間と演算時間の比較を行う．また，行列サイズや使用ノード数を変化させて実験を行う．次にキャッシュヒット率の調査実験を行い，従来手法と提案手法の比較を行う．最後に全体の処理時間の調査実験を行い，行列サイズを変化させた場合及び使用ノード数を変化させた場合の比較を行う．そして台数効果を調査する．

5.2　蛍光トモグラフィー

蛍光トモグラフィーとは，近赤外線を用いて生物個体の内部構造を可視化する，生命科学分野で最も期待されている技術である．生きた生物の断層像を非侵襲的に得る方法は，生体の構造と機能を解析する上で最も重要な技術の一つである．X 線による CT(Computed Tomography) や MRI(Magnetic Resonance Imaging) がよく知られた技術として存在する．しかし X 線は生体には有害であり多量に浴びると非常に危険である．一方の光トモグラフィーは近赤外線を用いるため安全であり，組織透過率がよく組織深層の蛍光分子を励起できる特徴がある．マウスなどの固体内の生体分子を可視化する目的では，生物発光を利用した画像化技術が開発研究されている [5]．励起光源を必要としないため，暗箱の中に試料となるマウスなどの生物をセットし，高感度 CCD カメラで画像取得するという単純な原理に基づいている 5.1．カメラの性能向上により秒単位で生体内の深部の発光を検出することができる．新たな分子画像化技術として注目されている．この画像化処理において，一般逆行列の演算を行う必要があり，マウスなどではなく人体のようなさらに大きな生体の画像化を行う際には膨大な計算時間が必要になる．よって並列処理による高速化が求められている．

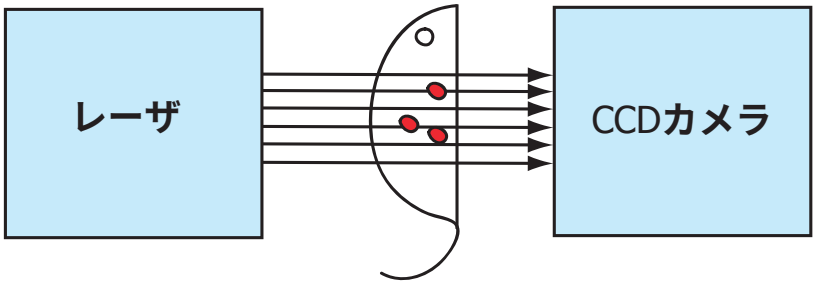

図 5.1: 蛍光分子トモグラフィーシステム

5.3 一般逆行列

本節では一般逆行列の数値計算法について，参考文献 [11] と [12] を引用して詳細な説明を行う．

5.3.1 定義

A を (m,n) 型長方行列とする．次の 4 条件をすべて満足する行列 A^- を一般逆行列（ムーア・ペンローズ型一般逆行列）と定義する．

$$(AA^-)^T = AA^- \tag{5.1}$$
$$(A^-A)^T = A^-A \tag{5.2}$$
$$AA^-A = A \tag{5.3}$$
$$A^-AA^- = A^- \tag{5.4}$$

式 (5.1)，(5.2) は AA^- と A^-A が対称行列であることを示している．A が正方行列 $(m=n)$ で，$|A| \neq 0$ の場合には，A^- と A^-1 とは一致する．A のランクを r とすると，A は次式のように分解できる．

$$^m_{}{}^n A = {}^m_{}B^r_{}{}^n C \tag{5.5}$$

ここに，

$$rank(B) = r, \;\; rank(C) = r \tag{5.6}$$

ここで，B^TB，CC^T の大きさは $r \times r$ となり，

$$rank(B^TB) = r, \;\; rank(CC^T) = r \tag{5.7}$$

となるから，$|B^TB| \neq 0$，$|CC^T| \neq 0$ が成り立つ．よって，$(B^TB)^{-1}$，$(CC^T)^{-1}$ は存在する．そこで，

$$^n_{}{}^m A^- = C^T(CC^T)^{-1}(B^TB)^{-1}B^T \tag{5.8}$$

と置くと，この A^- は式 (5.1) から式 (5.4) をすべて満足する．つまり，任意の行列は一般逆行列を持つことがわかる．ここで，式 (5.8) が式 (5.1) から式 (5.4) を満足することを示す．

1. $(AA^-)^T = AA^-$ $AA^- = BCC^T(CC^T)^{-1}(B^TB)^{-1}B^T = B(B^TB)^{-1}B^T$ であるから，$(AA^-)^T = [B(B^TB)^{-1}B^T]^T = AA^-$ となる．この誘導において，B^TB が対称行列であることを利用している．

2. $(A^-A)^T = A^-A$ $A^-A = C^T(CC^T)^{-1}(B^TB)^{-1}B^TBC = C^T(CC^T)^{-1}C$ であるから，$(A^-A)^T = [C^T(CC^T)^{-1}C]^T = C^T(CC^T)^{-1}C = A^-A$ となる．この誘導において，CC^T が対称行列であることを利用している．

3. $AA^-A = A$ $AA^-A = BCC^T(CC^T)^{-1}(B^TB)^{-1}B^TBC = BC = A$

4. $A^-AA^- = A^-$ $A^-AA^- = C^T(CC^T)^{-1}(B^TB)^{-1}B^TBCC^T(CC^T)^{-1}(B^TB)^{-1}B^T = C^T(CC^T)^{-1}(B^TB)^{-1}B^T = A^-$

A の分解は唯一ではないが A^- は唯一となる．このことを以下に示す．

式 (5.1) から式 (5.4) のすべてを満足する 2 個の行列 A_1^-，A_2^- が存在するとして，$A_1^- = A_2^-$ となることを示す．A_1^- は式 (5.3) を満足するから，$AA_1^-A = A$ である．この両辺に A_2^- を掛けると，$AA_1^-AA_2^- = AA_2^-$ となる．AA_2^- は式 (5.1) より対称行列であるから，$AA_1^-AA_2^- = (AA_1^-AA_2^-)^T$ となる．よって，式 (5.1), (5.3) を用いると，$AA_2^- = AA_1^-AA_2^- = (AA_1^-AA_2^-)^T = [(AA_1^-)(AA_2^-)]^T = (AA_2^-)^T(AA_1^-)^T = AA_2^-AA_1^- = AA_1^-$ となる．同様に，式 (5.2) と式 (5.3) を用いると，$A_2^-A = A_2^-AA_1^-A = (A_2^-AA_1^-A)^T = (A_1^-A)^T(A_2^-A)^T = A_1^-AA_2^-A = A_1^-A$ となる．以上の関係と式 (5.4) を用いると，$A_1^- = A_1^-AA_1^- = (A_1^-A)A_1^- = (A_2^-A)A_1^- = A_2^-(AA_1^-) = A_2^-AA_2^- = A_2^-$ となる．よって，一般逆行列は唯一であることがわかる．この証明において，式 (5.1) から式 (5.4) のすべての条件を利用していることに注目していただきたい．

一般逆行列は線形方程式系 $Ax = g$ の解法が研究される過程で導入，発展されてきた．その過程で，種々の目的に応じて，各種の一般逆行列が考案され，利用されてきている．以下に，各種の一般逆行列の定義と名称をまとめておく．

- 反射型一般逆行列

$$AA^-A = A \tag{5.9}$$
$$A^-AA^- = A^- \tag{5.10}$$

上式を満足する行列 A^- を反射型一般逆行列と呼び，A_r^- で表す．

- ノルム最小型一般逆行列

$$AA^-A = A \tag{5.11}$$
$$(A^-A)^T = A^-A \tag{5.12}$$

上式を満足する行列 A^- をノルム最小型一般逆行列と呼び，A_m^- で表す．

- 最小 2 乗型一般逆行列

$$AA^{-}A = A \tag{5.13}$$
$$(AA^{-})^{T} = AA^{-} \tag{5.14}$$

上式を満足する行列 A^{-} を最小 2 乗型一般逆行列と呼び，A_l^{-} で表す．

式 (5.1) から式 (5.4) の 4 条件をすべて満足する一般逆行列はムーア・ペンローズ型一般逆行列と呼ばれる．ムーア・ペンローズ型一般逆行列は A^{+} で表される場合が多いが，以下においては，特に断らない限り，ムーア・ペンローズ型一般逆行列を，単に一般逆行列と呼び，A^{-} で表し，使用することとする．

5.3.2 性質

(1) (m,n) 型行列 A の一般逆行列 A^{-} は (n,m) 型行列である．

(2) A を (m,n) 型の零行列とする．そのとき，A^{-} は (n,m) 型の零行列である．

(3) A の転置行列の一般逆行列は A の一般逆行列の転置である．つまり，

$$(A^{T})^{-} = (A^{-})^{T} \tag{5.15}$$

(4) A^{-} の一般逆行列は A に等しい．つまり，

$$(A^{-})^{-} = A \tag{5.16}$$

(5) A の一般逆行列のランクは A のランクに等しい．つまり，

$$rank(A^{-}) = rank(A) \tag{5.17}$$

(6) A のランクを r とする．そのとき，$A^{-},\ AA^{-},\ A^{-}A,\ AA^{-}A,\ A^{-}AA^{-}$ のランクは r である．つまり，

$$\begin{aligned} rank(A) &= rank(A^{-}) = rank(AA^{-}) = rank(A^{-}A) \\ &= rank(AA^{-}A) = rank(A^{-}AA^{-}) = r \end{aligned} \tag{5.18}$$

(7) 行列 A について次式が成立する．

$$(A^{T}A)^{-} = A^{-}(A^{T})^{-} = A^{-}(A^{-})^{T} \tag{5.19}$$

(8) 行列 A について次式が成立する．

$$(AA^{-})^{-} = AA^{-}(A^{-}A)^{-} = A^{-}A \tag{5.20}$$

(9) P を (m,m) 型直交行列，Q を (n,n) 型直交行列，A を任意の (m,n) 型行列とするとき次式が成り立つ．

$$(PAQ)^- = Q^T A^- P^T \tag{5.21}$$

(10) A が対称行列のとき，A^- も対称行列である．

(11) A が対称行列のとき，$AA^- = A^-A$ が成立する．

(12) A を n 次正方行列とする．$A^T = A$ および $A^2 = A$ を満足する行列 A を直交射影行列という．このとき $A^- = A$ が成立する．

(13) D を $d_{ii}(i = 1, \cdots, n)$ を対角要素とする n 次元の対角行列とする．このとき次の関係が成り立つ．

$$D = \begin{bmatrix} d_{11} & & & 0 \\ & d_{22} & & \\ & & \ddots & \\ 0 & & & d_{nn} \end{bmatrix}, \ d_{ii} \neq 0 \text{ のとき } D^- = \begin{bmatrix} d_{11}^- & & & 0 \\ & d_{22}^- & & \\ & & \ddots & \\ 0 & & & d_{nn}^- \end{bmatrix} \tag{5.22}$$

$$D = \begin{bmatrix} d_{11} & & & 0 \\ & d_{22} & & \\ & & \ddots & \\ 0 & & & d_{nn} \end{bmatrix}, \ d_{11} = 0, \ d_{ii} \neq 0 (i = 2, \cdots, n) \text{ のとき}$$

$$D^- = \begin{bmatrix} 0 & & & 0 \\ & d_{22}^{-1} & & \\ & & \ddots & \\ 0 & & & d_{nn}^{-1} \end{bmatrix} \tag{5.23}$$

(14) $A = \begin{bmatrix} B & 0 \\ 0 & C \end{bmatrix}$ の一般逆行列は $A^- = \begin{bmatrix} B^- & 0 \\ 0 & C^- \end{bmatrix}$ である．

(15) A を (m,n) 型行列とする．このとき次の関係が成り立つ．

$$rank(A) = m \text{ のとき：} A^- = A^T(AA^T)^{-1}, \ AA^- = I_m \tag{5.24}$$

$$rank(A) = n \text{ のとき：} A^- = (A^TA)^{-1}A^T, \ A^-A = I_n \tag{5.25}$$

(16) AA^-，A^-A，$I - AA^-$，$I - A^-A$ はすべて直交射影行列である．

(17) B をランク r の (m,r) 型行列，C をランク r の (r,n) 型行列とする．このとき次式が成り立つ．

$$(BC)^- = C^-B^- \tag{5.26}$$

5.3.3 数値計算法

任意の行列に対して，唯一に存在するムーア・ペンローズ型一般逆行列の数値計算法を大別すると次のようになる．

- 階数分解による方法
- 特異値分解による方法
- 収束計算による方法
- その他の方法

本節では上記に示した分類に従い，計算法の概説を行う．

階数分解による方法

この方法は，何らかの方法で行列 A を分解し，式 (5.8) を用いて A^- を求めるものである．以下では A を (m,n) 型行列とする．

ガウスの消去法による方法　A の i 行目の成分をピボット a_{ij}（A の i 行 j 列の成分）で割り，次いで a_{kj} 倍し，k 行目から引く．この作業を表す行列を M_i とする．この M_i を用いるとガウスの消去法は次式で表される．

$$M_r M_{r-1} \cdots M_1 A = {}^{r}_{m-r}\left[\frac{{}^{n}U}{O}\right] \tag{5.27}$$

上式において r は消去が終了する最終の行の番号，すなわち，A のランクである．また，U は (r,n) 型上台形行列である．式 (5.27) より，

$$A = M_1^{-1} \cdots M_{r-1}^{-1} M_r^{-1}\left[\frac{U}{O}\right] \tag{5.28}$$

ここで，

$$P^{-1} = M_1^{-1} \cdots M_{r-1}^{-1} M_r^{-1} \tag{5.29}$$

と置くと，式 (5.28) は次式となる．

$$A = P^{-1}\left[\frac{U}{O}\right] = {}^{m}[{}^{r}_{P_1} \,|\, {}^{m-r}_{P_2}]^{r}_{m-r}\left[\frac{{}^{n}U}{O}\right] \tag{5.30}$$

ここで，

$$L = P_1 \tag{5.31}$$

と置くと，L は対角成分がすべて 1 の (m,r) 型下台形行列となる．U と L を用いると

$$A = LU \tag{5.32}$$

と階数分解できる．また，A の一般逆行列は式 (5.8) より

$$A^- = U^T(UU^T)^{-1}(L^TL)^{-1}L^T = U^T(L^TAU^T)^{-1}L^T. \tag{5.33}$$

ハウスホルダー法を用いたQR分解による方法　ハウスホルダー変換（鏡映変換）を表す行列を変換を行う順に $P_1, P_2. \cdots, P_m$ とすると，A は次のように分解できる．

$$A = P_1 P_2 \cdots P_m \begin{bmatrix} R \\ Q \end{bmatrix} = QR \tag{5.34}$$

Q は $P_1, P_2. \cdots, P_m$ の前半 r 列から成る (m, r) 型分割行列で，互いに直交する r 個の m 次正規化列ベクトルから成っている．一般に，$P_i^2 = I_m (i = 1, \cdots, m)$ であるから，$Q^T Q = I_r$ となる．R は (r, n) 型上台形行列である．式 (5.34) の階数分解を用いると，式 (5.8) より

$$A^- = R^T (RR^T)^{-1} Q^T. \tag{5.35}$$

特異値分解による方法

行列 A が実対称行列の場合は次式で与えられる固有値分解が可能である．

$$A = UDU^T \tag{5.36}$$

ここに，U は変換行列，D は対角行列である．

A がランク r を持つ (m, n) 型行列のときには次のように分解することができ，これを特異値分解という．

$$\begin{aligned} A &= \mu_1 u_1 v_1^T + \cdots + \mu_r u_r v_r^T \\ &= U \Sigma V^T \end{aligned} \tag{5.37}$$

ここに，

$$\Sigma = \begin{bmatrix} \mu_1 & & 0 \\ & \ddots & \\ 0 & & \mu_r \end{bmatrix},\ U = [u_1, \cdots, u_r],\ V = [v_1, \cdots, v_r] \tag{5.38}$$

$A^T A$ の 0 でない固有値を $\lambda_i (i = 1, \cdots, r)$ とするとき

$$\mu_i = \sqrt{\lambda_i} \tag{5.39}$$

u_i,v_i は μ_i に対して一意に定まる正規化ベクトルで

$$U^T U = V^T V = I_r \tag{5.40}$$

の直交条件を満足している．このとき A^- は

$$\begin{aligned} A^- &= \frac{1}{\mu_1} v_1 u_1^T + \frac{1}{\mu_2} v_2 u_2^T + \cdots + \frac{1}{\mu_r} v_r u_r^T \\ &= V \Sigma^{-1} U^T. \end{aligned} \tag{5.41}$$

特異値分解を用いる方法　行列 A の特異値分解を求め，次いで，式 (5.41) より A^- を求める方法である．本研究では，一般逆行列を求める数値計算法として特異値分解を用いる方法を採用した．詳しくは 5.3.4 小節で述べる．

固有値分解を用いる方法　A^- の計算に固有値分解を用いる方法を述べる．

$$V = A^T U \Sigma^{-1} \tag{5.42}$$

式 (5.42) を式 (5.41) へ代入すると

$$A^- = V\Sigma^{-1}(\Sigma^{-1})^T V^T A^T = V(\Sigma^2)^{-1} V^T A^T \tag{5.43}$$

あるいは

$$A^- = A^T U \Sigma^{-1}\Sigma^{-1} U^T = A^T U(\Sigma^2)^{-1} U^T \tag{5.44}$$

ここで，式 (5.62) から式 (5.39) との比較より，$V\Sigma^2 V^T$ は $A^T A$ の固有値分解であり，$U\Sigma^2 U^T$ は AA^T の固有値分解であることがわかる．従って，式 (5.43)，あるいは，式 (5.44) より A^- は $A^T A$，あるいは，AA^T の固有値分解から求めることができる．その手順は

(i) $A^T A$（あるいは AA^T）を固有値分解し，すべての固有値 λ_i と固有ベクトル $v_i (i = 1, \cdots, r)$ を計算する．

(ii) λ_i と v_i を用いて

$$\Delta = \Sigma^2 = \begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_r \end{bmatrix},\ V = [v_1, \cdots, v_r] \tag{5.45}$$

を作成する．

(iii) Δ^{-1} を求め，$V\Delta^{-1}V^T$ を作る．

(iv) 右側から A^T を掛ける（AA^T のときは左側から掛ける）．

以上より A^- が得られる．$A^T A$ を使うか，AA^T を使うかは，結果がなるべく小さな行列になるようにすれば良く，A が縦長行列のときは $A^T A$，横長行列のときは AA^T を利用する．

収束計算による方法

繰り返し計算や収束計算により A^- を求める方法を述べる．

ペンローズの方法　以下の手順に従って繰り返し計算を行う.

(i) $B = A^T A$ とおく.

(ii) $C^{(1)} = I_n$ とおく.

(iii)

$$C^{(j+1)} = I_n \frac{1}{j} trace(C^{(j)} B) - C^{(j)} B \ (j = 1, 2, \cdots) \tag{5.46}$$

(iv) 上式の繰り返し計算をおこない，最初に $C^{(r+1)} B = 0$ となった時点で計算を終える.

(v) このとき，$trace(C^{(r)} B) \neq 0,\ rank(B) = rank(A) = r$ となり，

$$A^- = \frac{r C^{(r)}}{trace(C^{(r)} B)} A^T \tag{5.47}$$

この方法では A のランクに等しい回数で繰り返し計算が終わり，これから A のランクも知ることができる．しかし，数値計算の誤差が含まれるときには，$C^{(r+1)} B = 0$ の判定が困難となる場合も生じるので注意が必要である.

A が縦長行列 $(m < n)$ のときには A の代わりに A^T を用いて計算し，最後に $(A^T)^- = (A^-)^T$ より A^- を求めればよい.

ベン-イスラエルの収束計算法　λ_1 を $A^T A$ の最大固有値とし，実数 α を次の式を満たすように選ぶ.

$$0 < \alpha < \frac{2}{\lambda_1} \tag{5.48}$$

このとき，次のように与えられる行列の列

$$X_0 = \alpha A^T \tag{5.49}$$

$$X_{k+1} = X_k (2 I_m - A X_k),\ k = 0, 1, \cdots \tag{5.50}$$

は A^- に収束する．従って，本計算法では X_k が A^- に必要な精度で収束するまで式 (5.50) を繰り返すことになる.

その他の方法

前述した 3 つの分類に入らない計算法も提案されており，それらの方法を述べる.

グレヴィーユの方法　A を列ベクトルで表示すると

$$A = [a_1, a_2, \cdots, a_n] \tag{5.51}$$

ここで,

$$A_{n-1} = [a_1, a_2, \cdots, a_{n-1}] \tag{5.52}$$

と置くと，式 (5.51) は次式となる．

$$A = [A_{n-1},\ a_n] \tag{5.53}$$

上式を用いると A^- は次式の形に書くことができる．

$$A^- = \begin{bmatrix} A_{n-1}^- - A_{n-1}^- a_n b_n^- \\ b_n^- \end{bmatrix} \tag{5.54}$$

ここで，b_n^- は m 次元列ベクトル b_n の一般逆行列であり，次式で与えられる．

(1) $a_n \neq A_{n-1}A_{n-1}^- a_n$ の場合

$$b_n = (I_m - A_{n-1}A_{n-1}^-)a_n \tag{5.55}$$

(2) $a_n = A_{n-1}A_{n-1}^- a_n$ の場合

$$b_n = \frac{[1 + a_n^T (A_{n-1}A_{n-1}^T)^- a_n](A_{n-1}A_{n-1}^T)^- a_n}{a_n^T (A_{n-1}A_{n-1}^T)^- (A_{n-1}A_{n-1}^T)^- a_n} \tag{5.56}$$

また,

$$b_n^- = \begin{cases} b_n = 0 \text{ のとき } 0 \\ b_n \neq 0 \text{ のとき } \dfrac{b_n^T}{b_n^T b_n} \end{cases} \tag{5.57}$$

以上を準備として，次の手順で A^- を求めることができる．ただし,

$$A_k = [a_1, \cdots, a_k] \tag{5.58}$$

(i) $A_1 = a_1$ と置き，式 (5.57) から A_1^- を求める．

(ii) $A_2 = [A_1 a_2]$ と置き，式 (5.54) から A_2^- を求める．

(iii) 以下同様に，$A_k = [A_{k-1} a_k]$ より A_k^- を求める．

(iv) 最終的に $A_n = [A_{n-1}\ \ a_n]$ より A_n^- を求める．

近似計算法　A^- の近似計算法として，式 (5.25) を基とした方法がある．つまり，

$$A^- = \lim_{\delta \to 0}(A^T A + \delta I)^{-1} A^T \tag{5.59}$$

ただし，$\delta > 0$ で

$$|A^T A + \delta I| \neq 0 \tag{5.60}$$

δ を小さくするほど精度は良くなるが，$(A^T A + \delta I)^{-1}$ の計算中にランク落ちが生じないようにする必要があり，δ を小さくするには限度がある．

5.3.4　特異値分解による方法

本研究では　本小節では特異値分解による数値計算法について詳細に述べる．

特異値分解

行列 A が m 行 n 列であるとする．行列 A に対して，適当な直交行列 $\tilde{U}$ および $\tilde{V}$ を選ぶなら，行列 A を次のとおりに分解することができる．

$$A = \tilde{U}\tilde{\Sigma}\tilde{V}^T \tag{5.61}$$

ここで，$\tilde{U}$ および $\tilde{V}$ はそれぞれ m 行 n 列および n 行 m 列の直交行列であり，$\tilde{\Sigma}$ は $diag(\sigma_i),\ \sigma_i \geq 0$ なる m 行 n 列の対角行列である．

式 (5.61) を，行列 A の特異値分解といい，対角行列 Σ の各要素 σ_i を行列 A の特異値という．いま，$m \geq n$ と仮定する．ここで，$\tilde{\Sigma}$ が対角行列であることに着目すると，m 行 n 列の対角行列の左側より掛け合わせる行列 $\tilde{U}$ の場合，行列 $\tilde{U}$ の m 行 n 列だけが有効であり，他の要素は，結果になんら影響を与えていない．さらに，対角行列の右側より掛け合わせる行列 $\tilde{V}$ の場合，行列 $\tilde{V}$ の n 行 n 列だけが有効であり，他の要素は，結果になんら影響を与えていない．

$m < n$ の場合にも同様のことがいえる．

上記の事柄を踏まえて，実用的な特異値分解は，式 (5.61) ではなく，不要な領域を持たない，不要な計算を行わないために，下式のとおり行われる．

$$A = U\Sigma V^T \tag{5.62}$$

ここで，$l = min(m, n)$ として，U は m 行 l 列，V は n 行 l 列の行列，Σ は $diag(\sigma_i),\ \sigma_i \geq 0$ なる l 次の対角行列である．

特異値分解を実現するアルゴリズムを解説する前に，特異値に関する性質を述べる．

(1) 特異値 $\sigma_1, \sigma_2, \cdots, \sigma_l$ は，$A^T A$ の固有値の大きい方から数えて，l 番目までの正平方根である．また，V の第 i 列は，固有値 σ_i^2 に対する $A^T A$ の固有ベクトルであり，U の第 i 列は，固有値 σ_i^2 に対する AA^T の固有ベクトルである．

(2) A の条件数 $(COND)$ は，$COND(A) = \sigma_1/\sigma_l$ で得られる．ここで，σ_1 および σ_l は，代数的に 1 番目および l 番目に大きい特異値である．よって，$\sigma_l = 0$ であるとは，行列 A の階数落ちが発生しており，その条件数が無限に広がることを意味する．

(3) A の階数は，行列 A を特異値分解して得られる対角行列 Σ の，ゼロでない対角要素 σ_i の数に等しい．

このような性質を活用することにより，特異値分解による行列のもつ特性を把握することができる．

特異値分解を行うアルゴリズム

ここでは，ハウスホルダー法と QR 法を組み合わせたアルゴリズムについて述べる．m 行 n 列の行列 A が与えられたとき，どのように特異値分解されていくかを述べる．

(a) 行列の上 2 重対角行列化 2 つのハウスホルダー変換行列 $P_1, \cdots, P_n$ と $Q_1, \cdots, Q_{n-2}$ を適当に選び，行列 A の左右から掛け合わせると，上 2 重対角行列 J_0 を得ることができる．

$$J_0 = P_n \cdots p_1 A Q_1 \cdots Q_{n-2} \tag{5.63}$$

まず，行列 A を A_1 とする，ここで，

$$\begin{aligned} A_{k+1/2} &= P_k A_k \ \ (k = 1, \cdots, n) \\ A_{k+1} &= A_{k+1/2} Q_k \ \ (k = 1, \cdots, n-2) \end{aligned} \tag{5.64}$$

と定義する．また，行列 A_k の要素を $A_k = (a_{ij}^{(k)})$ とする，ここで，行列 P_k を

$$a_{ik}^{(k+1/2)} = 0 \ \ (i = k+1, \cdots, m)$$

となるように選び，行列 Q_k を，

$$a_{kj}^{(k+1)} = 0 \ \ (j = k+2, \cdots, n)$$

となるように選ぶ．

このような行列 P_k や Q_k をどのように決めたらよいかを，以下に示す．

$$\begin{aligned} P_k &= I - x_k x_k^T / b_k \ \ (k = 1, \cdots, n) \\ Q_k &= I - y_k y_k^T / c_k \ \ (k = 1, \cdots, n-2) \end{aligned} \tag{5.65}$$

ただし，

$$\begin{aligned}
x_k &= (0,\cdots,0,a_{kk}^{(k)}+d_k,\ a_{k+1,k}^{(k)},\cdots,a_{nk}^{(k)})^T \\
d_k &= ((a_{kk}^{(k)})^2+\cdots+(a_{nk}^{(k)})^2)^{1/2} sign(a_{kk}^{(k)}) \\
b_k &= d_k^2+d_k a_{kk}^{(k)} \\
y_k &= (0,\cdots,0,a_{k,k+1}^{(k+1/2)}+e^k,\ a_{k,k+2}^{(k+1/2)},\cdots,a_{kn}^{(k+1/2)})^T \\
e_k &= ((a_{k,k+1}^{(k+1/2)})^2+\cdots+(a_{k,n}^{(k+1/2)})^2)^{1/2} sign(a_{k,k+1}^{(k+1/2)}) \\
c_k &= e_k^2+e_k a_{k,k+1}^{(k+1/2)}
\end{aligned}$$

このように，行列 P_k および Q_k を用いて，行列 A を上 2 重対角行列化する方法をハウスホルダー法という．

(b) 上 2 重対角行列の対角行列化上 2 重対角行列を J_0 として，J_0 に対して QR アルゴリズムを逐次適用して対角行列化していく．

2 つの直交変換の列 $S_i, T_i, i = 0.1.\cdots$ を適当に選び，J_0 の左右から掛け合わせていく．

$$J_{i+1} = S_i^T J_i T_i \tag{5.66}$$

このような行列 J_{i+1} を次第に対角行列 Σ 化することができる．

$$J_0 \to \cdots \to J_i \to J_{i+1} \to \cdots \to \Sigma$$

すなわち，適当な回数 l の変換によって対角化を図る．

$$\begin{aligned}
J_1 &= S_0^T J_0 T_0 \\
J_2 &= S_1^T S_0^T J_0 T_0 T_1 \\
&\vdots \\
\Sigma &= J_{l+1} = S_l^T \cdots S_0^T J_0 T_0 T_1 \cdots T_l
\end{aligned}$$

となるようにする．

以上より，行列 A は，対角行列化される．変換行列 U と V は，

$$\begin{aligned}
U &= P_1 \cdots P_n S_0 \cdots S_l \\
V &= Q_1 \cdots Q_{n-2} T_0 \cdots T_l
\end{aligned}$$

と表現される．つまり，行列 U と V は，これまでに現れた変換行列を掛け合わせて得ることができる．

次に，直交である行列 T_i や S_i をどのように選ぶかを説明する．

(c) QR 法における変換行列の選び方ひと言でいうなら，直交行列 T_i は，$M_i = J_i^T J_i$ が対角行列に収束するように選び，もう一方の直交行列 S_i は，すべての J_i が上 2 重対角行列となるように選べばよい．

ここでは，上項で述べた上 2 重対角行列の対角行列化を行った際に用いた直交行列の選び方を，さらに，説明する．

説明を始める前に，これから使用される記号について定義しておく．

$$J \equiv J_i,\ \ \bar{J} \equiv J_{i+1},\ \ S \equiv S_i,\ \ T \equiv T_i,\ \ M \equiv J^T J,\ \ \bar{M} \equiv \bar{J}^T \bar{J}$$

ここで，$\bar{J}^T \equiv SJT$ なる変換は，J に対して，ギブンズの回転を，右および左側より交互に適用させて行う．

$$\bar{J} = L_n^T L_{n-1}^T \cdots L_2^T J R_2 R_3 \cdots R_n \tag{5.67}$$

ここで，

$$L_k = \begin{bmatrix} & & & & k-1 & k & & & \\ & 1 & & & \vdots & \vdots & & & \\ & & \cdots & & \vdots & \vdots & & & \\ & & & 1 & \vdots & \vdots & & & \\ k-1 & \cdots & \cdots & & cos\theta_k & -sin\theta_k & & & \\ k & \cdots & \cdots & & sin\theta_k & cos\theta_k & & & \\ & & & & & & 1 & & \\ & & & & & & & \cdots & \\ & & & & & & & & 1 \end{bmatrix}$$

$$R_k = \begin{bmatrix} & & & & k-1 & k & & & \\ & 1 & & & \vdots & \vdots & & & \\ & & \cdots & & \vdots & \vdots & & & \\ & & & 1 & \vdots & \vdots & & & \\ k-1 & \cdots & \cdots & & cos\phi_k & -sin\phi_k & & & \\ k & \cdots & \cdots & & sin\phi_k & cos\phi_k & & & \\ & & & & & & 1 & & \\ & & & & & & & \cdots & \\ & & & & & & & & 1 \end{bmatrix}$$

角度 $\theta_2, \cdots, \theta_n$ と $\phi_3, \cdots, \phi_n$ を，発生した 2 重対角外要素を順次消去するように選ぶなら，任意な角度 ϕ_2 に対して，$\bar{J}$ が上 2 重対角行列にすることができる．すなわち，以下のとおりにする．

$$J = J_{ij}$$

とするならば,

R_2 は，何も消去せず，非零要素 J_{21} を発生させる．
L_2^T は，J_{21} を消去して，非零要素 J_{13} を発生させる．
R_3 は，J_{13} を消去して，非零要素 J_{32} を発生させる．
$\cdots\cdots\cdots$
R_n は，$J_{n-2,n}$ を消去して，非零要素 $J_{n,n-1}$ を発生させる．
L_n^T は，$J_{n,n-1}$ を消去する．

$$\begin{aligned} S &= L_2 L_3 \cdots L_n \\ T &= R_2 R_3 \cdots R_n \end{aligned} \tag{5.68}$$

とおくと，式 (5.67) より,

$$\bar{J} = S^T J T \tag{5.69}$$

となる．ここで，$\bar{J}$ は上 2 重対角行列であるから,

$$\bar{M} = \bar{J}^T \bar{J} = T^T J^T S S^T J T = T^T J^T J T = T^T M T \tag{5.70}$$

は，対称 3 重対角行列となる．

ところで，いままで，行列 R_2 における角度 ϕ_2 を任意としていたが，ϕ_2 の値を次のとおり定める．つまり，式 (5.70) なる行列 M から行列 $\bar{M}$ への変換は，式 (5.71) に示す原点移動量 s を伴う QR 変換であるようにできる．

$$\bar{M}_s = T_s^T M T_s$$

ここで,

$$\begin{aligned} M - sI &= T_s R_s \\ R_s T_s + sI &= \bar{M}_s \\ T_s^T T_s &= I \end{aligned} \tag{5.71}$$

R_s: 上三角行列

原点移動量 s は，行列 M の右下隅の 2 行 2 列の小行列の固有値より決める．この s の選択によって，ほとんど 3 次のオーダで収束することが広く知られている．

実際には，T_s と T は，同じであることが確かめられる．このためには，$M - sI$ の第 1 列が R_2 の第 1 列に比例するように回転角 ϕ_2 を決めればよい．

これで，行列 S と T が確定したことになる．

特異値分解要素の一般逆行列

行列 A の特異値分解を

$$A = U\Sigma V^T \tag{5.72}$$

とする．また，行列 U の右側に，既存の列ベクトルと互いに直交する $(m-n)$ 個の列ベクトルを付加した，m 次の直交行列 $\tilde{U}$ を導入する．さらに，Σ の下側に $(m-n)\times n$ なるゼロ行列（すべての要素がゼロである行列）を付加した行列 $\tilde{\Sigma}$ を導入する．

導入された m 次の直交行列 $\tilde{U}$ および m 行 n 列の対角行列 $\tilde{\Sigma}$ を用いて，式 (5.72) は式 (5.73) のとおり表現できる．

$$A = \tilde{U}\tilde{\Sigma}V^T. \tag{5.73}$$

Σ の一般逆行列は，次式で定義される．

$$\tilde{\Sigma}^+ = diag(\sigma_1^+, \sigma_2^+, \cdots, \sigma_n^+)$$

ここで，

$$\sigma_i^+ = \begin{cases} 1/\sigma_i & (\sigma_i > 0) \\ 0 & (\sigma_i = 0) \end{cases}$$

このことは，$\tilde{\Sigma}$ が m 行 n 列の対角行列であることから，すぐにわかる．

$X = V\tilde{\Sigma}^+\tilde{U}^T$ と仮定する．この n 行 m 列の行列 X と式 (5.73) の行列 A を一般逆行列の定義へ代入してみると，式 (5.74) が得られる．

$$\begin{aligned} \tilde{\Sigma}\tilde{\Sigma}^+\tilde{\Sigma} &= \tilde{\Sigma} \\ \tilde{\Sigma}^+\tilde{\Sigma}\tilde{\Sigma}^+ &= \tilde{\Sigma} \\ (\tilde{\Sigma}\tilde{\Sigma}^+)^T &= \tilde{\Sigma}\tilde{\Sigma}^+ \\ (\tilde{\Sigma}^+\tilde{\Sigma})^T &= \tilde{\Sigma}^+\tilde{\Sigma} \end{aligned} \tag{5.74}$$

したがって，一般逆行列の一意性から，A^- は

$$A^- = V\tilde{\Sigma}^+\tilde{U}^T \tag{5.75}$$

で与えられることがわかる．

さらに，$\tilde{\Sigma}^+$ の性質と $\tilde{U}$ の定義から，

$$A^- = V\tilde{\Sigma}^+U^T \tag{5.76}$$

と書き換えることができる．

このように，行列 A の一般逆行列は、行列 A の特異値分解要素を用いて計算できる．

5.4 一般逆行列の並列化

本節では，本研究で行った一般逆行列の並列化について説明する．これまで説明したように本研究では特異値分解による一般逆行列の数値計算を行った．一般逆行列の計算を行う対象となる行列を A，特異値分解により分解された行列を U，V，特異値の対角行列を S，計算した一般逆行列を A^- とおくと，式 (5.72) より

$$A = USV^T$$

と表すことができる．次に分解した行列を用いて一般逆行列は，式 (5.75) より

$$VS^-U^T = A^-$$

と表すことができる．

これらの，本研究での並列化を図 5.2 に示す．行列 A に関しては列方向に分割する．特異値分解により分解された行列 U は行方向に分割される．S はすべてのプロセスが保持する．V^T は列方向に分割される．

次に，分解行列を用いて一般逆行列を計算するときに，行列 V はすべてのプロセスが保持する．つまり通信により各プロセスが保持していた V^T の部分行列をすべてのプロセスに送信する．S はすべてのプロセスが保持している．ここで対角行列 S の逆行列 S^- が必要であるが，これは対角行列の性質を利用すると，対角要素の逆数で除算することで計算を進めることができる．行列 U に関しては列方向に分割して各プロセスが保持する．このようにして各プロセスは一般逆行列 A^- の部分行列を計算し，最後にデータを集めて並列処理が完了する．

5.4.1 PureMPI 並列化の適用

PureMPI はすべてのコア数分のプロセスを実行し，MPI 通信により並列処理を進めていく．よって図 5.2 のプロセスの分割がコア数分に分割され細粒度になる．また，本研究では特異値分解に ScaLAPACK ライブラリを使用した．ScaLAPACK 内部で呼び出される LAPACK ルーチンには AMD Core Math Library を用いた．特異値分解以降の計算にはライブラリを用いていない．

5.4.2 NodeHybrid 並列化の適用

NodeHybrid はノード数分のプロセスを実行し，MPI 通信により並列処理を進めていく．よって図 5.2 のプロセスの分割が粗粒度になる．また，特異値分解には ScaLAPACK ライブラリを使用した．ScaLAPACK 内部で呼び出される LAPACK ルーチンには OpenMP により共有メモリ並列化を施された AMD Core Math Library を用いることでハイブリッド並列化を実現した．特異値分解以降の計算にはライブラリを用いていない．

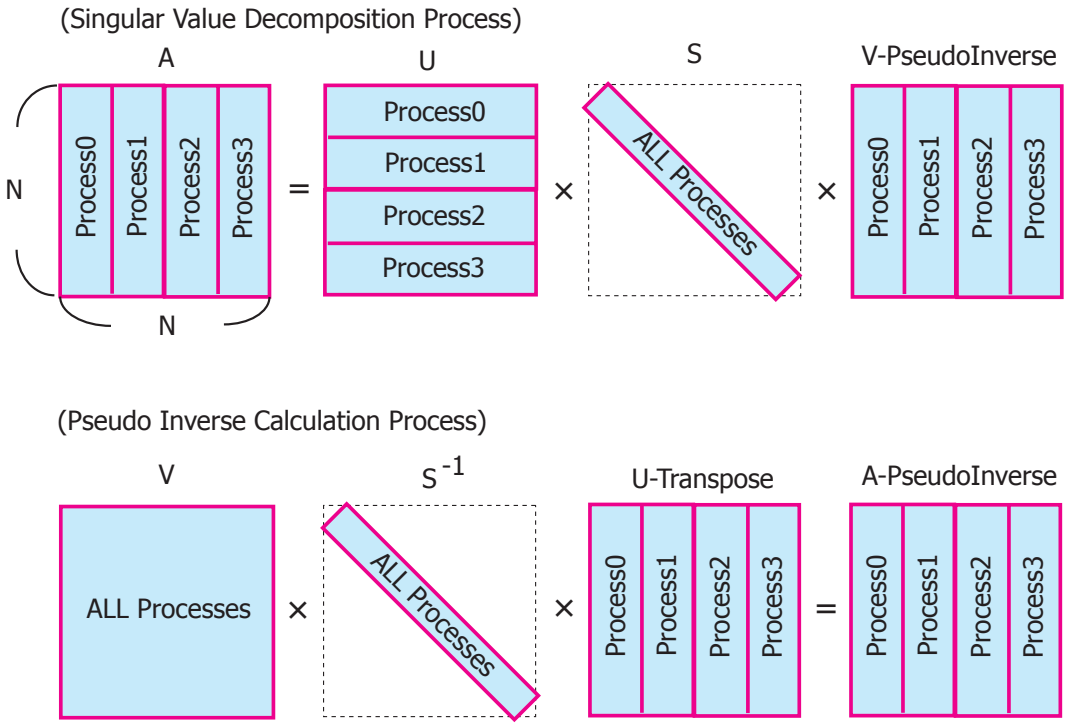

図 5.2: 一般逆行列の並列化

5.4.3 チップレベルハイブリッド並列化 (提案手法) の適用

チップレベルハイブリッド (提案手法) はマルチコアプロセッサチップ数分のプロセスを実行し，MPI 通信により並列処理を進めていく．よって図 5.2 のプロセス分割が中粒度になる．また，特異値分解には ScaLAPACK ライブラリを使用した．ScaLAPACK 内部で呼び出される LAPACK ルーチンには OpenMP により共有メモリ並列化を施された AMD Core Math Library を用いることで提案手法によるハイブリッド並列化を実現した．特異値分解以降の計算にはライブラリを用いていない．

5.5 評価実験

一般逆行列の並列処理について評価を行うために，通信時間の調査，キャッシュヒット率の調査，全体の処理時間の調査及び台数効果の調査を行った．

5.5.1 使用ライブラリ

本研究では，一般逆行列の計算途中で行われる特異値分解で数値計算ライブラリ ScaLAPACK を使用した．本節では，数値計算ライブラリ ScaLAPACK 及び，AMD Core Math

Library の説明を行う．

ScaLAPACK

ScaLAPACK[16] とは，テネシー大学 ICL(Innovative Computing Lab) の J.Dongarra らによって開発された数値計算ライブラリである．長年にわたり行列計算ライブラリの標準化が行われており，その成果として逐次計算ライブラリが LAPACK，分散メモリ型並列計算機用に並列化されたものが ScaLAPACK である．ScaLAPACK には，合計で約 200 種類のルーチンが提供されており，それらのほとんどで実数，複素数データを扱うことができる．また，単精度と倍精度がサポートされている．ScaLAPACK は次のような特徴がある．

- 豊富なライブラリ群
- 簡潔なプログラミングインタフェース
- 高い移植性
- 高い性能
- オープンソース

また，ScaLAPACK は以下のライブラリを用いて構築されている．

- メッセージパッシング・ライブラリ
  プロセス間の通信を行うルーチンを提供するライブラリ
- BLACS
  行列の転送に特化したプロセス間通信ライブラリ
- PBLAS
  BLACS の Interface を用いた，基本的な行列計算を分散メモリ型並列計算機上で効率よく行うためのライブラリ
- LAPACK
  多数の計算ルーチンを提供する線形計算ライブラリ

本研究では，一般逆行列の計算内部で行われる特異値分解に ScaLAPACK と ACML を組み合わせて使用した．

AMD Core Math Library(ACML)

AMD Core Math Library(ACML)[17] とは，AMD 社が提供する AMD64 プロセッサに最適化された数値演算ルーチンのセットである．ACML には以下のような特徴がある．

- BLAS のレベル 1，レベル 2，レベル 3 をすべて実装し，AMD プロセッサで高い性能が出るように主なルーチンを最適化してある．
- LAPACK ルーチンをすべて実装している．重要な部分に更なる最適化を施しており，LAPACK の標準的な実装よりもはるかに高い性能を得ることができる．
- 高速フーリエ変換 (FFT) は，データタイプとして単精度，倍精度，単精度複素数，倍精度複素数に対応している．

ACML は GCC や PGI，Intel など様々なコンパイラで利用できる．また OpenMP により共有メモリ並列化を施されたバージョンのライブラリも提供している．本研究では ScaLAPACK と ACML を組み合わせて使用している．これは ScaLAPACK 内部で呼び出される LAPACK ルーチン部分を ACML の LAPACK ルーチンに置き換えることにより可能となる．これはライブラリのリンク時に LAPACK 部分を ACML に置き換えるだけで簡単に利用できる．また本研究ではハイブリッド並列化を行っている．その場合には，OpenMP により並列化された ACML ライブラリをリンクすることにより MPI/OpenMP ハイブリッド並列化となった ScaLAPACK が利用できるのである．

5.5.2 通信時間の調査

一般逆行列の並列処理に関して，従来手法 (PureMPI，NodeHybrid) 及び提案手法を適用し通信時間の調査を行った．この実験で従来手法及び提案手法の通信時間と演算時間の違いを明らかにする．

図 5.3 にノード数 2(コア数 16) の実験結果を，図 5.4 にノード数 4(コア数 32) の実験結果を示す． 通信時間に着目すると，ノード数 2(コア数 16) のときには従来手法，提案手法ともに大きな差は見られなかった．しかしノード数 4(コア数 32) のときには従来手法の PureMPI が著しく通信時間が増加している．これは PureMPI がすべてのコアと通信を行っているために，通信・同期回数がボトルネックとなっているからであると考えられる．一方，提案手法と NodeHybrid の通信時間を比較すると，NodeHybrid の方が少し増加しているのが見て取れる．これは通信回数は NodeHybrid がもっとも少ないが，逆に一回の通信データ量が大きいために，ノード間での通信処理がボトルネックとなっているからであると考えられる．つまり提案手法が通信回数，通信データ量の点で最適であったために従来手法よりも優れた結果が出たと考えられる．

次に演算時間を見ると，PureMPI が最も良く，次に提案手法，最後に NodeHybrid が著しく増加するという結果になっている．PureMPI と比較して提案手法及び NodeHybrid

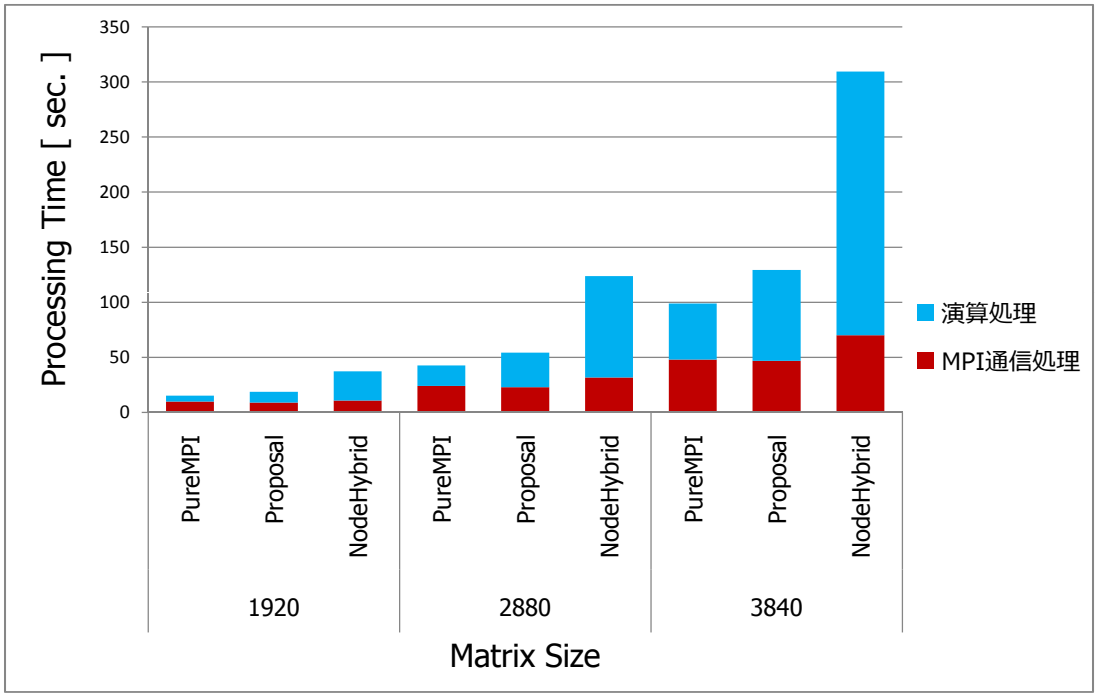

図 5.3: 一般逆行列 - ノード数 2(コア数 16) の解析結果

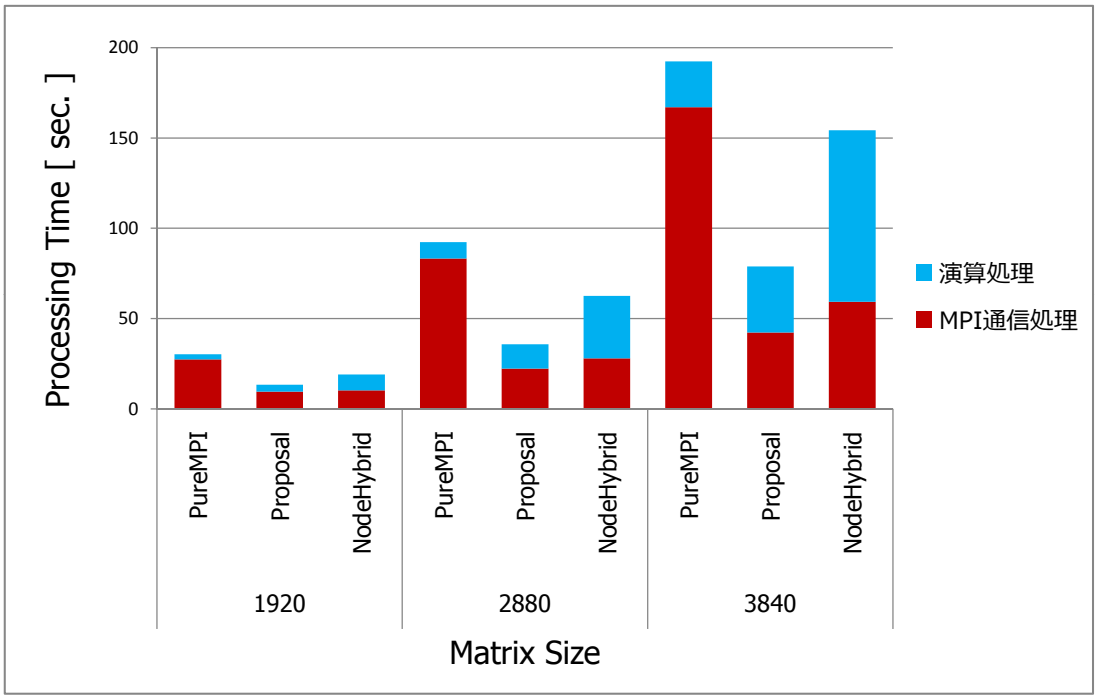

図 5.4: 一般逆行列 - ノード数 4(コア数 32) の解析結果

が増加している点は，キャッシュヒット率もしくはOpenMPによる共有メモリ並列化を行ったことによるオーバーヘッドなどが考えられる．この点を考察するために，次にキャッシュヒット率の調査を行った．

5.5.3 キャッシュヒット率の調査

従来手法及び提案手法について，キャッシュヒット率の調査を行った．この実験により，従来手法及び提案手法のキャッシュヒット率の違いを明らかにする．

表5.1に行列一辺のサイズが1920のときの，使用ノード数2及びノード数4の結果を，表5.2に行列一辺のサイズが3840のときの，使用ノード数2及びノード数4の結果を示す．また，ここでもAverage memory access time(平均メモリアクセス時間)の算出を行った．詳細については4.3.2節で説明したので参照して頂きたい．つまりここで示す平均メモリアクセス時間はあくまで目安である．導出に使用したキャッシュやメモリのレイテンシは参考文献[7]の値を利用した．

表 5.1: 一般逆行列 - 行列一辺サイズ N=1920 のキャッシュヒット率

N=1920	PureMPI		Proposal		NodeHybrod	
num of Nodes	2	4	2	4	2	4
L1 data cache hit (%)	99.6	99.6	98.0	98.0	98.1	98.1
L2 cache hit (%)	89.8	89.7	88.2	88.0	88.5	88.4
L3 cache hit (%)	4.49	4.81	8.16	9.03	8.43	9.15
Average memory access time (ns)	1.48	1.49	1.72	1.72	1.70	1.69

表 5.2: 一般逆行列 - 行列一辺サイズ N=3840 のキャッシュヒット率

N=3840	PureMPI		Proposal		NodeHybrod	
num of Nodes	2	4	2	4	2	4
L1 data cache hit (%)	99.6	99.6	98.1	98.0	98.1	98.2
L2 cache hit (%)	89.9	89.8	88.7	88.8	88.8	88.8
L3 cache hit (%)	8.10	7.73	9.41	9.35	9.08	9.23
Average memory access time (ns)	1.48	1.48	1.70	1.70	1.69	1.68

行列サイズ N=1920 と N=3840 で比較すると，キャッシュヒット率の傾向はほぼ同じである．次に、並列化手法で比較すると，L1 データキャッシュのヒット率はPureMPIの方が約1.5%程度優れており，提案手法とNodeHybridは同じ程度の結果となった．この違

いはOpenMPによる共有メモリ並列化を行っているかどうかの違いであると考えられる．OpenMPによる共有メモリ並列化はデータを共有して並列処理を行うために，あるスレッドが共有データへのアクセスを行った場合には，キャッシュ内でそのデータがあるラインはキャッシュの一貫性を保つためにアクセスできなくなる．これをFalse Sharingというが，この問題により共有メモリ並列化を行っている場合にはどうしてもヒット率が下がる傾向にあるようである．この傾向はL2キャッシュヒット率にも見て取れる．しかしL3共有キャッシュのヒット率を見ると共有メモリ並列化を行った提案手法とNodeHybridが優れた結果となった．これはPureMPIがメッセージパッシングによるプロセス並列化を行っているために，共有キャッシュ全体を利用することができない一方，提案手法とNodeHybridはスレッドによる並列化で共有キャッシュ全体を利用できるからである．L3共有キャッシュに関しては提案手法が最も良いか，もしくはNodeHybridと同じヒット率になるのではないかと予想していた3.6.3が，実際にはNodeHybridと同程度という結果であった．

通信時間の調査結果では，NodeHybridの演算時間が非常に増加しており，次に提案手法の演算時間が長かった．キャッシュヒット率の違いではないかと考えられたが，本節の結果を見るとNodeHybridと提案手法は同じようなヒット率であり，平均メモリアクセス時間を見てもほとんど同じ程度である．これらの結果から総合して，キャッシュヒット率低下による演算時間の増加ではなく，PureMPIと比較して提案手法及びNodeHybridの演算時間増加はOpenMPを利用したスレッド並列化によるスレッドの生成や同期処理によるオーバーヘッドが原因であると考えられる．事実，ノード内のコアすべてを用いてスレッド並列化を行っているNodeHybridの演算時間が著しく増加していることからも，OpenMPによるオーバーヘッドの可能性が最も高いと思われる．

5.5.4 処理時間の調査

従来手法及び提案手法について，全体の処理時間の調査を行う．この実験により，行列サイズを変化させたとき及び使用ノード数を変化させたときの，全体の処理時間の変化を明らかにする．

図5.5から図5.8に，ノード数1(8コア)からノード数4(32コア)までの行列サイズを変化させたときの処理時間を示す．

使用ノード数1(コア数8)及び2(コア数16)ではPureMPIが最も短い処理時間となった．これは使用コア数が少ないためにPureMPIでの通信によるオーバーヘッドが少なく，提案手法及びNodeHybridではOpenMPによるオーバーヘッドがあるためであると考えられる．使用ノード数3(コア数24)では行列サイズによって提案手法が最も短い処理時間の場合やPureMPIが最も短い処理時間の場合が見られる．使用ノード数4(コア数32)になると提案手法が最も短い処理時間となり，PureMPIは通信によるオーバーヘッドが大きく最も長い処理時間となっている．使用ノード数4の行列サイズ4160では提案手法がPureMPIの処理時間の57%と優れた結果が示されている．

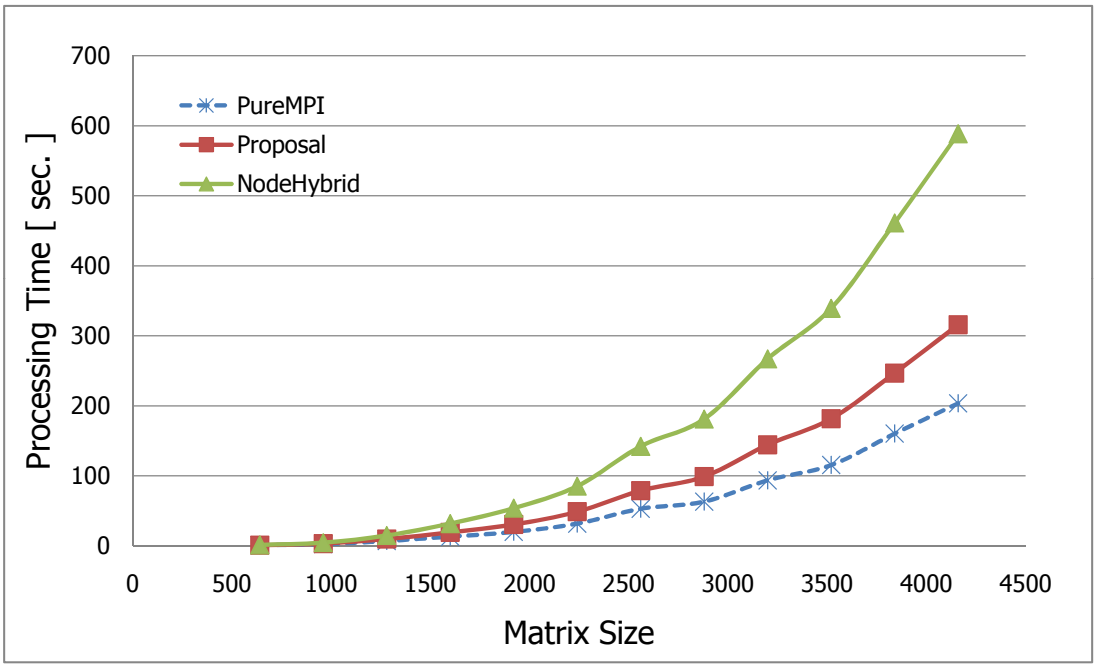

図 5.5: 一般逆行列 - ノード数 1(コア数 8) の処理時間

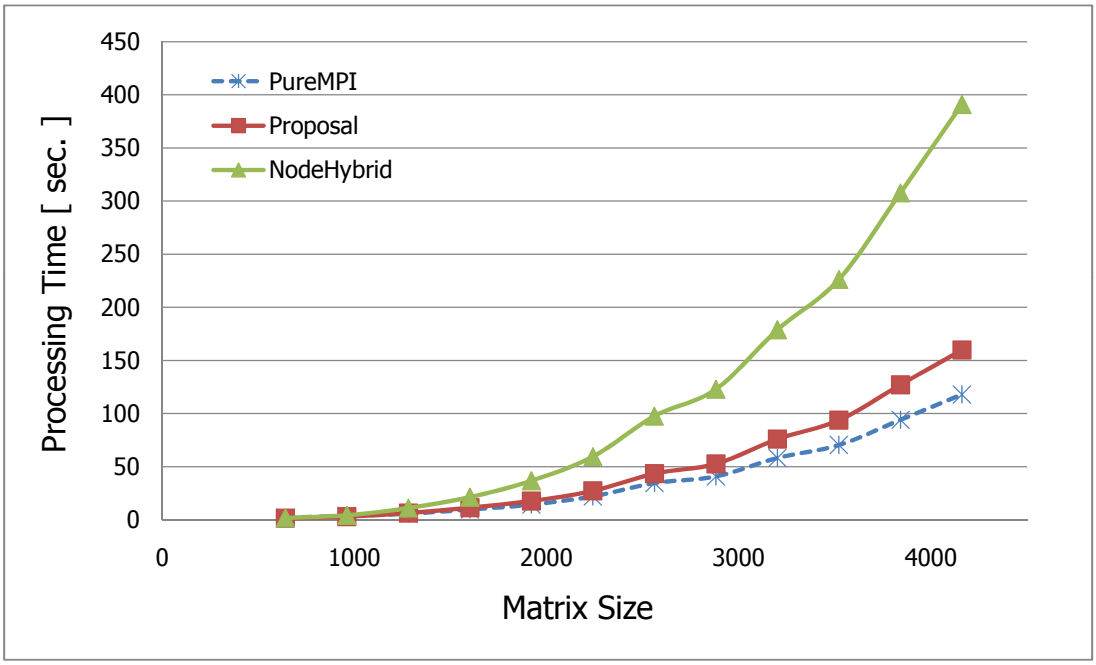

図 5.6: 一般逆行列 - ノード数 2(コア数 16) の処理時間

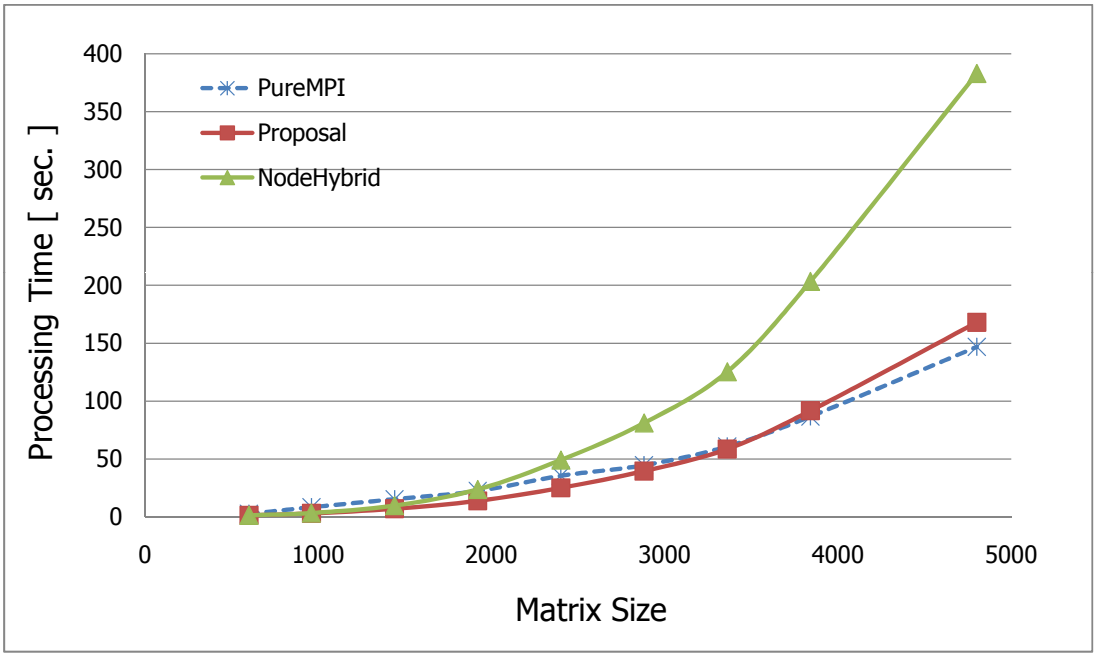

図 5.7: 一般逆行列 - ノード数 3(コア数 24) の処理時間

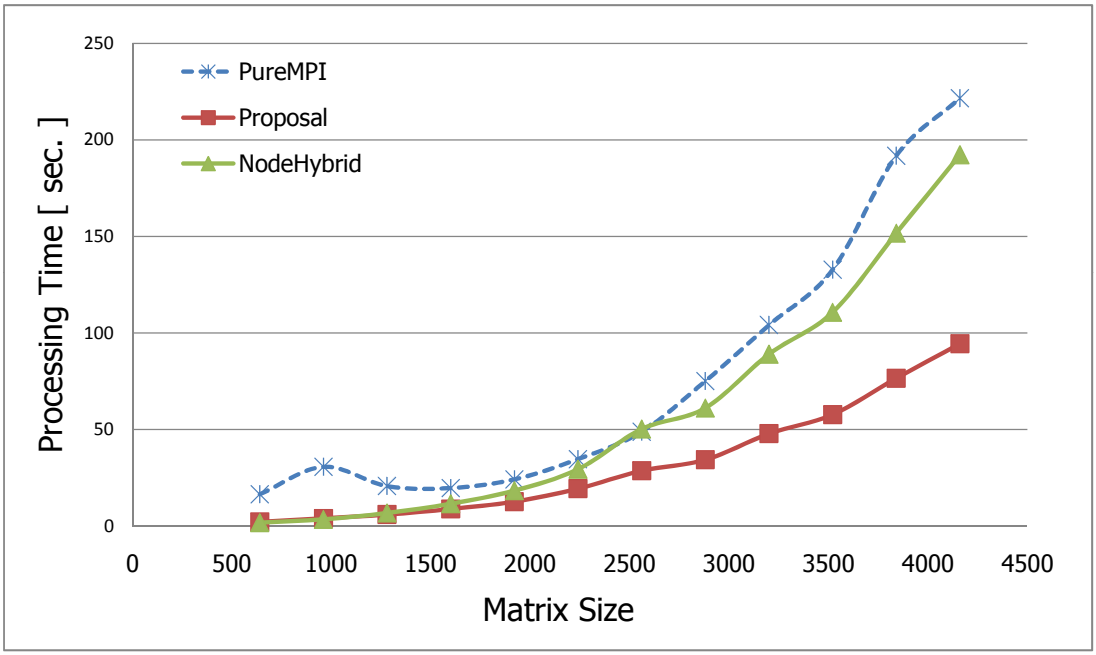

図 5.8: 一般逆行列 - ノード数 4(コア数 32) の処理時間

5.5.5 台数効果の調査

従来手法及び提案手法について台数効果 (Speedup) の調査を行う．この実験により，ノード数が増加した際の従来手法と提案手法の並列処理性能の違いを明らかにする．Speedup は逐次による処理時間との比較である．行列サイズ N=960 のときの Speedup を図 5.9 に，N=1920 を図 5.10，N=2880 を図 5.11，N=3840 を図 5.12 に示す．

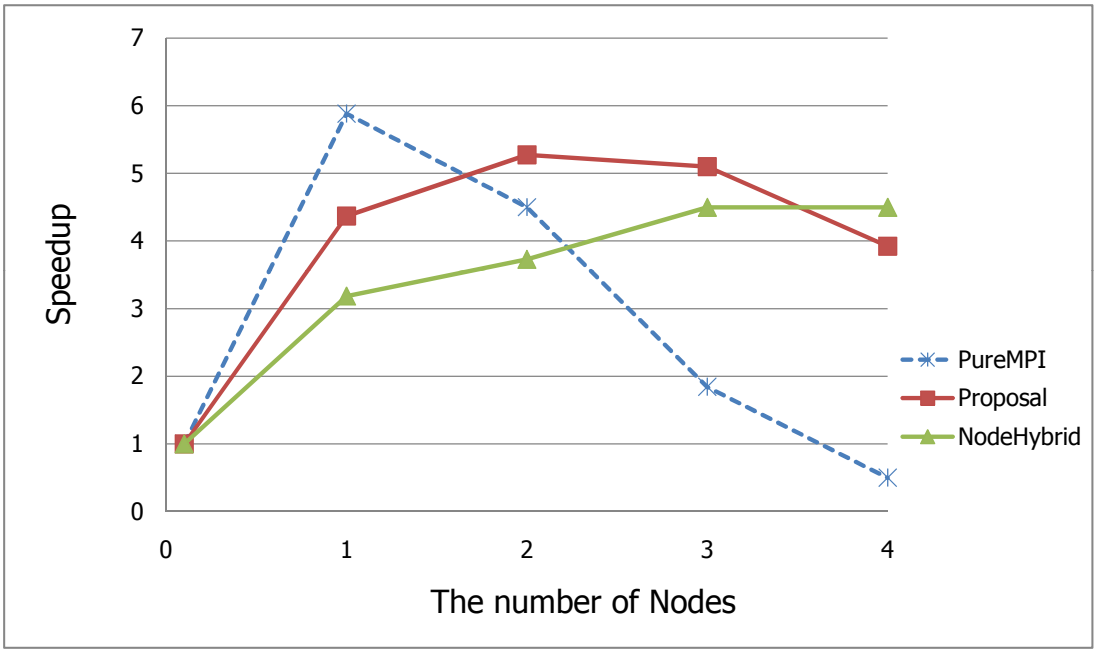

図 5.9: 一般逆行列 - 行列一辺のサイズ N=960 の Speedup

行列サイズ N=960 の結果では従来手法，提案手法ともに台数効果が表れていない．これは行列サイズが小さすぎたために，逐次処理と比較して並列処理によるオーバーヘッドが大きいからである．行列サイズ N=1920 と N=2880 では，使用ノード数 2(16 コア) までは PureMPI がもっとも良い Speedup を示しているが，ノード数 3 及び 4 になると提案手法が最も良い Speedup を示している．これはノード数 (コア数) が増加すると通信によるオーバーヘッドで PureMPI の性能が低下しているからである．行列サイズ N=3840 ではノード数 3(24 コア) までは PureMPI が良い Speedup を示しているが，ノード数 4 では提案手法が最も良い Speedup を示している．これも同じく，ノード数増加による PureMPI の通信オーバーヘッド増加で性能低下が起きたからである．

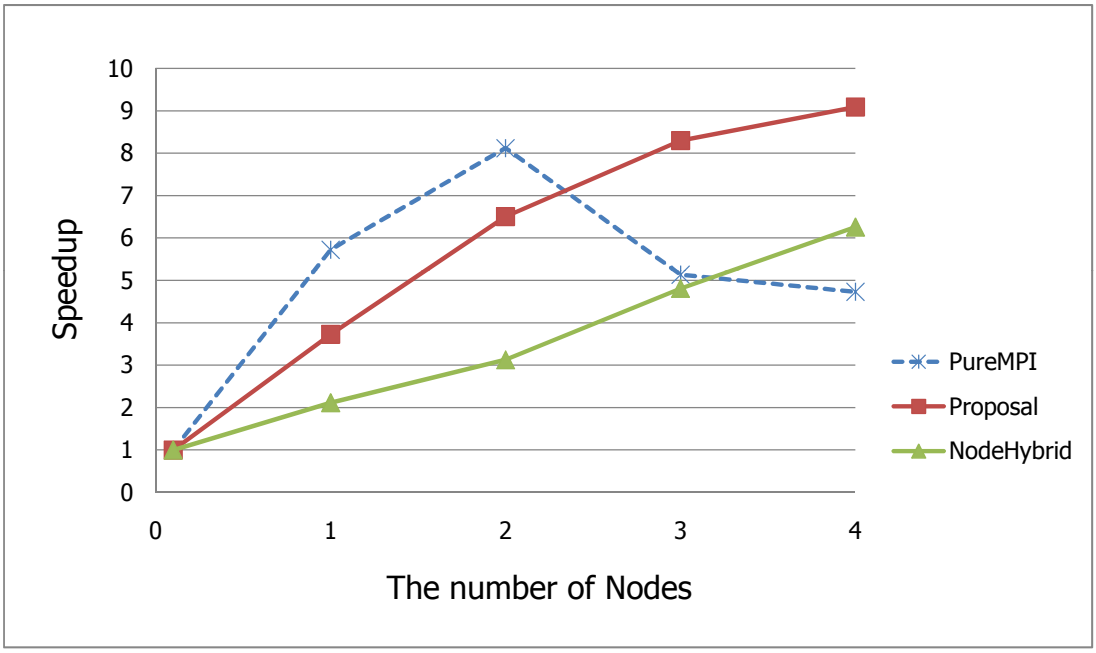

図 5.10: 一般逆行列 - 行列一辺のサイズ N=1920 の Speedup

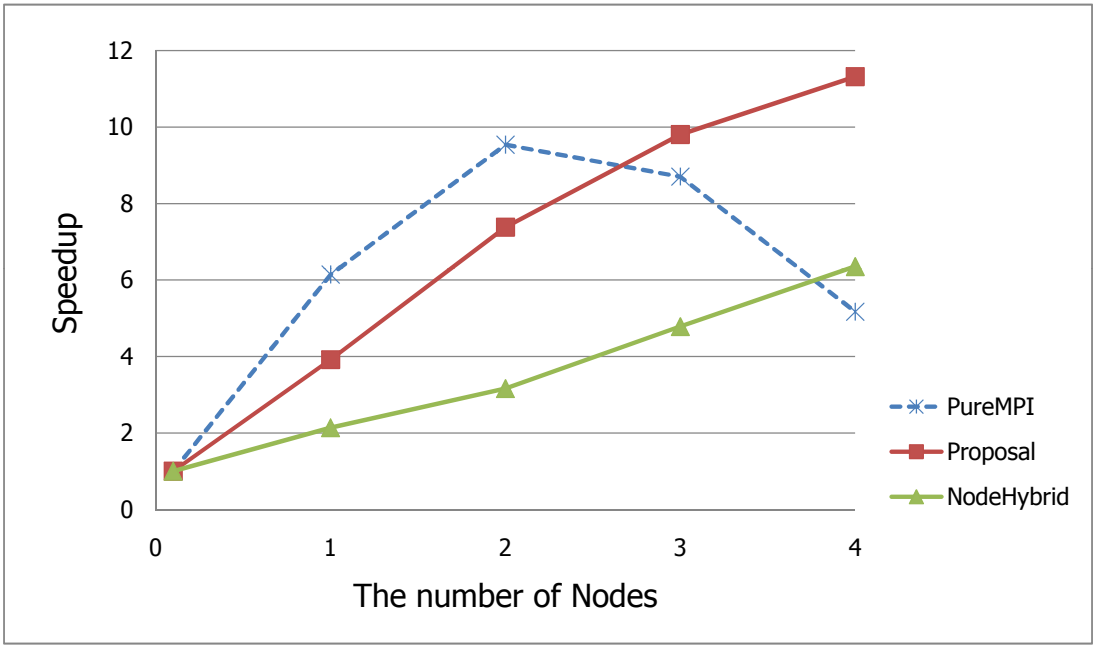

図 5.11: 一般逆行列 - 行列一辺のサイズ N=2880 の Speedup

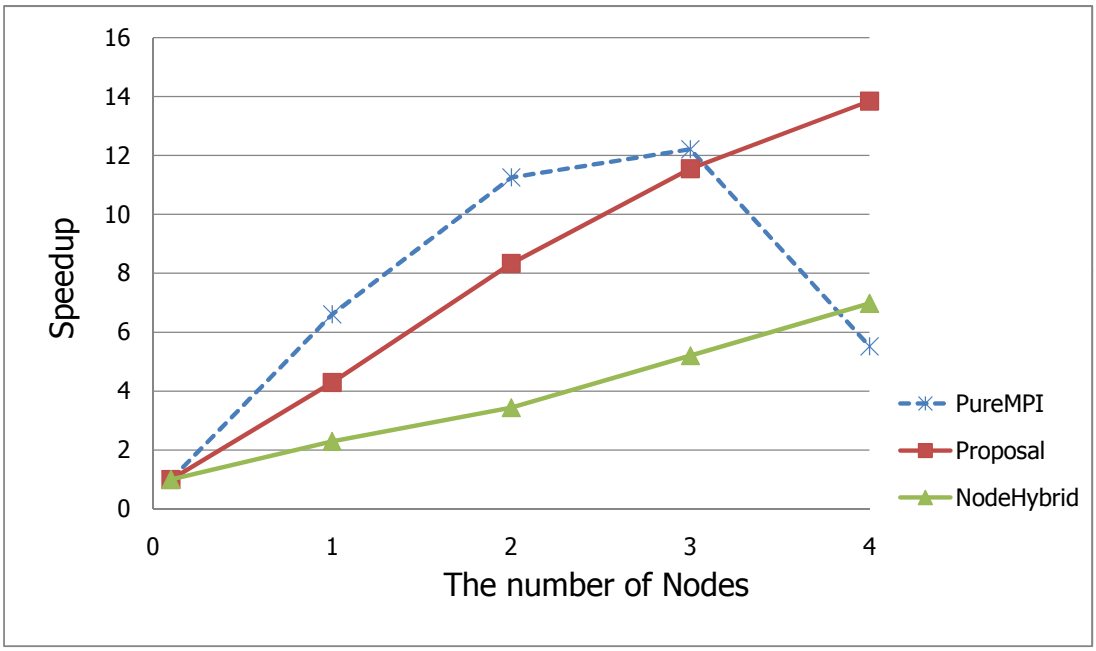

図 5.12: 一般逆行列 - 行列一辺のサイズ N=3840 の Speedup

5.6 まとめ

本章では，実際のアプリケーションによる評価として一般逆行列の数値計算をとりあげ，実験により提案手法の有効性を示した．一般逆行列の数値計算は，近年注目されている生体内の構造を画像化する技術である蛍光トモグラフィーにおいてボトルネックとなっている処理である．

まずはじめに通信時間の調査を行った．使用ノード数 2(16 コア) では各並列化手法において通信時間の差はほとんど見られなかった．一方，使用ノード数 4(32 コア) では従来手法の一つである PureMPI の通信時間が最も長くなっていることがわかった．これは PureMPI がすべてのコアと通信を行っているために，コア数が増えると通信・同期回数がボトルネックとなるからである．演算時間では PureMPI がもっとも短く，次に提案手法，最も長いのは NodeHybrid であることがわかった．この原因を調査するために，次にキャッシュヒット率の調査実験を行った．結果を見ると、コア間独立の L1,L2 キャッシュのヒット率は PurePMI が最も良く，コア間共有キャッシュの L3 キャッシュは提案手法と NodeHybrid が良い結果を示した．また提案手法と NodeHybrid の平均メモリアクセス時間を見るとほとんどその差はない．しかし，NodeHybrid の演算時間増加が起きている．これらの結果を総合すると，NodeHybrid の演算時間の増加はキャッシュヒット率の差ではなく，OpenMP によるスレッド並列化のオーバーヘッドである可能性が高いことがわかった．全体の処理時間の調査では，使用ノード数 3(24 コア) 程度までは PureMPI が良

い性能を示していたが，ノード数 4(コア数 32) になると提案手法が最も良い性能を示すことが示された．これは通信時間の調査からわかるように，コア数増加による PureMPI の通信時間がボトルネックとなるからである．最後に台数効果の結果では，ノード数が増えると提案手法が最も良い Speedup となることが示された．

以上の結果から，使用ノード数 1 から 3 の場合には PureMPI による並列化が良い性能を示すが，使用ノード数が 4(32 コア) になると提案手法による並列化が最も良い性能を示すことがわかった．

第6章　結論

本研究ではマルチコアプロセッサをノード内に複数搭載したクラスタを対象とし，ノード間及びマルチコアプロセッサのチップ間ではMPIによるメッセージパッシング並列化を行い，チップ内では共有キャッシュが存在することからOpenMPによる共有メモリ並列化を行う，チップレベルMPI/OpenMPハイブリッド並列化を提案した．また，本提案の有効性を調査するために，基礎的な数値計算による評価として行列積に，また実際のアプリケーションによる評価として一般逆行列に適用し，PureMPI及びNodeHybridとの比較実験を行った．ここで，PureMPIとはすべてのコアでメッセージパッシングによる並列化を行う手法であり，NodeHybridとはノード間をMPIメッセージパッシング並列化，ノード内のすべてのコアでOpenMPによる共有メモリ並列化を行う手法のことである．本研究で得た結論を以下に示す．

- 行列積による評価実験で本提案手法は，NodeHybridと比較して大幅な並列性能向上が見られた．またPureMPIと比較してほぼ同じ並列性能が得られた．
- 一般逆行列による評価実験では，すべてのノードを使用した場合，行列一辺サイズN=3840でNodeHybridと比較して51%の処理時間短縮を示した．またPureMPIと比較しても57%の処理時間短縮を示した．
- PureMPIはMPIによる通信オーバーヘッドを大きく受ける．一方NodeHybridはOpenMPによる共有メモリ並列化のオーバーヘッドを大きく受ける．本提案手法はその影響が少なかったために，従来手法よりも高性能が得られたと考えられる．
- マルチコアプロセッサを搭載したクラスタを想定した場合，本提案手法は有効な並列化手法であると考えられる．

マルチコアプロセッサが普及してきた現在，その性能を最大限発揮できるような並列プログラミングモデルを選択することは非常に重要な問題である．今後はさらに，1つのプロセッサパッケージ内に含まれるコア数が増加すると思われる．そのような状況で，すべてのコアでメッセージメッセージパッシングを行う並列モデルは通信処理の増加や共有キャッシュを利用できないなど，そのデメリットは大きい．一方，ノード内のすべてのコアでOpenMP共有メモリ並列化を行うのもまたオーバーヘッドがあり性能が出ない場合がある．そのような状況の中，本提案手法は，マルチコアプロセッサを搭載したクラスタで並列処理を行う際の最適な並列モデルの１つになるといえる．

6.0.1 今後の課題

本研究では行列積及び一般逆行列に提案手法を適用して，その有効性を示した．しかし本提案手法が有効に働くアプリケーションや計算機の具体的な条件について明確にはしていない．当然ながらどのようなアプリケーションでも，どのようなマルチコアクラスタの計算機性能でも有効なのではなく，最適に働く適用条件があるはずである．今後は本提案手法が有効な範囲について明確に提示できるようにする必要があると考えられる．

謝辞

本研究を行い論文をまとめるにあたり，私を支えていただいたすべての方々へ，心からお礼を申し上げます．

井口寧准教授にはご多忙の中，また今年は普段と異なる環境という中，ゼミでのアドバイスや研究の進め方などいつもと変わらない熱心なご指導を頂きました．本当に有難うございました．また，佐藤幸紀助教には，参考論文の紹介やゼミでの的確なアドバイス，また多くのご指導を頂きました．本当に有難うございました．また，適切なご指導を賜りました松澤照男教授，日比野靖教授，田中清史准教授に深く感謝致します．

そして研究室ではドクターの荒木さん，Duy さん，また高橋さん，近藤さんには先輩として未熟な私に多くの面で指導して頂きました．また田中研究室ドクターの請園さん，松本研究室のメンバー，山下君には研究以外でも楽しい思い出が多く，素晴らしい研究生活を送ることができました．

そして，最後になりましたが，これまで遠方より支えてくださった両親に心より深く感謝いたします．

多くの方々のご指導，ご支援によりこの研究生活が送れたことに感謝致しまして，結びに変えさせて頂きます．

平成 21 年 2 月 5 日　中尾　哲也

参考文献

[1] TA QUOC VIET, TSUTOMU YOSHINAGA, BEN A.ABDERAZEK and MASAHIRO SOWA Construction of Hybrid MPI-OpenMP Solutions for SMP Clusters IPSJ Transactions on Advanced Computing Systems, 46, No.SIG 3(ACS8), pp.25-37, 2005.

[2] 中島研吾，階層型領域分割によるマルチステージ並列前処理手法へのハイブリッド並列プログラミングの適用，IPSJ SIG Technical Report 2007-HPC-110, Vol.2007, No.59, pp.25-30.

[3] 馬場慎也，尾上勇介，南里豪志，藤野清次，ハイブリッド並列化した IDR(s) 法の計算時間に対するプロセス数とスレッド数の組み合わせ依存性について，IPSJ SIG Technical Report 2008-HPC-115, Vol.2008, No.43, pp.13-18.

[4] 山田進，今村俊幸，町田昌彦，荒川忠一，共有分散メモリ型並列計算機における新規通信手法，Transactions of JSCES, Paper No.20050010，2005.

[5] Edward E.Graves, Jorge Ripoll, Ralph Weissleder, and Vasilis Ntziachristos, A submillimeter resolution fluorescence molecular imaging system for small animal imaging, Medical Physics, Vol.30, No.5, 2003.

[6] Tian Tian, Chiu-Pi Shih, Software Techniques for Shared-Cache Multi-Core Systems Intel Software Network, July 9, 2007, http://softwarecommunity.intel.com/articles/eng/2760.htm

[7] Tech ARP - AMD Quad-Core Opteron (Barcelona) Technology Report http://www.techarp.com/showarticle.aspx?artno=424&pgno=2

[8] 安田絹子, 小林林広, 飯塚博道, 阿部貴之, 青柳信吾, マルチコア CPU のための並列プログラミング秀和システム, 2006.

[9] P. パチェコ，MPI 並列プログラミング，培風館，2001.

[10] 牛島省，OpenMP による並列プログラミングと数値計算法，丸善，2006.

[11] 半谷裕彦，川口健一, 形態解析 一般逆行列とその応用, 培風館, 1991.

[12] 大野豊，磯田和男，数値計算ハンドブック，オーム社，1990.

[13] TOP500 Supercomputer Sites: http://www.top500.org/

[14] MPICH Home Page: http://www-unix.mcs.anl.gov/mpi/mpich1/

[15] OpenMP.org: http://openmp.org/

[16] ScaLAPACK: http://www.netlib.org/scalapack/

[17] AMD Core Math Library(ACML): http://www.amd.com/acml/

[18] TAU - Tuning and Analysis Utilities: http://www.cs.uoregon.edu/research/tau/

[19] AMD CodeAnalyst: http://developer.amd.com/cpu/CodeAnalyst/

本研究に関する発表論文

[1] 中尾哲也，佐藤幸紀，井口寧，マルチコアクラスタに対するマルチコアプロセッサを意識した並列処理手法に関する研究， 2008 年度電気関係学会北陸支部連合大会，E-62，Sep 2008.