実行時の情報を用いてプロセッサ間の通信を最適化するコンパイラ

工学研究科
筑波大学

2003年3月

横田 大輔

概要

自然科学の分野で用いられるシミュレーションプログラムの実行には莫大な時間がかかる. しかし, このようなプログラムは実行時間に比べてコード長が短く, 特定の部分を莫大な回数繰り返す. この特定の部分は実行時間を決定する要因になるので, この箇所を強力に最適化することは極めて重要である. そこで, このようなプログラムをコンパイルする際, 莫大な回数繰り返される部分に時間がかかっても効果的な最適化を施すコンパイラが必要である. また, このようなプログラムは通信がボトルネックになっていることが多く, 通信を最適化することが重要である.

本研究では実行時の情報を用いることで, 容易にプログラミングでき, 効果的にハードウェアを利用するコンパイラを実装した. 本研究のコンパイラは実行時の情報を得るために, インスペクタ-エグゼキュータ手法を利用する. また, 本研究のコンパイラは実行時の情報を用いて行うことが困難な最適化を行なった. 例えば, 実行時の情報を用いた定数の畳み込みのようなコードの最適化, パラメータが定数的でかつ繰り返し用いることでオーバーヘッドが削減できる通信機構の利用などである. これらの最適化のためには, コード生成時に実行時の情報が必要になる.

本研究では実行時の情報をコンパイル時に利用するために, ソースコードからコンパイル中に実行時の情報を集めるための専用コードを生成するようにした. これをコンパイル中に実行することで, 実行時の情報をコンパイル時に利用可能になした.

本研究で実際に実装したコンパイラは 2 種類あり, 1 台の PC 上で動作するものと PC クラスタで動作するものがある. 1 台の PC 上で動作するコンパイラは簡便で高速なコードを生成するが, 受理できるプログラムにいくつか制限が加わる. PC クラスタで動作するコンパイラはより制限が加わらない分汎用的である. また, 開発したコンパイラがどの程度の性能があるのか確認するために, いくつかベンチマークを用いて実験を行なった. 本方式の 1PC 版は人間のプログラマが最適化した MPI のコードに比べて pde1 ベンチマークで 86%の速度, PC クラスタ版は人間のプログラマが最適化した MPI のコードに比べて pde1 ベンチマークで 73% の速度を得られた.

謝辞

筑波大学 電子・情報工学系 板野肯三教授には, 私が板野研究室に所属している 4 年間の間, 非常に親身に指導にあたっていただいた. 本研究においては, 研究の進め方, 論文の書き方など多岐にわたり代えることのできない貴重な意見をいただいた. また研究環境から人間関係に至るまで大変恵まれた環境を与えていただけて, 非常に研究の助けとなった. 師から受けた恩の数は枚挙に暇がない. また, 様々な場面において, 非常に有意義な議論を行うことができた. ここに深く感謝の念を表す.

東京工業大学 千葉滋助教授には, 本研究に対して有益なコメントをいただき, また研究発表に関する指導をしていただいた. ソフトウェアの改良に関するアドバイス, 発表の進め方など基礎的なことから高度なことまで様々な貴重な意見をいただいた. 筑波大学の学生である私に惜しみもなく貴重な時間を割いていただいた. ここに深く感謝の念を表す.

筑波大学 電子・情報工学系 新城靖講師には, 研究に関するアドバイスをいただいた. また, 筑波大学 電子・情報工学系 佐藤三久教授, 和田耕一教授, 田中二郎教授, 加藤和彦助教授, 東京工業大学 千葉滋助教授には, 貴重な時間を割いて審査していただき, 本研究に関して多様な意見をいただいた. 以上の方々に, ここに心から感謝の念を現す.

目 次

図 目 次

表 目 次

第1章　はじめに

今日, コンピュータを用いた大規模シミュレーションは, 物理や天文のような自然科学の分野の物理現象の解明に広く使用されている. このような自然現象のシミュレーションには1000以上のノード数を持った超並列計算機を使用することも多い. 並列計算機上でのシミュレーションでは, シミュレーションの対象空間を細かく区切って, 現象の経時変化を単位時間毎に繰り返すために, プロセッサに処理を分担させる. この種のシミュレーションは, 1回の実行で莫大な時間がかかるのが普通であり, 計算物理や天文のシミュレーションでは1週間またはそれ以上の時間がかかるものも希ではない. 並列計算機上でシミュレーションのプログラムを実行させる場合には, プロセッサ間で多量のデータを交換するので, プロセッサ間通信を高度に最適化する必要がある. しかし, 応用分野のプログラマはコンピュータの専門家ではなく, 専門的なハードウェアの知識を期待できないので, プログラマが手で通信を最適化することは現実的ではない. したがって, この最適化をコンパイラが行う必要があり, 仮にコンパイルに莫大な時間がかかるような最適化でも, シミュレーション時に効果があれば, 適用することには大きな意味があると考える.

1.1　計算物理のシミュレーション環境

コンパイラが最適化を行う場合, ターゲットマシンのアーキテクチャを効果的に利用できるコードを生成することが不可欠になる. 一般に, 計算物理のシミュレーションの計算量は莫大なので, ソフトウェアやハードウェアに高いスループットが要求される. このため, このようなシミュレーションプログラムを実行する計算機には, スループットを向上するために, 数1000のオーダーのノードを持った超並列の計算メカニズムや, 特定の物理計算に特化した計算メカニズムといった, 一般的なPCとは異なるアーキテクチャを採用することが多い.

特定の物理計算に特化した計算メカニズムを持つ計算機としては, 筑波大学計算物理学センターのQCDPAXや東京大学のGRAPE[43]シリーズなどがある. QCDPAXはQCD（Quantum Chromodynamics）と呼ばれる素粒子物理のシミュレー

ションに特化した計算機である. また, GRAPE シリーズは n 体問題を解くことに特化した計算機で, 天文学のシミュレーションに用いられる.

汎用の計算を目的にした計算機には筑波大学計算物理学センターの CP-PACS[40], Pilot-3[40], 日立の SR8000 や富士通の PRIMEPOWER シリーズなどさまざまな計算機がある. これらの計算機には, 汎用的な設計として, 超並列計算でボトルネックになりがちなノード間通信を高速に行うことができるようにするために, 独自な仕様の通信ハードウェアを装備している.

CP-PACS と Pilot-3 は, 筑波大学計算物理学センターが所有する汎用目的に設計された超並列計算機である. この計算機は分散メモリ機で, RDMA (Remote DMA)[44] と呼ばれる特殊な通信ハードウェアを持っている. CP-PACS は 2048, Pilot-3 は 128 個のノードを持ち, それぞれ 3 次元と 2 次元のハイパークロスバーで接続されている. CP-PACS の理論ピーク性能は 614Gflops であり, 両機のノード間通信のピーク性能は 300Mbytes/sec である. RDMA がサポートする特殊な機能にはブロックストライド通信, TCW 再利用型通信, 片側通信などがある. この RDMA のハードウェアが持つ特殊な機能を利用するためには, MPI[33][34][45], PVM[36][35][45], HPF[31][46] ではなく RDMA 独自のライブラリを用いなければならないという制約がある. RDMA に関しては 2.2.4 節で詳しく説明する.

1.2　高度な最適化の必要性

最適化の研究自体は長い歴史を持つが, 並列計算機における最適化は, 複雑度の高い処理であり, 一般性のある静的最適化アルゴリズムを見つけるのは困難な状況にある. このため, 最近の並列計算の最適化の研究は, 実行時の情報を用いるものが多くなっている. 実行時の情報を用いる方法は実際に実行中の情報を得ることができるので, 静的な解析でうまく処理できない場合でも最適化を行うことができるという利点がある. しかし, 実行時の情報を用いる最適化では, 実行中のプログラムの解析が必要であり, プログラムが実行時に自分自身の情報を集め, それをもとに自分自身の動作を変更するという作業を実行時に行わなくてはならない. 具体的には, 実行中に特定の範囲の実行時間や発生した通信などを記録し, 自分自身の挙動を決めるパラメータ (変数) の値を変えるか, コードを書き換える. このために, 当然のこととして, 実行時にオーバーヘッドが生じる.

一方, コンパイラが, プログラムのコードを解析して, 実行時の様子を推測して最適化を行う従来の静的なコンパイルの手法にも限界がある. そこで, これを改善するには, 実行時の情報をコンパイル時に利用できればよいが, 一般には, 実行す

る前にコンパイルは終了しているので, 実行時の情報をコンパイル時に用いることはできないという矛盾に陥る. これを打開するには, 図 3.1 に示すように, コンパイル中にソースコードの一部を解析のために実行し, 実行時の情報を取得して, これを最適化の処理に利用すればよい. この手法の不利な点は, コンパイル中にコードの実行を行うので, コンパイルの時間が大きくなることであるが, 最終的な実行の段階での時間短縮でこれがカバーできれば, 全体としては, 効果があることになる.

一方, この種のシステムのユーザが, コンピュータの専門家ではない自然科学系の研究者であることを考えると, シミュレーションのプログラムが, 間違いなく簡単にプログラムできるようなシステムを提供することは重要な要素である. この意味で, 言語の選択も重要な課題である. HPF[31][46] は自然科学系の研究者には, 標準的なコンパイラであるが, RDMA (Remote DMA) [44] などの特殊な通信ハードウェアを効果的にサポートしていない. レベルがハードウェアに近くて抽象化されていないライブラリインターフェースを利用しなければならないという問題がある. 多くのプログラマはシミュレーションのプログラムを記述する際, 簡潔に記述でき可搬性の優れている, MPI[33][34][45] や PVM[36][35][45] のようなライブラリを利用することを好む傾向がある. したがって, 利用者にとって使いやすいインターフェイスを持つコンパイラをベースにして最適化アルゴリズムを組み込む必要があると考える.

1.3　新しい最適化方式の提案

本研究では, プロセッサ間通信に最適化されたコードを生成するために, 実行時の情報を用いることによって発生するオーバーヘッドを除去することのできるコンパイラの方式を提案する.

このコンパイラは, コンパイルの初期に, まず解析専用のコードを生成し, このコードを実行する. この実行時の情報を記録し, コンパイラはこの情報を回収して, 最適化を行う. 本研究では特定の物理計算を対象とはしていないので, 汎用の計算を目的にした計算機を対象として, コンパイラの実装を行うことにした. 具体的には, 汎用性の高さと通信機能の豊富さから CP-PACS と Pilot-3 をターゲットマシンにすることにした. このコンパイラが受け付けるソースプログラムは, Fortran77 と何種類かの HPF ディレクティブで書かれたプログラムとし, 受理するディレクティブは `INDEPENDENT` , `PROCESSORS` , `DISTRIBUTE` , `BLOCK` と独自のディレクティブ `INSPECTONCE` とした. 出力としては, ターゲットマシンの通信ハードウェアに合わせて最適化した Fortran77 プログラムを生成することにした.

本手法の効果を確認するために, 実際に 2 種類のコンパイラを実装して確認した. どちらも基本的に同じ手法を用いており, 具体的に用いた最適化の手法もほぼ同じものである. 大きな違いはコンパイラが受理できるソースプログラムの性質の違いとコンパイラが動作する環境である. これら二つのコンパイラのうち, 1 つは PC 上で動作するコンパイラであり, もう 1 つは, PC クラスタ上で動作するコンパイラである. 1 台の PC 上で動作するコンパイラは簡便性が高いが制限が厳しいが, PC クラスタ版は汎用性が高いという違いがある.

また, このコンパイラの効果をいくつかのベンチマークで測定した. 用いたベンチマークプログラムは, genesis distributed benchmarks[37] から pde1, NAS parallel benchmarks[38] から FT と BT である. 本研究ではこの HPF 版 [39] を使った. そして, 人間が最適化したコード, 実行時の情報を用いて, 実行時にランタイムで動作を調整して最適化を行なう手法と比較した. シミュレーションは記述の容易さから, MPI や PVM のような広く知られた一般的な通信ライブラリで記述されることが多く, 最適化に用いたようなハードウェアを直接制御するような通信ライブラリは利用されないので, 一般的な通信ライブラリで人間の手で最適化したコードとの比較が必要であると考えた. また, 本手法の特徴である, 実行時の情報をコード生成に反映させることでどの程度の効果があるのか, 比較するためにランタイムで動作を調整して最適化を行なう手法と比較した.

本論文では, 以下, 2 章では関連研究を説明する. 3 章では提案する手法と目標を示し, 4 章では実際に実装したコンパイラの詳細と内部の処理に関して説明する. 5 章では実装したコンパイラをベンチマークを用いて測定し, 6 章でまとめる.

第2章　並列処理の高速化に関する従来の手法と問題点

並列計算の高速化をソフトウェアから行う研究は, コンパイラが静的にソースコードを調べて行う最適化の研究もあるが, 近年, 実行時の情報を用いて行う最適化の研究が盛んである. 実行時の情報を用いて行う最適化は, 実際にプログラムを実行して実行時の情報を回収し, プログラムを最適化する方法である. 静的に最適化する方法は, コンパイラがソースプログラムを実行する前に調べて, 実行した場合に無駄になるであろう部分を探し出し, それを最適化することによって行われる. このためにコンパイラは, コンパイル時にプログラムが実行されたときの状況をソースプログラムから推測しなければならなく, 高度な技術が必要になる. また, ソースプログラムの質, 調べたい情報によっては, 推測ができない可能性がある. これに対して, 実行時に情報を直接参照することができれば, これらの情報を回収することは容易である. この章では, 実行時情報を用いた従来の手法, 言語, ライブラリなどを論じ問題点を論じる.

2.1　実行時の情報を用いた最適化の問題点

実行時の情報を用いて最適化を行う研究にはさまざまな種類がある. これらの研究には並列処理を対象としていないものもあるが, 近年は並列処理を対象にする研究が多い. 並列処理に研究が多い理由には最適化の対象が多彩であるという点がある. 並列処理は逐次処理にない制御がある. 例えば, 通信, 同期, ノード間のキャッシングやノードへのデータのマッピングは逐次処理には必要ない.

本研究では実行時の情報を用いて最適化を行う研究を, コードを書き換えないものと書き換えるものに分類する. なぜなら, この二つは本質的に大きな違いがあるからである. どちらの方式も実行時の情報でプログラムの動作を変更しなければならない. コードは自分自身の動作を変更するために二種類の手段がある. 一つ目は自分自身を書き換える手法である. 二つ目は書き換えないで動作を決定するパラメータだけを変更する手法である.

実行時の情報を用いる手法はオーバーヘッドを生じる. これまでの実行時の情報を用いた最適化の研究では, これを除去することは困難である. このオーバーヘッドは実行中に自分自身の動作を最適化をするために埋め込まれたコードによって発生する. 自分自身のコードを書き換える手法では, このコードは自分自身の動作を観測して必要に応じて自分自身を書き換える. ランタイムでパラメータの調整をする手法では, このコードはパラメータを参照し書き換えを行う.

2.1.1　ランタイムによる最適化

パラメータの調整

動的な解析結果をプログラムの挙動に反映させる一般的な方法は, パラメータの調整である. この手法は, 実行時に動作を調整したい箇所にあらかじめ制御用の変数を埋め込んでおき, 実行中にその変数の値（パラメータ）を変えることによって, プログラムの挙動を制御する. この方法では, 調整されるパラメータは数値なので, 最適化の影響が数値から受けるものに用いられやすい. 例えば, ループのアンローリング, マイニング, タイリングの段数を変えて, プログラムの動作を微調整する. この変更により, プログラムはキャッシュのヒットなどが良くなるような動作に調整される. キャッシュのヒットを改善するパラメータを求めるためには, 実行環境であるハードウェアに依存した情報が必要になるので, 静的な解析だけでは行ないにくい.

この手法では, 動作を調整したい箇所に制御用の変数を埋め込むので, この変数を参照するオーバヘッドが残る. このようなオーバーヘッドを除去するためには実行時の情報を得てからコードを生成しなければならないので, 除去は困難である. しかし, 通常, 実行する前にコンパイルは終了しているので, 実行時の情報をコンパイル時に用いることは難しい.

マイグレーション

マイグレーションは実行時にデータを所有するプロセッサを変更する手法である. データが実行中に, 最も通信量が少なくなるように, プロセッサを移動する. プログラマやコンパイラが, データを配置される最的なプロセッサを判断できない場合や, 実行中にデータを配置される最適なプロセッサが変わる場合に有効である. この場合プログラムは, データが配置される最適なプロセッサを実行時の情報で判断して, 再配置する.

この手法でも 2.1.1 節で説明した手法と同様にオーバーヘッドが残る．マイグレーションを用いた最適化では, 実行時に通信の発生する量などを記録し, 必要に応じてデータを移動させなければならない．また, 本研究では HPF のディレクティブを採用している．HPF はデータの分割をプログラマが指示し, 計算をどのように分割してどのプロセッサが分割されたどの計算を行うのかをコンパイラが決定する．それに対して, マイグレーションは計算を行うプロセッサが決定されていて, データを再配置する手法なので, HPF で記述されたプログラムには向かない．

インスペクタ-エグゼキュータ

インスペクタ-エグゼキュータ方式はループの並列化に用いられる手法である．分散メモリ機でプログラムを並列に実行する際に, 必要になる通信を求めるために用いられる．この手法は, 配列変数の要素とループの繰り返しがどのようにプロセッサに分散配置されるかということが決まっている状況で用いられる．インスペクタ-エグゼキュータ方式は, プロセッサ間に分散配置されたデータの依存関係を実行時に解析し, それを元に必要になる通信を求める．ただし, インスペクタ-エグゼキュータ方式で並列化できるループは, ループのくり返しの間に依存関係がないループである．このような依存を持ったループにインスペクタ-エグゼキュータを用いる研究もあるが, オーバーヘッドが極めて大きい [15][16].

インスペクタ-エグゼキュータ方式では, プログラムは実行中に解析を行ない, 解析結果をテーブルに保存し, それを参照しながら通信を行なう．インスペクタ-エグゼキュータ方式を含む動的な手法の多くは, 実行時に解析をして実行時に解析結果を利用するので解析結果を元に最適なコードを生成することが難しい.

インスペクタ-エグゼキュータ方式の処理の流れは図 2.1 のように行われる．通常のインスペクタ-エグゼキュータでは, 並列化の対象になるループは実行時に二段階の処理で実行される．最初の処理はインスペクタと呼ばれ, 次の処理はエグゼキュータと呼ばれる．インスペクタは元のループと同じ形のループ (図 2.2) をプロセッサに分けて回る．この時, 分散配置される配列変数にいつ (ループの制御変数の値) どこに (要素) アクセスがあったか記録する．この時, 目的の計算は行なわず, 必要になるプロセッサ間の通信は行なわれない.

インスペクタの処理が終った後, 各プロセッサはリモートにある必要なデータが何であるかという情報を交換する．この処理を行なう直前に各プロセッサが持っている情報は, 自分が, どのプロセッサが持っているどのデータが必要か (または渡さなければならないか) という情報である．しかし, 実際に通信を行なうために

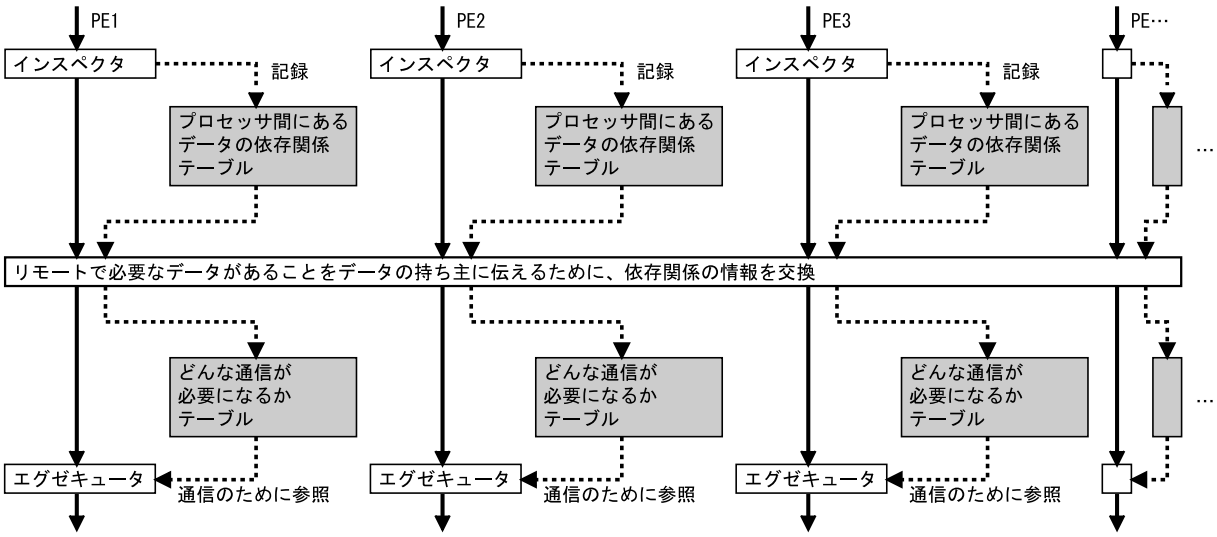


図 2.1: インスペクタ-エグゼキュータ方式を用いた並列処理

は, 受信に対応する送信命令を適切なプロセッサが適切なタイミングで行なわなければならないからである.

エグゼキュータは元のループと同じ形のループ (図 2.2) をプロセッサに分けて回る. この時, インスペクタによって得られたプロセッサ間のデータ依存に従って通信を行なう. また, 通信によって得られた情報をもとに目的の計算を行なう.

一般的にインスペクタ-エグゼキュータ方式は並列化のために用いられ, 最適化ではあまり使われない. しかし, インスペクタ-エグゼキュータ方式のターゲットはループと分散処理される配列変数であるので, 計算物理のシミュレーションプログラムを最適化に利用することに向いている.

計算物理のシミュレーションでインスペクタ-エグゼキュータ方式が効果的に動作するためには, インスペクタの解析結果を再利用できることが必要である. 図 2.3 のように, インスペクタの解析結果を利用できれば, 実線で示した流れのようにインスペクタ処理を何回も行なわないで済む. インスペクタの解析結果が再利用できなければ, 何度もインスペクタ処理を行なわなければならないのでパフォーマンスが大きく低下する. この判断のためにはプログラマからの指示などが必要になる. また, エグゼキュータは 2.1.1 節で説明した手法と同様にオーバーヘッドが残る. エグゼキュータはインスペクタが生成したデータの依存の情報にアクセスして, データを送らなければならないプロセッサと配列要素を決定しなければならない. この処理はインスペクタ処理後に残り, 実行時の効率を減らしている.

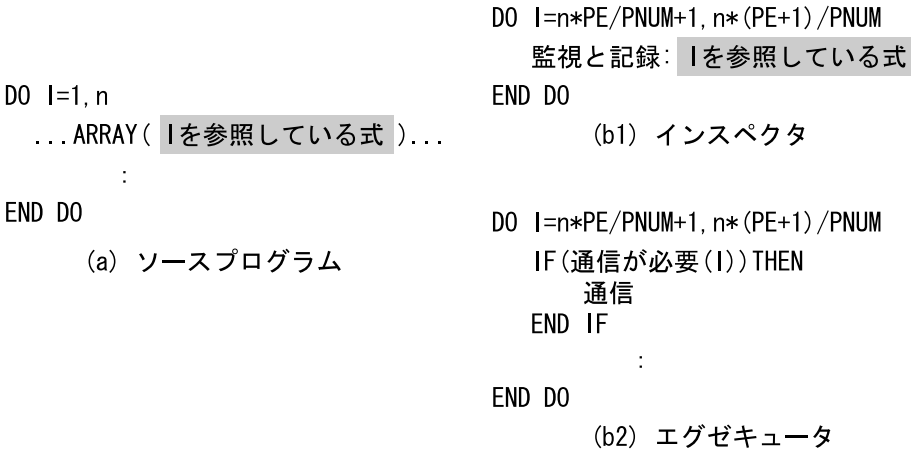


図 2.2: インスペクタとエグゼキュータのコード

実行時の情報を用いた研究

コードを書き換えずにランタイムで動作を調整する最適化の研究には, 次のようなものがある. Voss ら [11] の研究は実行時の情報でループを tiling や serializing し, そのパラメータを調整する. Diniz ら [12] の研究は同期の最適化を行う. Wu[10] らの研究は通信コストを減らすために, 小さいメッセージをまとめてひとつのメッセージにする. Viswanathan[9] の研究はデータのマイグレーションとレイテンシの隠蔽を取り扱う. Das ら [1][2] の研究はデータのマイグレーションとプロセッサ間のデータキャッシングを取り扱う. Ding ら [3] の研究はデータのマイグレーションとプロセッサ間のデータキャッシングを取り扱う. Fink ら [14] らは実行時の情報を用いたマイグレーションとデータの分割の研究をしている. Mitchel ら [18] は実行時の情報を用いてループの tiling を最適化する研究を行っている. Tomko ら [17] は実行時の情報を用いてデータの分割やレイテンシの隠蔽をする研究を行っている.

図 2.4 はループの tiling である. ループの tiling は多重ループの各ループを繰り返しの小さな複数の多重ループに分けて, その小さな複数の多重ループをループで繰り返して全部実行する手法である. この変換をすると, ループ自体のオーバーヘッドは増加するが, キャッシュのヒットが改善する場合がある.

また, 実行時の情報を用いてパラメータを調整し, 並列化ではなく最適化を行う研究もある. 最適化と同様に, 実行時にしか十分な情報を得られない場合, 実行時の情報を用いて並列化を行う. このような研究は不規則なデータアクセスをもたらすループを対象にしている場合が多い. 例えば代表的なものにはインスペクタ-エグゼキュータ方式 [4](2.1.1 節) がある. Lu ら [25] や Cox ら [22] はインスペク

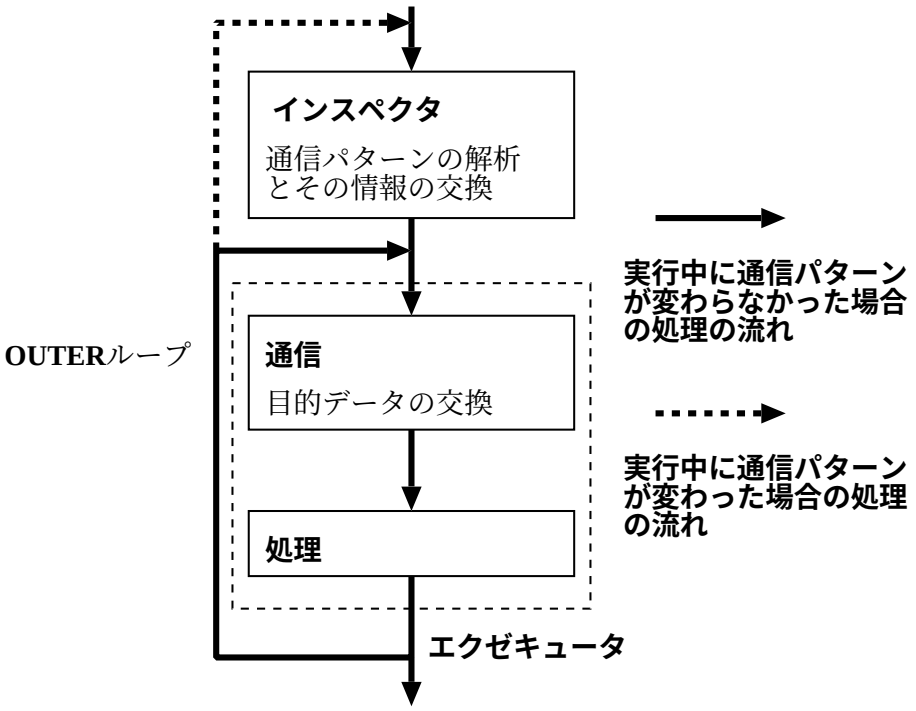


図 2.3: インスペクタの結果の再利用

タ-エグゼキュータ手法を用いたソフトウェア共有メモリのバックエンドシステムの研究を行っている. Benkner ら [23] はインスペクタ-エグゼキュータ手法を用いて, 複雑なループネストと依存関係を持った多重ループの並列化を研究している. Sussman[21] はインスペクタ-エグゼキュータ手法を用いて, スケジューリングを行う研究を行っている. Rauchwerger ら [26] や Zhang ら [27][28] は依存を持ったループを投機的に実行し, 実行時の情報を用いて並列化する研究を行っている. Chen ら [24] の研究は実行時の情報を用いたループの並列化であるが, 並列化を doacross で行う. Huelsbergen ら [19][20] は実行時の情報を用いて, 副作用のある関数の粗粒度並列化の研究を行っている.

これらの研究は実行時の情報を用いる手法に発生する固有のオーバーヘッドを除くことに着目していない. 計算物理のシミュレーションは実行時間が極めて長く, 特定の部分を繰り返す性質がある (3.4) ので, このようなオーバーヘッドを除く最適化が必要である.

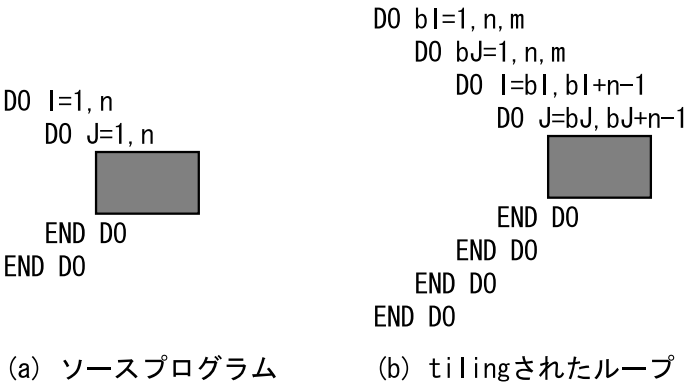


図 2.4: ループ tiling

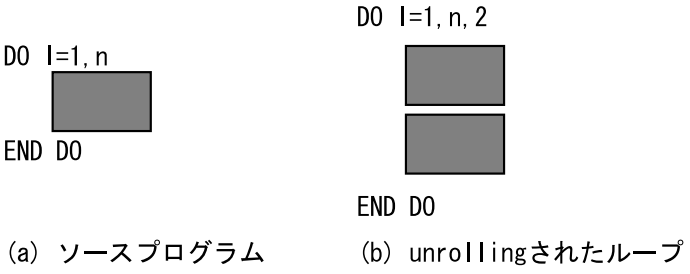


図 2.5: ループ unrolling

2.1.2　コード書き換えによる最適化

コードを書き換えて行う最適化の研究には, 以下のようなものがある. Philipssen ら [5] の研究は通信が最小になるようにオブジェクトを再配置する. 山崎ら [30] は実行時の情報を用いて, ループの tiling, unrolling を行う研究をしている. この研究は Java を対象にしており, バイトコードを直接編集することで, コードを書き換える.

図 2.5 はループの unrolling である. ループの unrolling はループのボディを複製し, 複製した数だけ反復回数を除算する手法である. これによって, 計算をそのままでループの反復数を減らすことができる. この変換をすると, ループ自体のオーバーヘッドを減らすことができる. また, パイプライン処理を改善する可能性がある.

実行時にコードを書き換えるためには大掛かりな仕掛けが必要になり, 書き換え自体が大きなコストになる可能性がある. コードの書き換えは, 実行時の情報を定

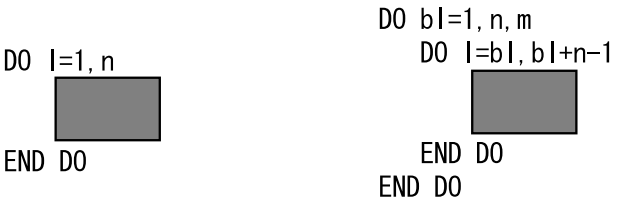


図 2.6: ループ mining

数伝播することができるので, より強力な最適化をすることができるがオーバーヘッドが大きい. したがって, このコストを実行時から除くことは必要である.

2.1.3　サポートツール

また, 最適化のための解析に特化したツールも研究されている. TEA EXPERT[29] はループの mining や unrolling の最適な段数を求めるためのサポートツールである. Ponnusamy ら [6][8] は最適なデータの分割, 通信のスケジューリング, プロセッサ間のデータキャッシングをコンパイラに伝える研究を行っている. Y. S. Hwang ら [13] は間接参照のある配列参照のある不規則ループを実行時の情報をもとに最適化するための情報をコンパイラに伝えるサポートツールに関して研究している. また ATLAS[7] は並列化ではないが実行時の情報を用いたループの最適化の研究が行なわれている.

図 2.6 はループの mining である. ループの mining はループを繰り返しの小さな二重ループに分けて, 少ない回数の繰り返しを持つループをループで繰り返して, 元のループの繰り返しを実行する手法である. この変換をすると, ループ自体のオーバーヘッドは増加するが, 生成されるコードの演算の単位を調整することができる. また, 依存を持ったループを並列化することができる場合がある.

これらの研究は実行時の情報を解析するために, さまざまな監視用のコードを実行コードに埋め込む. 従って, 実際に目的の計算を行っている間, そのオーバーヘッドは発生しつづける. このオーバーヘッドは多重ループの最内部などでは大きくなる可能性がある. また, 本研究がターゲットにするシミュレーションプログラムは最適化したい個所が多重ループの最内部になることが多いので (3.4), このオーバーヘッドは除去される必要がある.

2.2　並列プログラミングで使われる言語とライブラリの問題点

2.2.1　HPF

HPF[31] は計算物理の記述に向いたユーザインターフェースを持つプログラミング言語である. HPF (High Performance Fortran) はプログラマが容易に記述できて, 並列環境をよく抽象化して可搬性がよい. 計算物理のプログラマは計算機の専門家でないので, 記述が容易な事は重要である. しかし, HPF によって記述されたプログラムは実行速度の点で計算物理に向かない. なぜなら, コードが効果的に実行されるように, プログラマがハードウェアの特徴にあわせた記述をすることが難しいからである. また, 言語の仕様が抽象的であるため, コンパイラが抽象的な記述をされたプログラムと現実のハードウェアの間でコードを変換しなければならないので, 効果的なコードを生成する商用コンパイラが少ない.

HPF を用いた並列プログラミングが容易な理由は, 抽象度が高く, プログラマは明示的な通信命令を記述する必要がないからである. プログラマが手動で通信命令を記述しなくて済むことが長所になるのは, 以下の理由からである. プログラマが手動で通信命令を記述するためには, 実装しているシミュレーションでどのような通信が必要になるのか把握する必要がある. また, プログラマが手動で通信命令を記述しなければいけないとすると, 効果的なコードを生成するためには, プログラマはシミュレーションが必要とする通信だけでなくハードウェアの知識, 最適化の知識が必要になる.

HPF を用いた並列プログラミングが可搬性に優れている理由は, 抽象度が高く, ハードウェアを隠蔽しているからである. HPF の記述は特定のハードウェアに基づかず, プログラマは一般的な並列処理に関する情報を記述する. コンパイラはその情報をハードウェアにあわせた最適化に利用する.

HPF は通常の Fortran コンパイラにプログラマが並列化のヒントを与えられるようにした言語である. ヒントはディレクティブと呼ばれる. このディレクティブは$ HPF!で始まるコメント行で与えられる. プログラマは, このディレクティブをプログラム中に記述することで, 並列処理に関する情報を記述する. このディレクティブは, 特定のアーキテクチャを意識して設計されておらず, 抽象的である. これにより, HPF は可搬性がよく, プログラマにアーキテクチャの知識を要求しない. プログラマが記述するディレクティブは, プロセッサの個数 (PROCESSORS 等), メモリの分散 (DISTRIBUTE , BLOCK , ALIGN 等) , プログラマがプログラムの挙動を保証するもの (INDEPENDENT , NEW , REDUCTION 等) 等がある.

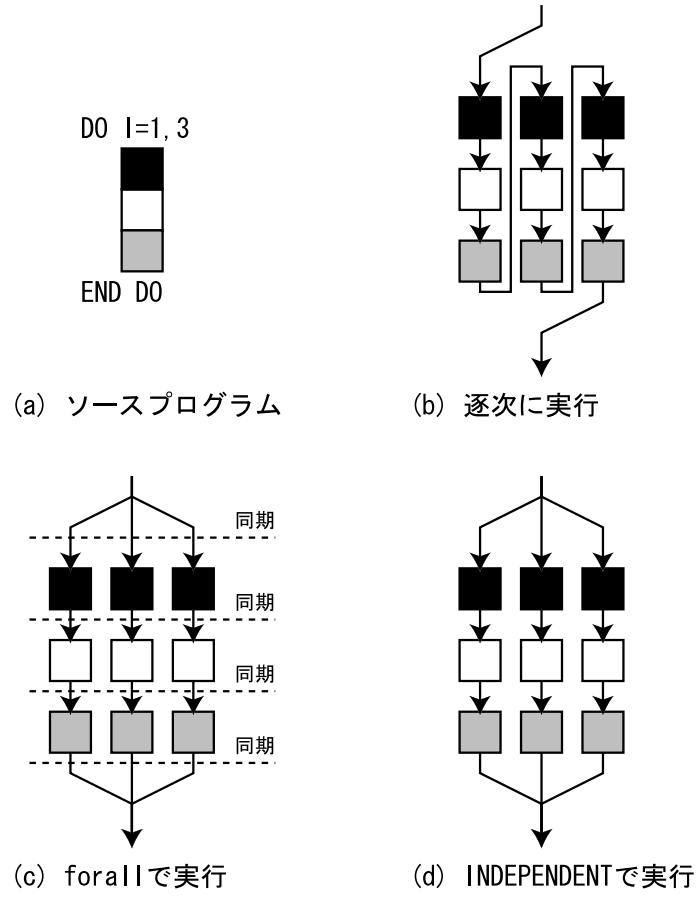


図 2.7: 並列化されたループの実行順序

しかし, シミュレーションが実行時間が莫大である場合, 効果的なコードを生成することが重要であるにもかかわらず, HPF で記述した場合, 効果的なコードを生成しずらい場合がある. この理由は, コンパイラが安全のため効果的でないコードを生成しがちな点と, プログラマが細かい制御をしずらい点による.

もし, プログラマが与えたディレクティブが十分でない場合, コンパイラが安全のため効果的でないコードを生成しがちである. HPF の命令をどのくらい詳細に記述するかという点は, プログラマに任されていて, 生成されるコードの実行効率はこのディレクティブ量に左右される. この理由は, ディレクティブがない部分はコンパイラが推測しなければならなくなるからである. 例えば, ループに INDEPENDENT のディレクティブが与えられていれば, コンパイラはこのループをループが運ぶ依存がないものとして並列化することができる. このディレクティブはループの繰り返しの実行順序が結果に影響を及ぼさないことをプログラマが保証することを示

している (図 2.7). しかし, もしこのディレクティブがなければ, コンパイラは並列化のために配列変数の添え字やリダクション変数を調べて, ループが運ぶ依存を解析しなければならない. この解析がうまくいかなければ, コンパイラはこのループを並列化することをあきらめるかもしれない.

HPF は抽象度が高いので, プログラマがハードウェアに関して細かい制御をしずらい. これは, HPF は容易な記述と可搬性のため, ハードウェアを隠蔽しているからである. このため HPF は, プログラマがハードウェアを意識して効果的なコードを記述しずらい. 例えば, CP-PACS/Pilot3 がサポートする TCW (Transfer Control Word) の再利用を記述することができない. この機能は TCW を再利用して, 通信の設定にかかる時間を減らすことができる. HPF コンパイラがこの機能をサポートするためには, HPF を拡張してどのような通信が行われるのかコンパイラに伝えるディレクティブの種類を増やすか, より強力な解析を行う必要がある.

HPF では, プログラマが, より詳細なディレクティブをコンパイラに与えることができるようにするために, ディレクティブが追加されてきた. 例えば, `SHADOW` は通信で扱うデータをキャッシュするかどうかコンパイラに伝えるディレクティブである. また, `ASYNCID` は通信を非同期に行い, 演算とオーバーラップできることをコンパイラに伝えるディレクティブである.

これらの拡張は HPF の抽象度を選択的に下げる. これらの拡張されたディレクティブを与えることで, コンパイラにより詳細な情報を与えることができるが, プログラマはより高度な知識が必要になる. また, HPF はハードウェアを隠蔽しているので, 特殊なハードウェアをサポートすることは難しい. しかし, この問題を解決するためにハードウェアの隠蔽をあきらめて, ハードウェアごとにディレクティブを追加していくことは現実的ではない. なぜなら, ディレクティブの種類が不用意に増え, 可搬性も悪くなるからである. また, 公認の拡張仕様でも, コンパイラによってサポートされていないものがある [32]. このようにサポートされていないディレクティブを使った場合, 記述したディレクティブが無駄になり, コンパイラは効果的なコードを生成できない可能性がある.

新しいディレクティブを追加せずにハードウェアを駆使した効果的なコードを生成したり, プログラマにハードウェアの詳しい知識なしで効果的なコードを生成するためには, より強力な解析が必要になる. 計算物理のシミュレーションでは, 容易な記述でより効果的なコードが望まれているので, この改良は重要である.

2.2.2　MPI

MPI (Message-Passing Interface)[33] はメッセージ通信のための標準的なインターフェースである. Fortran や C 言語から利用でき, シンプルで使いやすく, 可搬性に優れている. プログラマは通信命令を記述しなければならないが, ハードウェアの特性はライブラリの中に隠蔽されているので, プログラマはターゲットマシンの特性を意識したプログラミングをする必要がない. MPI は, 単一のアーキテクチャのノードから構成されたハードウェア上で, SPMD スタイルや MIMD の環境で用いられる.

HPF と異なり, MPI を用いたプログラミングでは, プログラマは明示的に通信命令を記述しなければならない. このため, プログラマはプロセッサ間で分割したデータ間の依存関係を把握しなければならない. プログラマがデータの依存関係を把握し, プログラマが通信を記述しなければならないため, MPI は HPF よりプログラマに負担をかけることになる. しかし, HPF と異なりコンパイラは並列化コンパイラでなく, 通常のコンパイラにライブラリを加えた形態であるので, コンパイラは通信命令を挿入する必要がない. このため, MPI を用いたプログラミングでは, 通信をプログラマが管理する分, HPF より効果的なコードを得やすい.

MPI を用いた通信の手順は次のようになる.

```
CALL MPI_INIT(STAT)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, PID, STAT)
送信側{
CALL MPI_SEND(ARRAY(1), ELM_NUM, TYPE, PID, TAG, MPI_COMM_WORLD, STAT)
}
受信側{
CALL MPI_RECV(ARRAY(1), ELM_NUM, TYPE, PID, TAG, MPI_COMM_WORLD, STAT, STAT)
}
CALL MPI_FINALIZE(STAT)
```

まず, MPI_INIT でライブラリを初期化する. 次に MPI_COMM_RANK でランク (0 起源の自分自身のプロセスまたはプロセッサ番号) を求める. MPI_COMM_WORLD はマルチキャスト通信を行う際のグループを示している. 次に必要に応じて, プロセッサ間で MPI_SEND と MPI_RECV でデータを交換する. TAG はメッセージの識別子である. 最後に MPI_FINALIZE で通信ハードウェアの解放などを行う.

MPI のライブラリはハードウェアの特性を隠蔽している. このため, プログラマは分散処理しているデータ間の依存を把握し, 通信命令を記述しなければならない

が, ターゲットマシンのハードウェアの特性を把握する必要はない. そのかわり, MPI のライブラリは内部で, ハードウェアの持っている特性を利用して, 効果的な通信を行おうとする.

しかし, 利用したい特性を MPI のインターフェースで隠蔽すると, その特性をうまく利用できない場合がある. 例えば, CP-PACS/Pilot3 がサポートする TCW (Transfer Control Word) の再利用を記述することができない. この機能を利用できるようにするには, 同じ通信 (転送元と転送先のアドレス, サイズ, 転送元と転送先のプロセッサ) が繰り返される個所をプログラマが記述できるように MPI のインターフェースを修正しなければならない. このため, ハードウェアの特性にあわせた最適化を行って, 少しでも実行時間を短縮する要求が強い場合には向かない.

2.2.3　PVM

PVM (Parallel Virtual Machine)[36] はメッセージ通信のための標準的なインターフェースである. Fortran や C 言語から利用でき, シンプルで使いやすく, 可搬性に優れている. MPI と同様に, PVM を用いたプログラミングでは, プログラマは明示的に通信命令を記述しなければならない. MPI と大きく異なる点は, 異なるアーキテクチャのマシンの集合を単一の並列計算機として利用することを可能にしている点である.

PVM は異なるアーキテクチャのマシンを協調動作させることを可能にしているので, 典型的な並列計処理と異なる点がある. PVM ではプロセッサ番号の扱い, プログラムの起動, 通信の形式などが大きく異なる. まず, プロセッサ番号の扱いに関して説明する. PVM では, あらかじめネットワークで構成されている複数のマシンから必要な台数だけ利用して問題を解く. このため, ネットワークを構成しているマシンの中で自分自身を区別する識別子が必要になる. またそれとは別に, 問題を解くために利用しているマシンを識別する 0 起源の通し番号がないと, プログラミング上不便である. PVM ではプロセッサを区別する識別子に, これらの二種類があり, それぞれタスク識別子, インスタンス番号と呼ばれる. 次に, プログラムの起動に関して説明する. PVM では SPMD でなくマスタスレーブ方式でプログラムを記述する. マスタスレーブ方式は親のプロセスが, 実行させたいコードを指定して子のプロセスを起動する. これにより, マスタスレーブ方式は各プロセッサごとに異なるコードを実行できる. このため, マスタスレーブ方式は SPMD よりも柔軟である. しかし, 典型的な並列計算機では, ノードのアーキテクチャが同一であるので, ノードごとに異なるコードを渡す必要性は低い. 最後に, 通信の形

式に関してである. PVM は異なるアーキテクチャのマシンを協調動作させるので, ノードごとにデータの保存の方式 (例えば, ビッグエンディアンやリトルエンディアン) が異なる可能性がある. このため, 通信でデータを送る前に, PVM で取り決められている標準的なデータフォーマットに変換してから通信を行う. この変換は送信命令の内部で行われる.

PVM を用いた通信の手順は次のようになる.

```
CALL PVMFJOINGROUP("GROUP NAME", INSTID)
CALL PVMFSPAWN("FILE NAME", PVMDEFAULT, "*", CHILD_NUM, STAT(1), STAT)
CALL PVMFGETTID("GROUP NAME", 1, TASKID)
送信側{
CALL PVMFINITSEND(PVMDEFAULT, STAT)
CALL PVMFPACK(TYPE, ARRAY(1), ELM_NUM, STRIDE, STAT)
CALL PVMFSEND(TASKID, TAG, STAT)
}
受信側{
CALL PVMFRECV(TASKID, TAG, STAT)
CALL PVMFUNPACK(TYPE, ARRAY(1), ELM_NUM, STRIDE, STAT)
}
CALL PVMFLVGROUP("GROUP NAME", STAT)
CALL PVMFEXIT(STAT)
```

まず PVMFJOINGROUP で, そのプログラムが動作しているプロセッサが参加するグループ名 (協調動作するコードの集合の識別子) を宣言する. この時, 自分自身のプロセッサのインスタンス番号 (プロセッサ番号に相当する) を求めることができる. 次に必要に応じて PVMFSPAWN で, 他のプロセッサが実行するコードを指定して, そのコードを実行させる. 次に必要に応じて, 通信でプロセッサ間でデータを交換する. 通信を行う際, 対象になるプロセッサはインスタンス番号でなく, タスク識別子で指定されなければならない. インスタンス番号から, タスク識別子への変換は PVMFGETTID で行う. 送信する側は PVMFINITSEND を呼び出して, 送信バッファを初期化する. この時, データフォーマット (そのまま送るのか, PVM で取り決められているフォーマットに変換して送るか等) を選択する. PVM では複数のデータの塊を一度に送信することができる. PVMFINITSEND を呼び出してから, 必要な回数 PVMFPACK を呼び出して送信したいデータを選択する. この後, PVMFSEND で実際にデータをネットワークに送り出す. この時の TAG はメッ

セージの識別子である. 受信する側はPVMFRECV を呼び出してデータを受信する. この後, 送信側がPVMFPACK でパックした順にPVMFUNPACK でデータを取り出す. TAG によって区別されている通信はPVMFPACK やPVMFUNPACK で混線することはない. 最後にPVMFLVGROUP でプロセッサは参加しているグループから抜け, PVMFEXIT で通信ハードウェアの解放などを行う.

PVM もMPI と同様に, ハードウェアを隠蔽するインターフェースを持っているので, 容易に記述できて可搬性に優れているが, ハードウェアの特性を利用することが難しい場合がある.

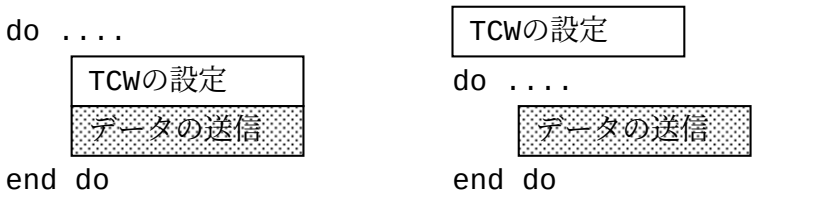
2.2.4　RDMA 用ライブラリ

CP-PACS/Pilot-3 には, 持つ独自の通信ハードウェアであるRDMA (Remote DMA)[44] を直接操作するライブラリを備えている. RDMA のライブラリのインターフェースはRDMA のハードウェアに特化して作られているので, 基本的にハードウェアが備えている全ての機能をプログラマに提供している. プログラマはRDMA の持つ機能を効果的に利用したい場合, このライブラリを用いるか機械語を直接記述しなければならない. CP-PACS/Pilot-3 はMPI やPVM の通信ライブラリを持ち, HPF の処理系を備えているが, これらを使った場合でも内部でRDMA を使用している. しかし, MPI, PVM のインターフェース越しにRDMA を利用した場合, インターフェースの差異を埋めるため, ピンポンのように単純な通信でもスループットに差が出る. このピーク時の性能差は大きい [41]. このため, RDMA 用ライブラリを用いてRDMA を直接利用することは, 特に少しでも実行時間を短くしたい場合に有効である.

RDMA ライブラリを通してRDMA がサポートする通信機構にはTCW の再利用, ブロックストライド通信, 片側通信の機能がある. ここではその三点について説明する.

ブロックストライド通信 (図 2.9) は等間隔不連続のデータを一回の送信命令で送ることができる (ブロック長:4〜1020bytes, ストライド長:4〜65532bytes). ブロックストライド通信は多次元の配列変数の端を転送する場合に効果的な場合がある. なぜなら, 2 次元以上の配列変数の端はアドレス空間上で不連続になるからである. 一般に通信は配列変数の端のデータをやり取りすることが多いので, 多くのシミュレーションではブロックストライド通信が適している通信が多発する.

TCW 再利用型通信はTCW (Transfer Control Word) を再利用することで行うオーバーヘッドの少ない通信である (図 2.8). TCW 再利用型通信は特定の通信 (送信, 受信のプロセッサ, アドレス) が何度も行われる場合, 有効である.



(a) TCWを再利用しない通信　(b) TCWを再利用する通信

図 2.8: TCW を再利用する通信

片側通信は, 受信側で明示的に受信命令を実行しないでも行える通信である. RDMA では RREAD と呼ばれる受信命令も存在しているが, この命令なしでも通信を行うことができる. RREAD は同期を取るために用いられる. RDMA で, あるノードから他のノードへ通信を行う場合, 送信側は受信側のメモリアドレスを指定して送信権を獲得しなければならない. 通信の際, 送信側は受信側のメモリへ直接データを書き込む. これによって無駄なメモリコピーが発生しない. また, 受信命令は実行することも実行しないこともできるので, 例えば, プログラマが同じディスクリプタ (メッセージを識別するために付けられるタグ) で RWRITE (送信命令) を二回実行して受信確認は RREAD で一度だけ行なうようなプログラムを記述することができる.

RDMA を用いた基本的な通信は以下のように行う.

```
*COPTION COMBUF . . . . ::ARRAY
STAT=$RPARAM(. . . . )
STAT=$RCREATE(. . . . )
STAT=$RRIGHT(PID, . . . . )
CALL $RSETOFF(OFFSET, . . . . )
CALL $RSETMSG(. . . . , ELM_NUM, ARRAY)
CALL $RWRITE(. . . . )
CALL $RREAD(. . . . )
```

RDMA で送信または受信するデータを通信バッファ上に配置するように, *COPTION 命令でコンパイラに伝える. $RPARAM で通信バッファに識別子をつける. $RCREATE で通信バッファ上に配置されたメモリを実際に通信バッファにする. $RRIGHT で送信したい相手から送信権をもらう. $RSETOFF で送信したいデータの通信バッファ上からのオフセットを求める. $RSETMSG で送信した

いデータの長さ, または$RSETSTR で送信したいデータのブロック長, ストライド長, 繰り返し数を指定する. $RWRITE か$RSTRID で実際にデータを送信する. 必要な場合, $RREAD でデータの受信確認をする. TCW を再利用したい場合は, $RWRITE の代わりに$RMKTCW, $RSTRID の代わりに$RMKSTCW で TCW を設定し, $RKICK で送信する.

プログラマが RDMA を直接利用してプログラムを記述することは効率的ではない. プログラマは広く知られている MPI や PVM のライブラリや HPF のような言語ではなく, 特定のハードウェアでしか使えないインターフェースを学習するのは効率的ではない上, HPF と異なり, プログラマが明示的に通信命令を記述しなければならない. RDMA のライブラリは特定のハードウェアに密着しているため, RDMA を直接利用した記述プログラムは可搬性は低い. また, RDMA のライブラリはハードウェアを直接反映しているので, 効果的な記述をするにはこのハードウェアの詳細な知識を必要とするので, プログラマは RDMA を直接利用したプログラムを記述しにくい.

例えば, TCW の再利用, ブロックストライド通信を効果的に使うためには, 分散処理されたプログラムで, どのような通信が必要になるか正確に把握する必要がある. TCW を効果的に再利用するためには, プログラマは同じ相手 (プロセッサ, アドレスなど) にくり返し通信が行われることを把握する必要があり, ブロックストライド通信を効果的に使うためには, プログラマは通信したいメモリの領域の形状を正確に把握する必要がある.

例えば, 片側通信を効果的に使ってメッセージの冗長な到着確認を除去するためには, 高度なプログラミングが必要になる. メッセージの冗長な到着確認を除去するために, プログラマは必要な個所のみ RREAD 命令を記述しなければならない. RREAD を利用するかどうかを決めるのはプログラマなので, RDMA のライブラリは同じディスクリプタ (通信の識別子) の通信に関して, どの RWRITE とどの RREAD が対応するのか知ることができない. このため, RDMA は実行された RWRITE の回数だけ RREAD が行なわれないことを前提に実装されている. 具体的には, RDMA では一つのディスクリプタに関して RREAD されていないメッセージの個数は保持せず, 同じデスクリプタを用いてくり返し通信を行なう場合, 到着確認が上書きされるようになっている. このため, プログラマが RDMA で並列プログラムを記述する際, 同期を慎重に記述しなければならない. ループで繰り返し同じアドレスから始まるデータを繰り返し送信する場合, ダブルバッファやバリア同期などを使ってプログラマが期待している RWRITE と RREAD の対応がずれないようにしなければならない.

表 2.1: 実行時の情報を用いた最適化の研究の分類

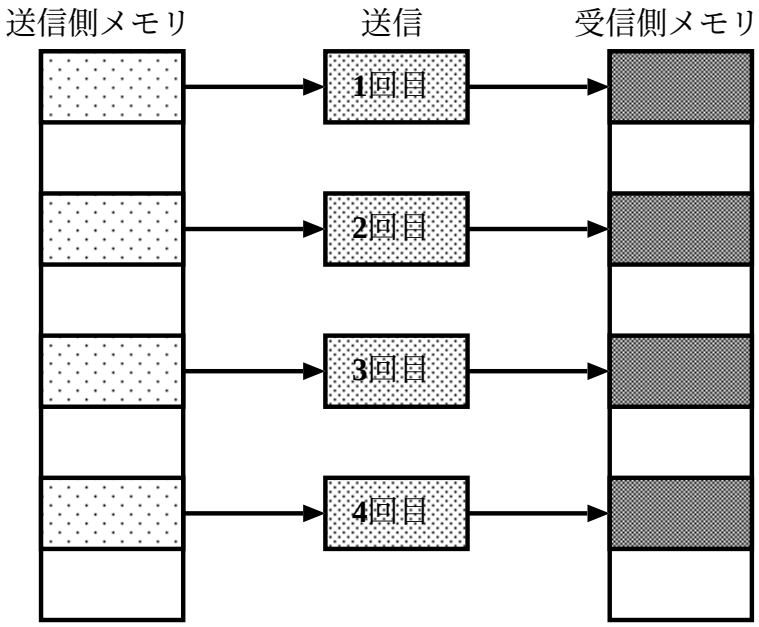
	通信	通信以外
ランタイムで動作を調整	J. Wu ら C. Ding ら R. Das ら G. Viswanathan ら	M. Voss ら P. Diniz ら
コンパイラで動作を 調整されたコードを生成	本研究	プロファイラ

2.3　まとめ

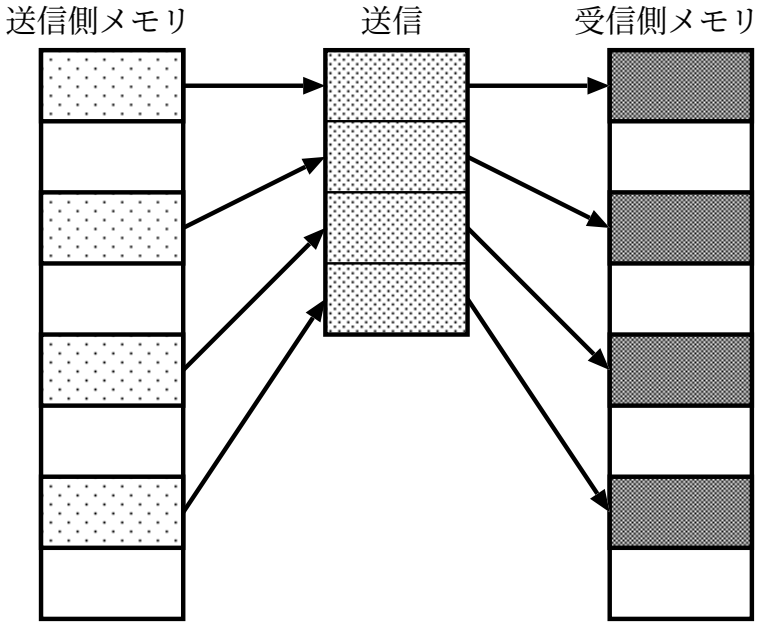
本章では, 実行時情報を用いた従来の手法, 言語, ライブラリなどの問題点を論じた. 最適化にはプログラムの解析が実行時の情報を用いた最適化には, ランタイムのパラメータを調整する方法と, コードを再生成する方法等がある. また, 解析専用のサポートツールの研究も行われている. これらの手法はオーバーヘッドがあり, このオーバーヘッドを除去することは難しい. これらの手法はプログラムの実行時にプログラムの動きを監察し, 解析を行い, 状況に応じてプログラムの動作を調整するので, 観察, 解析, 調整のためにオーバーヘッドが生じる. 数々の実行時の情報を用いた最適化の研究は様々な最適化の対象を研究しているが, このようなオーバーヘッドを除く研究は行われていない.

また, 通信の最適化を扱って, コードを再生成する研究は行われていない. 表 2.1 は実行時の情報を用いた最適化の研究を分類したものである. 実行時の情報を用いた最適化の研究の多くは通信を対象にして, ランタイムで動作を最適化する. また, 実行時の情報をコンパイラから利用する技術はプロファイラなどがある. 本研究は実行時の情報をコンパイラから利用して, 通信を最適化することを目指している. コンパイラから実行時の情報を利用することで, ランタイムだけでは難しい最適化を行なう.

また本章では, 広く使われている並列処理のための言語とライブラリの問題点について論じ, CP-PACS/Pilot-3 の通信ライブラリの１つである RDMA 用のライブラリの問題点について論じた. 一般的な並列言語の HPF と一般的な通信ライブラリの MPI と PVM は記述が容易である反面, ターゲットの並列計算機が特殊な仕様のハードウェアを持っている場合, それに合わせた効果的なプログラミングができない. それに対して, RDMA 用のライブラリは CP-PACS/Pilot-3 の通信ハードウェアである RDMA を利用して効果的なプログラムを記述することができるが, プログラマに対して高度な技術や労力を要求する.



(a) 通常の通信機構による通信



(b) ブロックストライドの機能を持つ
通信機構による通信

図 2.9: ブロックストライド単位の通信

第3章　目標とするHPFコンパイラ

目的プログラムを実行することで最適化に必要な情報を集め, 実行時の情報を用いて最適化するにもかかわらず, 発生するオーバーヘッドを除去するコンパイラを提案する. 簡単な静的解析で情報が得られる箇所は静的に解析し, 静的な解析が難しそうな箇所は実行時の情報を使う. 実行時のデータを実行時にとることによって発生する実行時のオーバーヘッドを避けるために, ソースコードの一部分をコンパイル時に実行することで解析を行なう. このようにして, 動的な手法と静的な手法を使い分けることで, 静的な解析でやりにくい解析を処理しつつ, 動的な解析でありがちな実行時間と本手法で発生しがちなコンパイル時間の増加を押える.

また, 提案するコンパイラはコードを最適化する. 通常, 実行時の情報を使った最適化は解析結果を参照するためのオーバーヘッドが残る. 本研究の目的は強力な最適化であるので, このようなオーバーヘッドも除く. 目標のコンパイラは実行時の情報をコンパイル時に利用することで, 強力な最適化を実現する.

コンパイルに実行時の情報を用いた解析をする理由は, 解析力の強さと簡便性である. 本研究がターゲットにしているプログラムは計算物理のシミュレーションであるので, 実行時間が長い. また最適化の成果で回収できるのであれば, コンパイル時間が増加する問題は小さくなる.

コンパイラはHPFを入力として受け付け, ターゲットマシンに特化したコードを生成する (例えば 2.2.4 節の RDMA) . HPF を採用した理由は記述の簡便さである (2.2.1 節). 利用者が計算機の専門家でないことを考えると, 通信を明示的に記述する必要がない点からHPFが最適であると考えられる.

本章では, 目標とするコンパイラがどのような手法を用いて最適化を行なうのか, 結果としてどのようなコードを生成するのかを説明する.

3.1　計算物理に求められるコンパイラ

計算物理のためのコンパイラは, プログラムが実行される環境にあわせて最適化することが望ましい. このようなプログラムは実行時間の長さとかかる費用か

ら, 少しでも実行時間を短くしたので, ハードウェアが実行時間を短縮するために有効な機能を持っている場合これを活用したい. しかしながら, プログラマがそのようなハードウェアを効果的に利用するように最適化されたプログラムを記述するためには, プログラマは高度な知識や労力を必要とする. しかし, プログラマは計算機の専門家ではないので, プログラムは容易に記述できなければならない. このため, 入力として受け付ける言語の仕様は, ハードウェアを隠蔽していないといけない.

計算物理のためのコンパイラは, 入力される言語はシンプルであるが強力な最適化を行わなければならないので, 強力な解析が必要になる. しかし, 計算物理のシミュレーションは莫大な実行時間を要するので, 少しでも強力な最適化を行うために, 解析時間が多少長くなってもかまわない. この場合, 実行時間の短縮で解析時間の増加を回収をすることが必要である. またコンパイル時間が増加した場合でも, コンパイル環境がターゲットマシンでないのであれば, 費用的な問題は低くなる.

3.1.1　通信の最適化

計算物理のシミュレーションは分散メモリのアーキテクチャを持った並列計算機で行われることが多いので, 超並列計算では通信の最適化が重要である. 分散メモリ機での負荷分散は通信がボトルネックになりがちである. このため, 通信による負荷を減らすために様々な研究が行われてきた (2 章).

本研究のコンパイラの最適化の主な対象は通信である. 詳細には以下のような最適化を行った. 通信量が最小になるようにループの繰り返しをプロセッサに分配する. HPF では, プログラマがデータをプロセッサに分割する指定をする. コンパイラはその指定に基づいて, 通信量が最小になるようにする. メッセージを融合してメッセージの数を減らし, 通信に発生するレイテンシやハードウェアの初期化にかかる時間を減らす. これにより通信で扱うデータ量が変わらなくても, 結果的に通信にかかる時間は短くなる. 実行時の解析を用いた最適化に発生するテーブル参照にかかる時間を減らす. 実行時の解析を用いた最適化はパラメータを調整することで, 自分自身のプログラムの挙動を調整する. このため, プログラムの各部からそのパラメータを参照することになる. また, 可能であれば TCW (Transfer Control Word) を再利用する通信を利用する. TCW とはデータの転送を制御するための情報を保持したテーブルである. TCW を再利用すれば, このテーブルを生成するのに必要なコストを削減できる.

3.1.2　容易なインターフェース

本研究では, コンパイラが受理する言語に HPF を選んだ. 計算物理のプログラマは容易に記述できる言語でプログラミングできることが望ましい. プログラマは計算機の非専門家であるので, 容易に記述でき, かつ広く知られた言語であることが望ましいので, HPF が適している. HPF を用いればプログラマは通信を意識せず分散プログラミングをすることができる.

3.2　実行時の情報を用いた手法

本研究では,実行時に行われる最適化や並列化の中で, インスペクタ-エグゼキュータを用いる. この節では, 並列処理に用いられることが多い動的な手法の例として, パラメータの調整, マイグレーション, インスペクタ-エグゼキュータの比較を行ない, その後インスペクタ-エグゼキュータに基づいた理由を説明する.

本研究のコンパイラはインスペクタ-エグゼキュータ手法 [4] を用いて, 実行時の情報を解析する. 本研究のコンパイラの最適化は次のような特徴がある. コンパイラはインスペクタの結果を用いてエグゼキュータを最適化する. この最適化はエグゼキュータのコードに静的に行なわれるので, 通常のインスペクタ-エグゼキュータより強力な最適化を行なうことができる. コンパイラはコンパイル中にインスペクタの情報を利用したいので, インスペクタをコンパイル中に実行する. このため, コンパイラはインスペクタとエグゼキュータを分割して生成する. コンパイラはインスペクタの実行後, インスペクタが生成したプロセッサ間のデータの依存情報を集める. この後, コンパイラはこの情報を使って, プロセッサ間の通信が最適化されたエグゼキュータを生成する. この最適化は, コンパイル時の定数の畳み込みを含んでいる. これにより, 実行時の情報を用いることによって発生するオーバーヘッドを除去する. コンパイラが最終的に生成するコードは, インスペクタによって得られた情報で静的に最適化されたエグゼキュータである. ユーザはこのエグゼキュータを実行して, シミュレーションを行なう.

3.3　問題点と解決法

3.3.1　問題点

通常, 実行時に得られた情報を用いてコードを最適化することは難しい. なぜなら, 通常のコンパイラでのコード生成と実行のながれは順序があり, コードを生成

するのはコンパイル時であり, 実行時の情報はコードが生成された後に実行されてはじめて得られるからである. このため, 実行時の情報を用いてコードを最適化するためには, 実行した後にコードを最適化しなければならない.

実行時の情報を用いてコードを最適化する代わりに, ランタイムを用いて最適化する方法がある (2.1.1 節) . この方法はランタイムの情報で最適化されたコードを生成する代わりに, パラメータを調整してプログラムの挙動を調整するので, 適応できない最適化がある. 例えば実行時の情報の定数畳み込みを行なうことができない.

実行時の情報を用いてコードを最適化するために, 実行時にコードを書き換える手法がある. しかし, この手法は二つの点からさまざまな工夫が必要になる. 一つは実行されるコードはバイナリであることと, もう一つはコードを書き換えるためにかかる時間的コストである. まず, 実行されるコードはバイナリであるため, ソースコードにある変数などのような抽象化された情報がない. プログラムのコードと実行時のデータはアドレスとバイト単位でしか表されていない. このため, どこを書き換えるとどのようにプログラムの動作が書き換わるのか, 書き換えのためのランタイムが保持したり, 適切に目印をつけたソースプログラムなどが必要になる. さらに, 実行を中断してコンパイルしたり, 実行の途中を保持してコードを変えたりしなければならない. 次に, 書き換えにかかる時間が大きく, 逆に実行時間が増加してしまう可能性がある点である. コードの書き換えのための呼び出しを, 適切な場所に組み込まなければ, コードは目的の計算ではなく書き換えのために莫大な時間を使ってしまう. これらの処理はプログラマからの支援なしでは難しく, 適切に指定するのは難解である.

3.3.2　解決法

本研究で提案する方式は, コンパイル時の 1 フェーズとして目的のプログラムの一部分を実行する. 提案したコンパイラは, 実際にコードを実行して実行時の情報を回収する処理をコンパイル時に移動することで, 実行時の情報をコンパイル時に利用できるようにする. このため, コンパイラは実行時の情報を得るためだけに実行するコードを生成し, それを実行してから改めて最適化されたコードを生成する (図 3.1) . これにより, 実行時の情報を用いて最適化するにもかかわらず, 実行時の情報を用いることによって発生するオーバーヘッドを除去することを可能にする. この時, 解析のためにコードを実行する時間が長くなり過ぎないように, 工夫をする必要がある. 解析のためのコードも実際に目的の計算を実行するコード

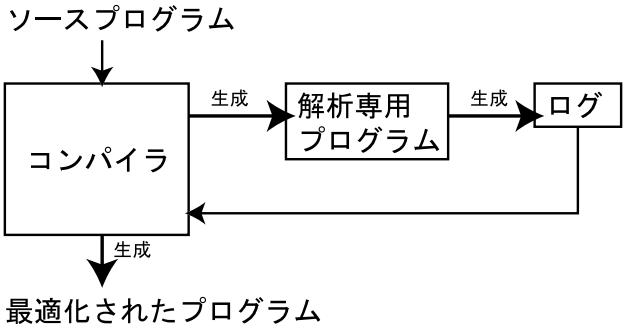


図 3.1: 実行時の情報を利用するコンパイラ

も, もとは同じソースプログラムであり, 実行時書き換えのようにバイナリをソースコードと対応づけて変換していく必要がなく, 目的の計算を中断, コードの変更, 適切な位置とメモリの状態で再開する必要がないので容易であり, 解析と反映のコストが小さい.

本方式は, コンパイル時にソースコードの一部を実行するため, コンパイル時間が増加する. しかし, 計算物理のプログラムは核になる特定の計算を繰り返すために実行時間が長いので (3.4) , 核になる計算を少しでも強力に最適化できれば, 実行時間の増加を回収することができる.

3.4　計算物理のプログラムの特徴

計算物理のプログラムが持つ特徴は実行時間が長いことである. このようなシミュレーションの実行は数日から数週間に及ぶ. 計算物理のシミュレーションには計算規模を少しでも大きくしたいという要求があるので, このようなプログラムは長い実行時間がかかる傾向がある. また, 長いシミュレーション時間と高価なハードウェアのため, かかる費用は莫大なものになる. 例えば日立の超並列計算機 SR2201 は管理費に月 500 万ほどかかる (2002 年現在). この費用は 1 分あたり 116 円になる. 現実的には, このような並列計算機の全てのノードを占有して行うシミュレーションは少なく, ノードをいくつかのグループに小分けして同時に複数のシミュレーションを走らせる場合が多いが, これでも一本のシミュレーションあたりの費用は大きい. 例えば,8 つのグループに分けて 1 週間のシミュレーションを実行すると, 一本のシミュレーションあたりの費用は 15 万円ほどになる.

計算物理のソースプログラムが持つ特徴は, 実行効率に大きな影響を与えている. またその特徴をうまく利用すれば効果的な最適化ができる. 計算物理のソース

プログラムは2つの大きな特徴を持っていると考える. 一つ目の特徴は計算物理のソースプログラムが典型的な構造を持っている点である. もう1つはコーディングするプログラマの問題である. 計算物理が持つ典型的な構造は, 最適化に極めて有効である. しかし, 多くのシミュレーションプログラムはコーディングの問題から実行効率が悪いため, 強力な解析をするコンパイラが必要になる.

3.4.1　典型的な構造

本研究のコンパイラが対象にするシミュレーションプログラムは典型的な構造を持っている場合が多い. そこで, その構造に着目して最適化を行なう.

計算物理のプログラムには典型的な構造がある. 図 3.2 は典型的な計算物理のシミュレーションのプログラムを表している. 計算物理のプログラムは核になる計算を持っていて, これを莫大な回数繰り返す. 核になる計算を繰り返すループを *OUTER* ループと呼び, 核になる計算の内部で実行されるループを *INNER* ループと呼ぶ.

このようなプログラムは次のように並列に実行される. 中心になる配列変数がプロセッサに分割され, *INNER* ループのいくつかが繰り返しをプロセッサに分配される. 繰り返しがプロセッサに分配された *INNER* ループは分割された配列変数が保持するデータの依存関係に基づいて, 通信を行なって並列に動作する (核になる計算). *OUTER* ループはこれを繰り返すが, 通信の原因になるデータの依存関係は, *OUTER* ループの繰り返しや入力データで変わらないプログラムがほとんどである.

シミュレーションプログラムが図 3.2 のような構造をとる場合がほとんどである理由は次のためである. 核になる計算の1回分の処理は, シミュレーションの単位時間当たりの処理を行なう. 典型的なシミュレーションではこれを莫大な回数繰り返し, ある時間の範囲で物理的な状態がどのように変化するのか調べる. この時, シミュレーションの時間を表すループが *OUTER* ループで, 1単位時間での状態の変化を計算するために使われるループが *INNTER* ループである.

例えば, 図 3.3 は空間のエネルギー状態の変化を調べるシミュレーションの例である. このシミュレーションでは空間のエネルギーを配列変数で表現し, 核になる計算を一度行なうたびに現在の配列変数の値から 1nsec 後の状態を求めて, 配列変数の値を更新する. このシミュレーションでは 1sec 後の状態を求めるために, *OUTER* ループで核になる計算を 1, 000, 000, 000 回繰り返す. また, 核になる計算の中では空間のエネルギーを表している配列変数の値を更新するために, 配列変

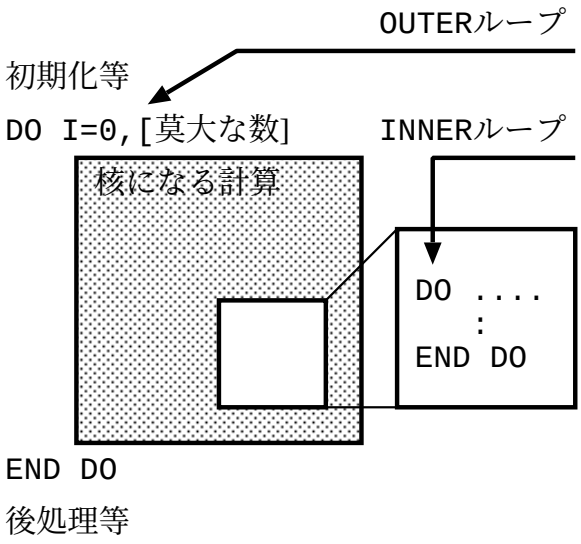


図 3.2: 典型的な計算物理のプログラム

数の全ての要素の新しい値を求めている. 空間のエネルギーを表している配列変数は 3 次元の配列変数であるため, 3 重の *INNER* ループを持っている. 通常, 並列化はいくつかの *INNER* ループを並列化することで行なわれる. なぜなら *OUTER* ループにおいて空間のエネルギーを表す配列変数は帰納変数であるので, 古い値から新しい値を求める処理しなければならなく並列化が難しいからである.

そこで本コンパイラはコンパイル時に *OUTER* ループの繰り返しを一度だけ調べて, 全ての繰り返しで同じプロセッサ間のデータの依存が発生すると仮定してコードを生成する. プログラムのどのループが *OUTER* ループであるかは, プログラマが指定する. このため, 本研究のコンパイラは実行時にプロセッサ間のデータの依存が変わるようなプログラムは, 正しくコンパイルできない. 入力のデータが変わるとプロセッサ間のデータの依存が変わったり, 核になる計算を繰り返すうちに核になる計算でプロセッサ間のデータの依存関係が変わるようなプログラムは処理できない. なぜなら, 本研究のコンパイラはコンパイル時にとった実行時のデータでコードを最適化しているからである. このため, コンパイル時と実行時でデータの依存関係が変わるようなプログラムは, 必要なタイミングで必要なデータが通信でやりとりされなくなる. 例えば Nas Parallel Benchmarks の IS (マルチプロセッサによる整数の整列) は正しくコンパイルできない. なぜなら, IS は整列する数列の内容で通信パターンが決定するからである. しかし, 提案するコンパイラも現実的であると考えている. なぜなら, 実際に自然科学者から次のような発言を

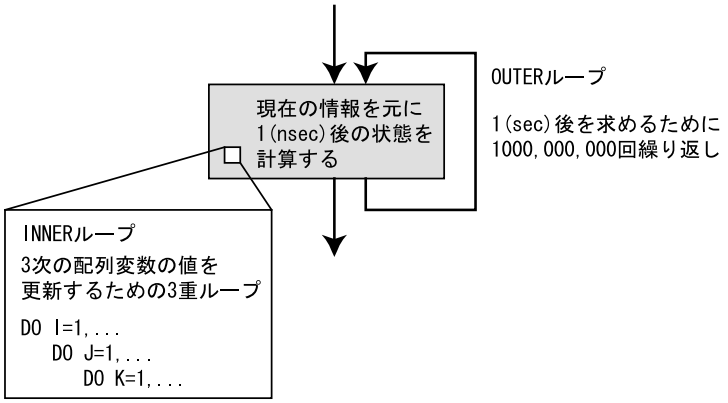


図 3.3: 空間のエネルギー状態の変化を調べるシミュレーション

いただいたからである. 計算物理の典型的なシミュレーションプログラムはある程度複雑なプロセッサ間のデータ依存を持つので, インスペクタ-エグゼキュータのような手法が必要な場合が多い. そのような場合でも, 実行中にプロセッサ間のデータの依存関係は変化しない.

3.4.2　典型的なコーディング

計算物理のプログラムの多くは, 実行するハードウェアの細かいアーキテクチャを考慮しないで作られる. この理由は二つある. 第1の理由はプログラマが計算機の専門家でないためである. まず, プログラマは物理, 天文などの専門家である. このため, 書かれたプログラムが最適化されていることは期待できない. もう一つの理由は, 次に計算機の特性は把握するには複雑だからである. このような計算機は利用方法が難解であることと, 機種毎にある独自の仕様が多い.

多くの並列計算機のハードェアは少しでも効果的な通信を利用できるように, 個性的な機能を持っている場合が多い (例えばRDMA:2.2.4 節). この理由は常に実行速度を最優先して設計されるからである. しかしこのため, プログラマから活用されない機能があることが多い. もし, このハードェアが低機能でも簡潔で広く使われている通信のライブラリなどを持っている場合, プログラマはそのようなライブラリを利用する.

しかし, このようなハードウェア固有の機能は一般的に標準的なインターフェースであるMPIやPVMからは利用できない. これを利用するためには, 例えばCP-PACSではRDMAを直接制御しなければならない. 標準的な通信のインターフェー

スはプログラマに環境に依存しないインターフェースを提供するために, 一般的な機能のみを提供し, ハードウェアの特徴を隠蔽しているからである.

3.5　提案する最適化手法

提案する手法は, 目的プログラムの通信を最適化する. これは分散メモリ機での実行は通信がボトルネックになりがちなのと, 通信には個性的なハードウェアが多く, 効果的に利用されにくいからである. また, 対象にするプログラムは計算物理のプログラムで, ループを $INNER$ と $OUTER$ に区別できるものを対象とする.

この節では以降, コンパイラが行なう最適化に関して説明する. コンパイラが行なう最適化は, ループ分配による通信量の削減, メッセージの融合, TCW の再利用, 実行時の情報の定数の畳み込みである.

3.5.1　ループ分配による通信量の削減

提案する手法は, ループの繰り返しをプロセッサに分配するとき, 通信の量 (バイト数) が少なくなるように分配する. ループをマルチプロセッサで並列処理する場合, データの分散の仕方が同じでもループの分け方で通信量が大きく変わる. ある反復をあるプロセッサに処理させた場合, 分散処理された配列変数にアクセスしても通信が必要ないとする. この場合, この反復でアクセスするデータは全てそのプロセッサが持っている. この反復を別のプロセッサが処理すると, この別のプロセッサはこの反復でアクセスするデータを得るために, 最初のプロセッサから通信でデータをもらわなければならない.

図 3.4 はプロセッサ数 2 の場合の, 最適なループの分割の模式図である. 並列化したいループの繰り返しは受け持ったプロセッサによって通信量が変わってくる. そこで, それぞれの反復に関して, 通信量が最も少なく処理できるプロセッサが反復を受け持つようにする.

本手法は, プログラマが指定したデータの分散の仕方に基づき, 通信量が少なくなるようにする. プログラマはHPFのディレクティブでデータの分散処理の仕方の指定と並列化可能なループをコンパイラに伝える. 通常のHPFコンパイラと同様に, 提案するコンパイラは`DISTRIBUTE`, `PROCESSORS`, `BLOCK`等で配列変数をプロセッサに分割する. また, 本手法は`INDEPENDENT`で並列化可能なループを知り, 必要に応じて並列化する.

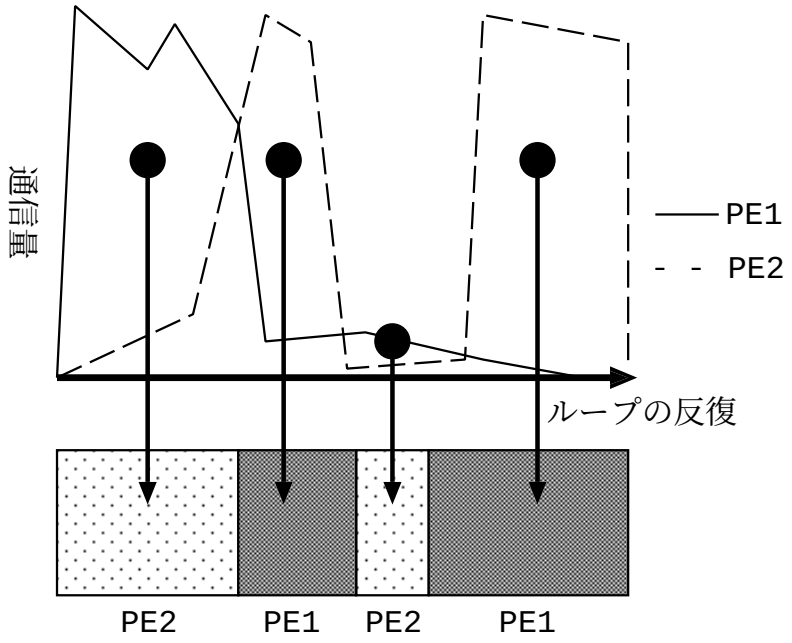


図 3.4: 最適なループの分割

3.5.2　メッセージの融合

提案する手法は, 同時に送信できる複数のメッセージを１つのメッセージにして, 通信の回数を少なくする. この最適化は, 通信でやりとりするデータ量はそのままで通信の回数を減らす. 通信にはデータが届くまでのレイテンシや通信のたびに必要になる装置の初期化などの時間がかかるので, 通信量はそのままでも, 回数を減らすことで通信にかかる時間を減らすことができる. また, RDMA にあるブロックストライド通信など, ハードウェアが特殊な単位の通信機構を持っている場合, これを直接利用すればMPIなどのインターフェースを使うよりもより強力にメッセージを融合できる可能性がある.

3.5.3　TCW の再利用

提案する手法は, TCW の再利用を試みる (図 2.8). 同じプロセッサに同じデータ (同じアドレス, サイズ等) を繰り返して送る場合, TCW を再利用することで通信にかかる時間を減らすことができるので, 本手法は同じプロセッサに同じデータを送る箇所を解析して探す.

3.5.4　実行時の情報の定数畳み込み

提案する手法は, 実行時の情報をコードに定数として畳み込む. 定数畳み込みをすることで, コンパイラが生成するコードは定数畳み込みをしないで同じような通信の最適化を行なったコードに比べて利点がある. まず一つめは変数をルックアップする回数が減ることである. これにより, 実行時間が短縮する. 場合によってはTCW の再利用のようにパラメータが定数でないと難しい最適化が可能である. 次に無駄な変数を除去し, メモリの消費量が減らせる可能性がある. しかし, 通常の実行時の情報を用いた最適化は定数の畳み込みをすることができない. なぜなら, 実行時の情報を得てからコードを生成する必要があるからである.

本手法は, プログラムの挙動を実行時の情報で制御するために使われるパラメータの参照を除去することで定数の畳み込みを行なう. 調整のために必要な変数を定数に置き換えて, バックエンドのコンパイラに定数伝搬の処理を任せることで, 通常の「実行時の情報を用いた最適化」の手法を改良する.

3.6　まとめ

この章では, 本研究が目標とするコンパイラを提案, 機能について論じた. 提案するコンパイラは実行時の情報を用いることで, 最適化のための強力な解析を実現する. コンパイラは実行時の情報をコンパイル時に用いて最適化を行うことで, コード生成時に実行時の情報を用いて最適化を行うことができる. この最適化は, 実行時の情報を用いた手法に発生する実行時の情報を回収したりプログラムの挙動を変更するようなオーバーヘッドを除去することができる.

強力な解析が必要な理由は二つある. 一つ目は, ハードウェアが持つ特殊で扱いにくいが, 効果的に利用することで実行時間の短縮を期待できるハードウェアにあわせた最適化を行うためである. 二つ目は, プログラマが計算機の専門家でないので, 効果的なコードを手動で記述することができない, またはその労力をかけさせたくないためである.

提案するコンパイラは, 実行時の情報をコード生成時に利用できるようにするため, ソースプログラムの一部を解析のためにコンパイル中に実行するようにした. この解析はインスペクタ-エグゼキュータの手法を応用するようにした. この手法だと, コンパイル中にソースコードの一部分を実行するため, コンパイル時間が増加する可能性があるので, 典型的な計算物理のシミュレーションのプログラムの構造を利用して, 解析のためのコストを減らすことにした.

提案するコンパイラは実行時の情報を用いて, ループ分配による通信量の削減, メッセージの融合, TCW の再利用, 実行時の情報の定数畳み込みなどを行なう. コンパイラは, 実行時の情報をコード生成時に利用することでより強力な最適化を行なう. これを実現するためにソースプログラムの一部分をコンパイル時に実行し, 実行時の情報を集めて, 改めてコンパイラがコードを生成するようにした. これのよって, 実行時にランタイムで最適化をする手法ではやりにくい最適化を行なう.

第4章　OREコンパイラの実装

本研究では 3 章で提案した手法を実際に実装した. この章では実際に実装したコンパイラの仕様と, それがどのような最適化をどのように行なったのかを内部の処理に焦点をあてて詳しく説明する. 3 章で提案した手法を用いたコンパイラを二種類実装した. 本章ではこれらを比較しながら詳しく説明していく.

4.1　OREコンパイラの仕様

実装したコンパイラ (ORE コンパイラ) は, 次のような仕様を持つ. 入力は HPF のディレクティブを含んだ Fortran で, 出力は RDMA のライブラリを使って通信を記述した SPMD の CP-PACS / Pilot-3 用の Fortran[47] である. 入力されるソースコードは Fortran77 ライクな並列化に向いていない記述が想定されている. 例えば Fortran90 配列断面のようにコンパイラが容易に並列化できる記述が効果的になされていなくても, コンパイラは効果的なコードを生成する. 出力されるプログラムは 3.5 節で説明した最適化がなされている. ヒントとして活用するディレクティブは, HPF の `INDEPENDENT`, `PROCESSORS`, `DISTRIBUTE`, `BLOCK` と独自のディレクティブ `INSPECTONCE` である. このディレクティブは OUTER ループを INNER ループから区別するために用いられる.

本研究では, 計算物理のシミュレーションのためのコンパイラを二種類実装した. 1 台の PC の上で動作するコンパイラと, PC クラスタ上で動作するコンパイラである. 1 台の PC の上で動作するコンパイラは, 簡便性を重視しているが, 汎用性が低い (4.4 節). それに対し, PC クラスタ版はより幅広い条件に対応できる.

4.2　CP-PACSとPilot-3

CP-PACS と Pilot-3 は汎用目的に設計された筑波大学計算物理学センターが所有する超並列計算機である. この計算機は分散メモリ機で, RDMA (Remote DMA)[44] と呼ばれる特殊な通信ハード ウェアを持っている. CP-PACS は 2048, Pilot-3

は 128 個のノードを持ち, それぞれ 3 次元と 2 次元のハイパークロスバーで接続されている. CP-PACS の理論ピーク性能は 614Gflops であり, 両機のノード間通信のピーク性能は 300Mbytes/sec である.

RDMA は MPI[33][34][45], PVM[36][35][45], HPF[31][46] と RDMA 独自のライブラリから利用することができる. しかし, RDMA のハードウェアが持つ特殊な機能を利用するためには, RDMA 独自のライブラリを用いなければならない. RDMA がサポートする特殊な機能にはブロックストライド通信, TCW 再利用型通信, 片側通信などがある. RDMA に関しては 3.1 節で詳しく説明する.

4.3　インスペクタエグ-ゼキュータの利用

本研究のコンパイラはインスペクタ-エグゼキュータ方式を利用する. コンパイラの並列化のターゲットはループであり, 最適化の対象は通信であるので, ループ並列化のために通信を解析するインスペクタ-エグゼキュータ方式と合致する. そこで本研究では最適化のためにインスペクタ-エグゼキュータ方式を用いる. ただし一般にインスペクタ-エグ-ゼキュータ方式で並列化できるループはループが運ぶ依存がないループ (ループの繰り返し同士の間に依存関係がないループ) であるので, 本方式で並列化できるループもプログラマが `INDEPENDENT` ディレクティブでループが運ぶ依存がないことを保証しているループだけである.

本研究のコンパイラが生成するインスペクタとエグゼキュータは, 通常のインスペクタ-エグゼキュータと大きく異なる. まず, コンパイラは並列化のためだけでなく, 最適化のためにインスペクタ-エグゼキュータを用いる. また, コンパイラはインスペクタの解析結果でコードを最適化しなければならない. このため, インスペクタ処理をコンパイル中に行なうようにした.

本研究のコンパイラはインスペクタとエグゼキュータのコードを別々に生成する. コンパイラはまず, インスペクタだけを生成して, これをコンパイルの一処理として実行する. 次に, インスペクタによって得られたプロセッサ間のデータの依存情報を解析し, 最終的に実行コードを生成する. この実行コードはエグゼキュータだけを含んでいて, インスペクタを含んでない. これによって, コンパイラはエグゼキュータのコードに様々な種類の静的な最適化技術を適用することができる. 生成コードは静的に最適化されたエグゼキュータを含んでいる. しかし, 生成されたコードはインスペクタのコードを含んでいないので, 次のようなプログラマの保証が必要になる. 必要になる保証は, $OUTER$ ループの繰返しでプロセッサ間のデータの依存関係が変わらないという点である. しかし, 典型的な計算物理, 天文

などのシミュレーションプログラムはこの条件を満たすので, 大きな制約ではないと考える.

本研究のコンパイラはターゲットマシンでインスペクタを行なわない. このため, コンパイラは制限が加わったり, コンパイルのために PC クラスタが必要になる. コンパイラがターゲットマシンでインスペクタを行なわない理由は社会的なものである. コンパイラはインスペクタの実行を行ない, それが終了するとコンパイルを再開し目的のコードを生成し, ユーザが目的のコードでシミュレーションを行なう. このため, コンパイルと実行を合わせて二回, ターゲットマシンにタスクを投入する必要が生じてしまう. しかし, 多くの並列計算機や本研究がターゲットにする並列計算機ではユーザはタスクをバッチ処理とキューで投入しなければならず, 投入毎に自分のタスクが処理され始めるのを他人のタスクが処理され終るまで待たなければならない [42]. このため, コンパイル中にインスペクタをターゲットマシンに投入する方式をとると, インスペクタの処理にかかる時間にキューの中で開始を待つ時間が加わることになり, 現実的でない.

4.4　PC 上への実装

本研究で実装した 1 台の PC 上で動作するコンパイラは, 制限を持つ. なぜなら, このコンパイラはインスペクタ処理で全てのプロセッサの動作を解析しないからである. 本来, 並列計算機の上で並列に実行されるインスペクタの処理は 1 台の PC 上で行なわれるには計算量が多い. このため, このコンパイラのインスペクタは並列計算機の 1 プロセッサ分しか解析しない. このコンパイラはその解析結果を全プロセッサに適用できる情報として処理する. このため, このコンパイラは, 全てのプロセッサに関して, リモートにあるデータの依存関係が同じでないと処理できない. プログラマはソースプログラムがこの条件を満たしていることを保証しなければならない.

このコンパイラが対応する最適化はメッセージの融合 (3.5.2 節), TCW の再利用 (3.5.3 節), 実行時の情報の定数畳み込み (3.5.4 節) である. ループ分配による通信量の削減 (3.5.1 節) は行なわない. なぜなら, ループ分配による通信量の削減を行なうためには, 並列化したいループの全ての反復に関して, どのプロセッサが受け持つのが最適なのか調べなければならない. このためには, マルチプロセッサで並列に実行されるループを 1 台の PC で調べなくてはならなくなるので, 1 台の PC が処理するのには計算量が多き過ぎる.

4.5　PC クラスタ上への実装

本研究で実装した PC クラスタ上で動作するコンパイラは, 1 台の PC 上で動作するコンパイラに比べ, 制約を緩くするために PC クラスタ上で動作する. このコンパイラは 1 台の PC 上で動作するコンパイラに比べ, 効率よりも汎用性を重視している. このコンパイラが PC クラスタ上で動作する目的は, オリジナルのインスペクタと同様にインスペクタを並列で動作させることである. これによって, 1 台の PC 上で動作するコンパイラにある問題, 「全てのプロセッサに関して, リモートにあるデータの依存関係が同じであることをプログラマが保証しなければならない」という問題を解決できる.

PC クラスタ上で動作するコンパイラは並列実行に用いたいプロセッサ数 +1 のノード数で動作する. PC クラスタ上で動作するコンパイラは実機のプロセッサと PC クラスタのノードを一対一に対応づけて動作する. また, ノードを制御したりノード間で依存が強い情報を処理するために, メインマシンが 1 台ある. クラスタ側ではコンパイルのためのデーモンプログラムが動作して, 並列にコンパイル処理を行なえるフェーズをメインマシンからこのデーモンプログラムにリクエストを送ることで処理する.

このコンパイラが対応する最適化はメッセージの融合 (3.5.2 節), TCW の再利用 (3.5.3 節), 実行時の情報の定数畳み込み (3.5.4 節), ループ分配による通信量の削減 (3.5.1 節) を行なう. ループ分配による通信量の削減を行なうためには, 並列化したいループの全ての反復に関して, どのプロセッサが受け持つのが最適なのか調べなければならない. このためには, マルチプロセッサで並列に実行されるループを調べなくてはならなくなるので, 実機のプロセッサ数と同じ台数だけの PC クラスタノードが妥当である. このため, 1 台の PC 版ではできなかったループ分配による通信量の削減を行なうことができる.

4.6　コンパイル処理の流れ

本研究のコンパイラは図 4.1, 4.2 に示される流れで並列化を行なう. どちらのコンパイラもインスペクタ処理を行なうが, PC クラスタ用は行なう最適化の種類が多いのと, クラスタのノード間で情報を交換しなければならないので, フェーズが多い. 以下にそれぞれのコンパイラに関して詳しく説明する.

まず, 1 台の PC 版に関して説明する. 1 台の PC 版は受けとったソースプログラムもとにインスペクタ専用のコードを生成する. このフェーズは Fortran から

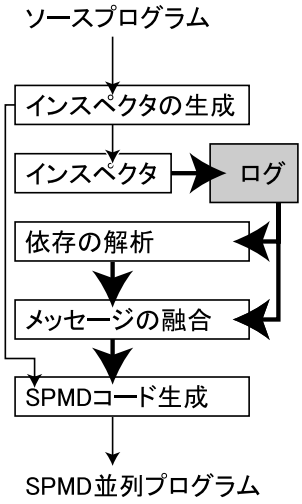


図 4.1: 本コンパイラの処理 (1 台の PC)

Fortran への変換で, 内部から f2c と gcc を呼び出してバイナリにする. 次に生成されたバイナリ (インスペクタ) を system で呼び出して, ノード間のデータの依存に関する情報をハードディスクに書き出させる. この時, プロセッサ一台分しか解析しない. 次に処理をコンパイラに戻して, ハードディスクから依存に関する情報を回収する. 次に回収された依存の情報を解析して, 実際に通信を行なうために必要な情報 (送りたいデータのアドレス, 送りたい相手のプロセッサなど) を調べる. 次にメッセージの融合で, 同時に送信可能なデータを探しだし, ハードウェアがサポートする最大サイズに近くなるようにまとめられるだけまとめる. この後, ソースプログラムと得られた通信命令から SPMD プログラムを生成する.

次に, PC クラスタ版に関して説明する. PC クラスタ版はインスペクタ処理を PC クラスタの各ノードの上で実行するので, ソースプログラムを処理に参加するノード全てに配る. 次に, 各ノードの上でインスペクタ専用のコードを生成する. このとき各ノードは 1 台の PC 版と同様に Fortran で書かれたソースプログラムから Fortran で書かれたインスペクタプログラムに書き換えを行い, f2c と gcc を呼び出してバイナリにする. 次に, 各ノードは生成されたバイナリ (インスペクタ) を system で呼び出して, ノード間のデータの依存に関する情報をハードディスクに書き出させる. 各ノードはこの情報をハードディスクから回収し, どんな通信が必要になるか解析する. このあと, 情報をメインマシンに集めて, メインマシンは並列化するループの各繰り返しをどのプロセッサに割り当てるのが効果的か解析する. この後, 各クラスタのノードは並列計算機の自分が担当しているプロセッサが処理

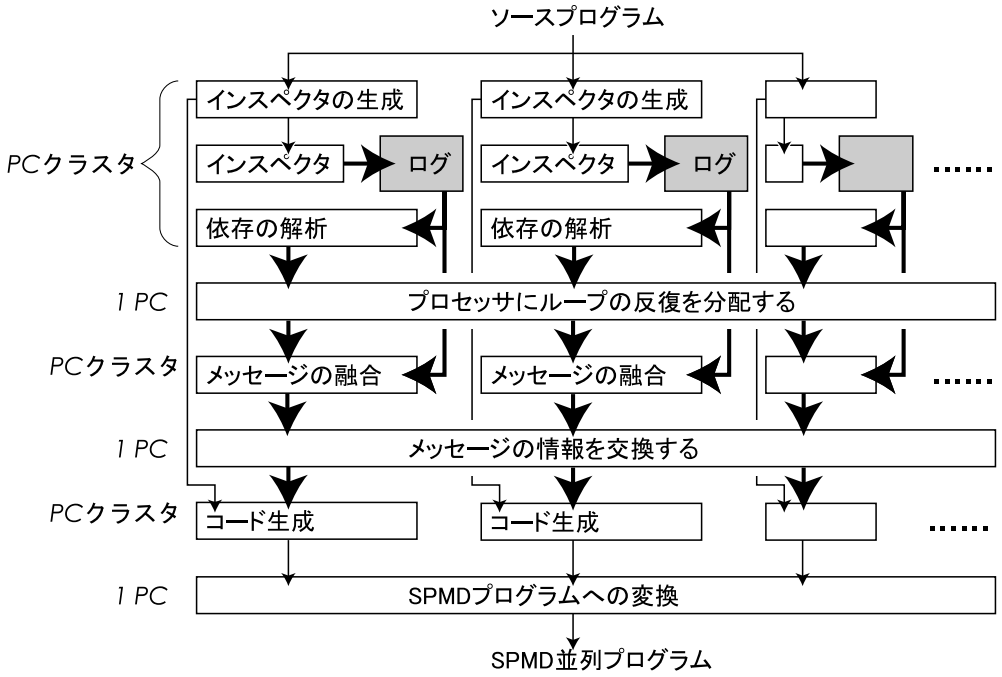


図 4.2: 本コンパイラの処理 (PC クラスタ)

しなくてはならないループ反復をメインマシンからもらう. 次にメッセージの融合で, 同時に送信可能なデータを探しだし, ハードウェアがサポートする最大サイズに近くなるようにまとめられるだけまとめる. 次に, 各クラスタのノードは並列計算機の自分が担当しているプロセッサが実行するコードを生成する. このあと, 各ノードが生成したコードをメインマシンに集めて SPMD プログラムに変換する.

4.7　重要なフェーズの処理

この節で実装したコンパイラのユニークな処理を説明する.

4.7.1　インスペクタ

通常のインスペクタは解析結果をメモリ上に保持し, それをそのまま同じプログラム上にあるエグゼキュータに渡すが, 本研究のコンパイラはハードディスクに書き出す. 通常のインスペクタ-エグゼキュータはプログラムの一回の実行で実行中にインスペクタとエグゼキュータを実行するので, インスペクタの解析結果をメモ

リ上に置くだけで, それをそのままエグゼキュータが利用することができる. しかし, 本研究のコンパイラではインスペクタとエグゼキュータ (コンパイラの生成物) とコンパイラはインスペクタの解析結果にアクセスするにもかかわらず, 別プログラムとして動作する. このため, インスペクタの解析結果をメモリ上に置くことは適切でない. また, インスペクタの解析結果がどの程度の容量になるのかインスペクタのコンパイル時に推測しにくい. なぜなら, インスペクタ-エグゼキュータが利用されるのは, プログラマやコンパイラがどのような通信が必要になるかということを理解しにくい場合だからである. そこで, 本研究のコンパイラではインスペクタの解析結果をハードディスク上に置くことにした.

本方式のインスペクタがとる記録は, マルチプロセッサに分散配置される配列変数へのアクセスと, それを囲むループの挙動である. これらの値は, アクセスがそのプロセッサの受け持ち分をはみ出した時に記録される. 通信は配列がプロセッサに分割される次元に関して, 配列の添字がはみ出した時に発生し, はみ出した時の添字の値がわかれば通信が必要な相手のプロセッサやデータのアドレスなどを知ることができる. この配列変数が LVAL ならば送信で相手に送らなければならなく, LVAL でないのであれば受信で相手から受けとらなければならない. インスペクタが解析するデータは, 具体的に次のものである. ソースコード上の位置, アクセスしている要素番号, それを囲むループのインデックス値である (図 4.3). ソースコード上の位置とは, ソースプログラムを構文解析した時に見つかった順でつけられた, 分散配置する配列変数のアクセス箇所の通し番号である. この番号は一意な番号で, 構文解析することで何度でも同一の番号を得ることができる. アクセスしている要素番号とは, 配列変数のどの要素にアクセスがあったかということを指す. この番号は, 一度のアクセスあたり, その配列変数の次元の数だけ記録される. それを囲むループのインデックス値は, そのアクセスがいつ発生するのかということを知るために記録される. この番号は, 一度のアクセスあたり, そのアクセスを囲むループの数だけ記録される. これらは隣接した値をとりがちなので, それぞれの値に関して差分を取り, ランレングスで圧縮しながら保存する (図 4.4).

実装したコンパイラで用いるインスペクタは, 通常のインスペクタと同様に, 配列変数の添字を監視するコードを配列変数のアクセス箇所に埋め込む (図　4.5). 監視するコードは関数の形で実装され, 配列変数をアクセスする要素番号を受けとり, 要素番号をもとに処理を行ない, その要素番号をそのまま返す. この関数の呼び出しは, マルチプロセッサに分散配置される配列変数の添字に埋め込まれる. この関数は [ソースコード上の位置] と配列変数の次元の番号をキーにして情報を整理し, 圧縮しながら適宜ハードディスクにデータを書き出す. これらの関数は状態

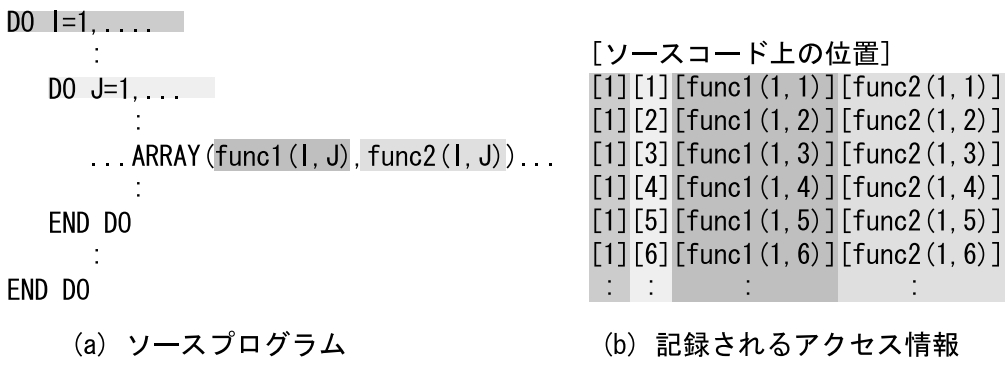


図 4.3: 配列変数へのアクセス記録

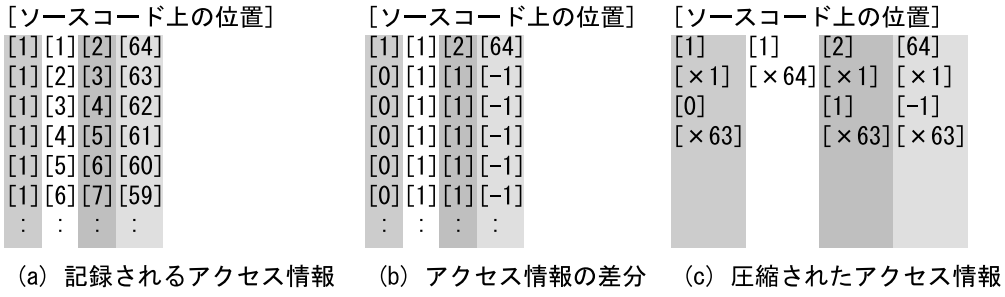


図 4.4: 配列変数へのアクセス記録の圧縮

を持つため, 初期化とフラッシュを行なうための関数とプログラムの先頭と終了に呼び出しが埋め込まれる.

実装したコンパイラは, データのアクセスがある時刻をループのインデックスの値で管理するので, 制御変数を持たないまたは判別しにくいループを持つ場合, 円滑に処理するために仮の制御変数を組み込んで処理を行なう. このようなループにはサブルーチン呼び出しによるものや, GOTO 文によるものなどがある. 図 4.6 はサブルーチン呼び出しによって発生する疑似的なループの例である. ここでは (a) でサブルーチンを二回呼び出している. このため, 実際は (b) で示されるようにサブルーチンボディを二回実行するループと同じであるが, 制御変数は存在しない. もし, サブルーチンでリモートのデータアクセスがある場合, インスペクタはこのアクセスを記録しなければならないが, サブルーチン呼び出しに制御変数は存在しないので, 最初の呼び出しでリモートのデータアクセスがあるのか, 次の呼び出しでリモートのデータアクセスがあるのか, 区別することができない. そこでコンパイラは, サブルーチンや関数を見つけると図 4.7 のように疑似的な制御変数を埋め

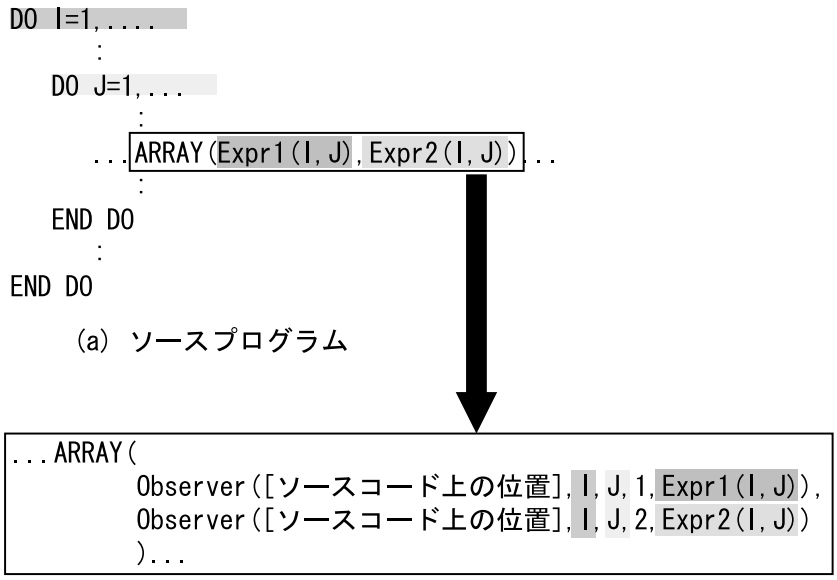


図 4.5: アクセス監視の呼び出しの埋め込み

込む. サブルーチンや関数の最初に疑似ループ制御変数の確保や初期化を行ない, END と RETURN 文の前に疑似ループ制御変数のインクリメントを行なう. しかし本手法では, ENTRY 文のようにサブルーチンの途中から実行を行なうような記述がある関数やサブルーチン, GOTO 文によって作られたループがある場合は, 正しくコンパイルできない可能性がある. これらに正しく疑似制御変数の埋め込みを行なうためには, 複雑な制御フローの解析が必要になる.

実装したコンパイラは, 一部の分割される配列変数のアクセスをインスペクタで監視しないで, 静的に解析する. コンパイラが全ての解析を実行時情報で行なわない理由は, 処理の重さである. この理由により, 静的な手法の中で特に安易なもので解析できるアクセスをインスペクタで解析することを避ける. また, 本研究は高度な静的な解析が目的ではないので, 簡単な静的な解析しか行なわない. これにより, 例えば配列変数を初期化したり, 同じ分割方向を持つ配列変数の間で行なう単純なコピーをするためのループと配列変数のアクセスを静的に処理する. 具体的には, 本コンパイラは, 以下のような配列変数のアクセスをインスペクタで解析しない. ここでは, その配列変数を囲み並列化される可能性のあるループ (INDEPENDENT のついたループ) 中でアクセスされる全ての分散処理される配列変数を AR と呼び, アクセスを AC と呼ぶ.

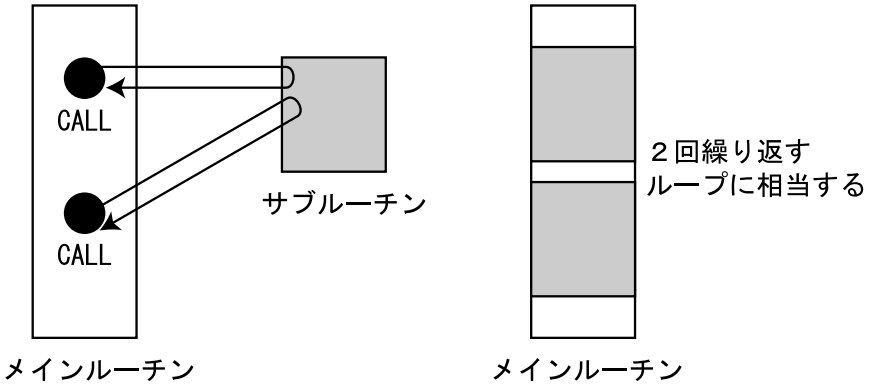


図 4.6: サブルーチンによる疑似ループ

```
SUBROUTINE SUB1(...)
      :
      :
END
```

(a) 擬似制御変数挿入前

```
SUBROUTINE SUB1(...)
  INTEGER LOOPI
  SAVE LOOPI
  DATA LOOPI/1/
    :
    :
  LOOPI=LOOPI+1
END
```

(b) 擬似制御変数挿入後

図 4.7: 疑似ループ制御変数の埋め込み

- AR の分割される次元の素数が等しい
- AR の分割指定が揃っている
- AC の分割される次元の添字の式が変数の参照
- AC の分割される次元の添字の式が全て同じ変数
- AC の分割される次元の添字の式が並列化される可能性があるループ制御変数
- AC の分割される次元の添字の式が並列化される可能性のある配列変数
- AR の分割指定と AC でアクセスされている幅が揃っている

このような AC はそれを取り囲み並列化される可能性のあるループを配列変数の BLOCK 指定に従って分割することで, 通信なの並列化を行なうことができる.

1 台の PC 上で動作するコンパイラと PC クラスタ上で動作するコンパイラのインスペクタは異なる点がある. これらの違いは主に, ループをマルチプロセッサに分割する時期の違いによって生じる. 1 台の PC 版ではインスペクタを行う前にループを分割しなければならなく, PC クラスタ版ではインスペクタの後にループを分割しなければならない. まず, 1 台の PC 版は計算能力の関係で, ターゲットマシンの 1 台分しかインスペクタを行わないので, インスペクタ前にループを分割し, その分割にしたがってインスペクタを生成する. 次に, PC クラスタ版では, インスペクタはループの繰り返しをどのようにプロセッサに分配するか決定するための情報を集めたいので, インスペクタ処理の時点でループを分割することができない. PC クラスタ版のインスペクタは, 全てのクラスタノードがマルチプロセッサで並列化する予定のループの全ての繰り返しを試験する. また, 1 台の PC 版はインスペクタを生成するフェーズで, DEF-USE チェーンを使って解析以外の計算を除去して最適化を行う. しかし, 目的の計算の一部をインスペクタのプログラムから除去しておけば, バックエンドのコンパイラがほぼ同様の最適化を行うため, 本コンパイラ内での最適化の効果が薄い. このため, PC クラスタ版ではこのような最適化は実装されていない.

4.7.2　ループのくり返しの分配 (PC クラスタ版のみ)

実装したコンパイラは, プログラマが指示した HPF ディレクティブ (DISTRIBUTE , BLOCK 等) に従って配列変数をプロセッサに分割する. 配列全体はいくつかのブロックに分けられて, 全プロセッサはそのうちの一つをローカルメモリ上に持つ. プログラマは, 配列がどのように分割されて, どのプロセッサが受け持つかということに責任を持つ.

実装したコンパイラは INDEPENDENT ディレクティブのついたループを発見した場合, 必要に応じてそのループの反復をプロセッサに分配する. この処理はプログラマの手助けを必要とせず, 自動的に行われる. また, INDEPENDENT ディレクティブのついたループが多重ループになっている場合, コンパイラは試験的にそれぞれのループを本節で説明する手法で分割し, 最終的に分割して発生する通信量の合計が少ないループを分割する.

以下のような状況を考える. DO ループの反復回数が n で, プロセッサ数が N の場合, 単純な分配では, 各プロセッサは連続した反復を $\frac{n}{N}$ 受け取る. しかし, この

ような単純な分配と, 通信で扱われるデータ量を最小にする分配は異なる可能性がある.

通信で扱われるデータ量を最小にするために, 実装したコンパイラは反復をそれぞれ最適なプロセッサへ分配する. コンパイラは, プログラマが指示したデータ分散に基づいて各反復ごとに最適なプロセッサを探す. この処理は, PC クラスタ版のインスペクタが可能性のあるループの反復の配置の全てに関して発生するであろう通信量を調べた後に, この情報を用いて行われる.

実装したコンパイラは, 以下の方法で全ての反復に関して実行したときに最も少ない通信量ですむプロセッサを探す. 反復の分配はインデックスの値が小さい順に行われる. (1)i 番目の反復に関して, 配置した場合に通信量が最も少なくなる一意のプロセッサを探す. (2) もし, (1) の条件で複数のプロセッサが選ばれた場合, この中から $(i-1)$ 番目を処理するプロセッサを選ぶ. (3) もし, (2) の条件を満たすプロセッサがなかった場合, (1) の条件を満たすプロセッサの中で, 最も受け持ちの反復数の少ない一意のプロセッサが処理を受け持つ. (4) もし, (3) に該当するプロセッサが複数合った場合, (3) の条件を満たすプロセッサのうち最も小さい番号のプロセッサが受け持つ.

このアルゴリズムでは, 全ての反復が一台のプロセッサに割り当てられる可能性がある. この場合, ループの反復は逐次に行われることになる. しかし, そのような場合に無理に並列処理を行うことは, 通信量によってかえって実行時間が長くなる危険性が強いので, コンパイラは逐次に実行するプログラムを生成する.

実装したコンパイラは, 図 4.8 のようにして連続しない反復の分配を実現する. 通常, ループをプロセッサに分割する際, 各プロセッサは連続した反復を受け取る. これにより, ループの上限と下限を調整することで, 全プロセッサが他のプロセッサと全く重複しない反復を得ることができ, 並列化を行うことができる (a). しかし, コンパイラは連続しない反復をプロセッサに配る. そのような反復は一組の上限と下限だけでは表現できない. そこで, コンパイラは分配した反復が一つの領域に連続している場合, 図 4.8(a) に示されるようにコードを生成し, 分配した反復が複数の領域に分かれている場合, 特別なループをつくることにより複数の上限と下限から構成されている不連続な反復を繰り返すコードを生成する.

図 4.9 は図 4.2 で説明した PC クラスタ版のコンパイラの処理の流れの一部分を抜き出したものである. この図はループの分割に関係した処理の一連の流れである. インスペクタが回収した情報 (a) はソースコードのどこで, ループ制御変数がいくつのときに, どの配列変数のどの要素がアクセスされたかということに関した情報である. 「通信量の集計」はこの情報を用いて, 並列化の候補になっている

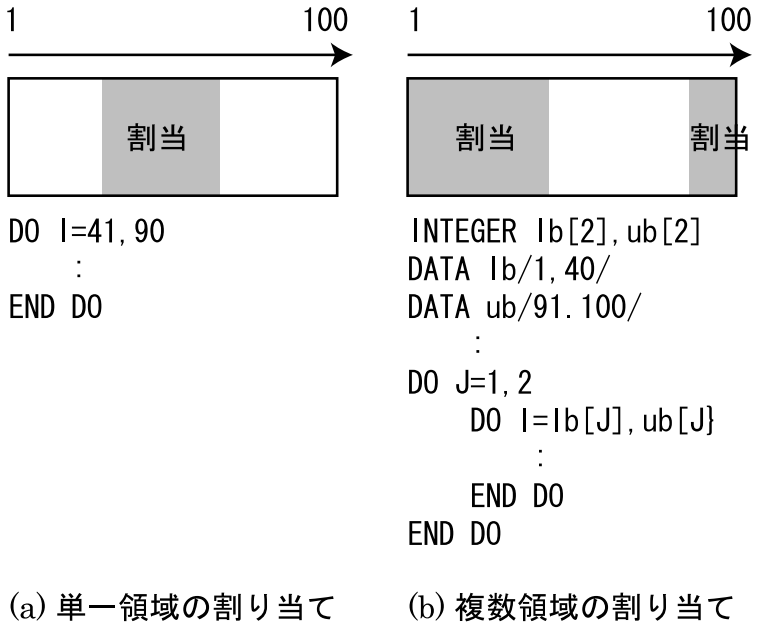


図 4.8: 連続しない反復の割当

ループの全反復に関して, このノードに対応するプロセッサが各反復を実行した場合, どのくらいの通信量 (bytes) が発生するか数える. この結果 (b) は通信でメインマシンに渡され, この節で説明したアルゴリズムに従って各プロセッサが受け持つべきループの反復を調べる (「プロセッサにループの反復を分配する」) . この結果 (c) はクラスタの各ノードに渡される. インスペクタが回収した情報 (a) はこのプロセッサがループの全繰り返しを受け持ったときに必要になるリモートデータであり, 実際はこの全てのデータは通信で扱われない. なぜなら, 実際に生成されるコード (エグゼキュータ) が実行するループの繰り返しは (c) の範囲だからである. そこで, コンパイラは「必要な通信の選別」で, 実際に必要になるリモートデータのアクセス (d) だけ取り出す.

4.7.3　ブロックストライド通信の利用

実装したコンパイラは, 可能な場合, 複数のメッセージを一つのメッセージにまとめて送信する. この処理によって, 通信の回数を減らすことができる. メッセージを送る時, 全ての通信は制御のためのオーバーヘッドを含む. このため, コンパイラは通信を減らすことで処理時間を短縮することができる.

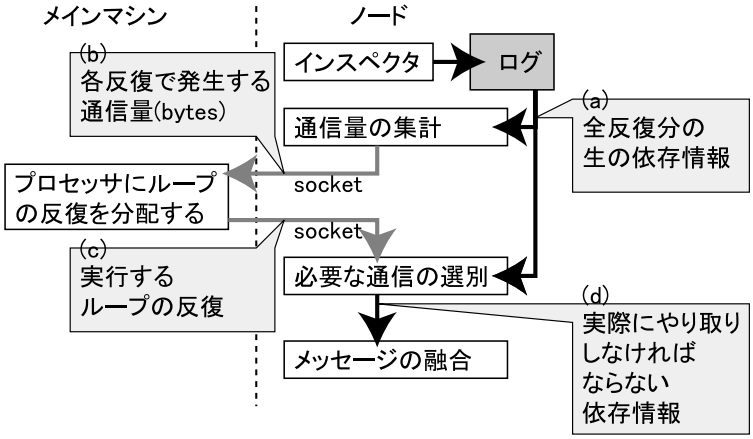


図 4.9: ループ反復の分配の処理

実装したコンパイラはブロックストライド通信を利用するために, 実行時の情報と HPF ディレクティブを用いる. INDEPENDENT 指定されたループの反復で送られるメッセージは融合できる可能性がある. INDEPENDENT 指定されたループは繰り返しの実行順序が計算結果に影響を及ぼさないことをプログラマが保証しているので, 必要になるリモートのデータを別の繰り返しで通信で入手しておいて, 一時変数に保存しておき, 必要になった時に参照することができる. このため, 図 4.10 のように, ループで必要になるリモートデータをループにはいる前にまとめて通信で得ることができる. もし, ソース上の同じ配列参照で発生したリモートデータのアクセスがブロック-ストライドの形にのった場合, 一つのメッセージにまとめることができる. INDEPENDENT 命令は, そのループの反復の実行順番が結果に影響しないことをコンパイラに伝えるための命令である. また, INDEPENDENT を指定されたループは, ループが運ぶ依存を調べることなく並列化できる.

実装したコンパイラは, 各反復で送られるメッセージのデータの送信元のアドレスと受信先のアドレスを調べる. また, この処理に必要なプロセッサ間のデータの依存をインスペクタが調べる. コンパイラは送受信のデータがブロックス-トライドにのるように集める. コンパイラは, このようにして集めたメッセージを一つのメッセージとして送信するコードを生成する. ターゲットマシンの通信機構 (Remote DMA) はそのようなデータを余計なメモリコピー無しで直接ターゲットのメモリに送信できる. コンパイラは RDMA によってハードウェアでサポートされたブロック-ストライド通信を用いる. RDMA は等間隔不連続なデータを送信する場合, パックして連続した一つのメッセージとして送信する. 受信側でデータで

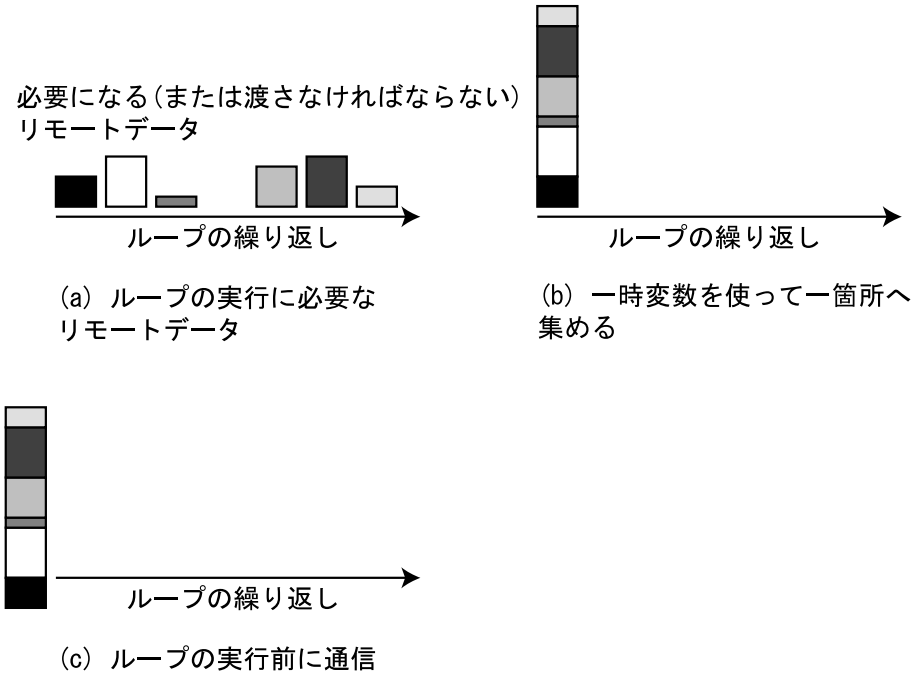


図 4.10: INDEPENDENT 指定されたループ上の通信の移動

ハードウェアはメッセージをアンパックし, 指定したアドレスに指定した間隔で展開する.

実装したコンパイラは次のようにして, 同時に発行できる通信を探し, ブロックストライド単位の通信を生成する. まず, 同時に発行できる通信を探す方法である. (1) まず, 必要になるリモートアクセスを発生した「ソースコード上の位置」ごとに分類する. これで, プログラム中の異なった箇所の無理な融合を避ける. (2) 次に, (1) で分類した各分類に関して図 4.11 に示されるようにテーブルを用いてリモートアクセスを分類, 融合する. (a) は融合する前の状態である. ここでは 2 重ループの繰り返しの間にリモートアクセスがあり, 一方は通常のループで他方は INDEPENDENT 指定されたループである. 通常のループは繰り返しの実行順序を守らなければならないので, 通信を融合することはできないが, INDEPENDENT 指定されたループは実行順序を変えることができるので, 通常のループに関してはループの繰り返の番号で整理して, INDEPENDENT のループの方に関してはループの繰り返し番号を無視して融合する. これにより同時に通信で処理できるリモートデータの集合 (b) を得ることができる. また, テーブルがメモリを使い過ぎないようにするため, 本コンパイラは 4.4 節で説明した圧縮をしたまま処理を行なう.

通常のループの繰り返し番号	INDEPENDENT指定されたループの繰り返し番号	リモートデータ
[1]	[1]	[アドレス0x0000から始まり0x10バイト]
[1]	[2]	[アドレス0x0050から始まり0x08バイト]
[2]	[4]	[アドレス0x0080から始まり0x20バイト]
[2]	[5]	[アドレス0x00a0から始まり0x20バイト]
[2]	[6]	[アドレス0x00c0から始まり0x10バイト]
[3]	[3]	[アドレス0x0030から始まり0x10バイト]
:	:	: :

(a) 記録したアクセス情報

通常のループの繰り返し番号	INDEPENDENT指定されたループの繰り返し番号	リモートデータ
[1]	[*]	[アドレス0x0000から始まり0x10バイト, アドレス0x0050から始まり0x08バイト]
[2]	[*]	[アドレス0x0080から始まり0x50バイト]
[3]	[*]	[アドレス0x0030から始まり0x10バイト]
:	:	: :

(b) 融合されたリモートアクセス

図 4.11: テーブルを使った通信の融合

実装したコンパイラは, 同時に通信できるリモートデータの集合を, パターンマッチングによってブロックストライドの集合に変換する. パターンマッチングは全てのリモートデータがブロックストライドで表されるまで, 繰り返す. この処理は次のようになる. (1) アドレスの小さい側から 1byte 以上の連続したリモートデータの集合を一つ探す. (2) 次にこの領域を一つ目のブロックとみなして, 次のブロックを探す. このブロックが一つ目のブロック未満のサイズを持っている場合 (4) の処理を行なう. 二つ目のブロックが一つ目のブロック以上のサイズを持っている場合, 二つめのブロックの下側から一つ目のブロックのサイズだけ取りだし, これを改めて二つめのブロックにする. 一つ目のブロックと二つ目のブロックでストライドが決定される. (3) 決められたストライドに従って三つ目以降のブロックを探す. このブロックが十分なサイズを持っていない場合 (4) の処理を行なう. このブロッ

クが十分なサイズを持っている場合はブロックストライドの繰り返しのか図を 1 増やして, 改めて (3) の処理を行なう. これ以上ブロックが見つからなければ一組のブロックストライドが完成する. (4) もし, 現在見つかっているブロックストライドのブロック幅を小さくすることによって, 次のブロックがブロックストライド上に載り, かつブロックストライドの容量が増すのであればそうする. そうでなければ, これで一組のブロックストライドが完成する. このアルゴリズムは必ずしも最適な解を出すとは限らないが, 計算量と効果のバランスからこの手法を採用した.

図 4.16 は同時に通信できるリモートデータの集合を, ブロックストライドの集合に変換する例である. ここではサイズの違うブロックが等間隔にならんでいる (a). まず, 最初のマッチングで一つ目のブロックが決定する (b). 次に, 次のブロックを探し, サイズが同じなのでこれを二つ目のブロックとし, ストライドが決まる (c). 次に, 3 つ目のブロックを決まっているストライドで調べると, 十分なサイズがない. しかしこの場合, ブロックのサイズを 4 から 3 にすることで 3 つ目のブロックをブロックストライドに載せることができ, かつ容量が $4 \times 2 = 8$ から $3 \times 3 = 9$ に増加するので, ブロックのサイズを 3 に改めて, 3 つ目のブロックをブロックストライドに載せる (d). 次のブロックはブロックストライドに載らないが, ブロック幅を 2 にすると, $2 \times 4 = 8$ と容量が減ってしまうので, ブロックストライドが確定する (e).

4.7.4　実行時の情報の定数畳み込み

実装したコンパイラは実行時の情報をコンパイル時に得られるので, インスペクタ-エグゼキュータ方式に必要なテーブルの参照を定数の畳み込みで削除する. 典型的な通常のインスペクタ-エグゼキュータの実装では, インスペクタはプロセッサ間のデータの依存を表したテーブルを生成する. エグゼキュータはこのテーブルを参照し, 送信または受信する相手のプロセッサやデータのアドレスなどを決定する. コンパイラは, このテーブルの参照を除去する. なぜなら, コンパイラではこのテーブルを定数として扱えるからである. このテーブル参照はループの制御変数と定数による線形な式に置き換えられる (図 4.12) .

インスペクタによって得られた情報は定数なので, その情報をもとに通信命令を生成することで, テーブル参照が除去された (図 4.12) コードを生成することができる. コンパイラは, 通信命令を直接プログラムに埋め込むことで定数の畳み込みを行なう.

```
DO I=1,n
    IF(TABLE[I])THEN
        SETTING(TABLE[I])
        SEND(TABLE[I])
    END IF
        :
        :
END DO
```

(a) 最適化前

```
SETTING(定数または簡単な式)
SEND(定数または簡単な式)
DO I=1,n
        :
        :
END DO
```

(b) 最適化後

図 4.12: テーブル参照の除去

しかし単純に通信命令を生成するだけでは, 莫大な個数の通信命令を生成してしまいがちである. なぜなら, 4.7.3 節で説明した手法で通信命令を生成すると, INDEPENDENT 指定のないループの繰り返し毎に通信命令が必要になるからである. 例えば図 4.11 では通常のループの制御変数が 1 の時, 2 の時, 3 の時用にそれぞれ通信命令と制御のための IF 文を生成する必要がある.

そこで, 実装したコンパイラはループ制御変数で通信命令の数を減らすことを試みる. コンパイラのターゲットは RDMA でありブロックストライド通信を利用するので, 配列変数のオフセット, ブロックサイズ, ストライドサイズ, ブロックストライドの繰り返しをループ制御変数で縮退させなければならない.

実装したコンパイラは, 生成しなければならないブロックストライドの通信の集合を, パターンマッチングによって制御変数でまとめる. パターンマッチングは制御変数でまとめられなくなるまで, 繰り返す. この処理は次のようになる. (1) ループのインデックスが小さい側から通信を一つ探す. このインデックスを i とする. (2) 次に $i+1$ に通信がなければこれでコード上に生成される一つの通信命令とする. $i+1$ に通信があれば, 配列変数のオフセット, ブロックサイズ, ストライドサイズ, ブロックストライドの繰り返しをループの制御変数を使って線形に表す式を

```
DO I=1,n
        :
    SETTING(ループ中で定数)
    SEND(ループ中で定数)
        :
END DO
```

(a) 最適化前

```
ID=SETTING(ループ中で定数)
    :
DO I=1,n
        :
    SEND(ID)
        :
END DO
```

(b) 最適化後

図 4.13: TCW を再利用するコード

求める. 二点が決まれば線形の式は求まる. (3)$n = 2$. (4) 次に $i + n$ に通信がなければこれでコード上に生成される一つの通信命令とする. $i + n$ に通信があるが, その通信が線形に予測される $i + n$ をメモリ上で完全に包含しないのであれば, $i + n$ は含めずにこれでコード上に生成される一つの通信命令とする. $i + n$ に通信があり, その通信が線形に予測される $i + n$ をメモリ上で完全に包含するのであれば, 線形に予測される $i + n$ をその通信から削除し, n の値を一つ増やし (4) を繰り返す. このアルゴリズムは必ずしも最適な解を出すとは限らないが, 計算量と効果のバランスからこの手法を採用した.

4.7.5　TCW 再利用型通信の利用

もし, 通信に必要なパラメータ (対象のプロセッサ, アドレス等) が定数ならば, コンパイラは TCW (Transfer Control Words) 再利用型の通信を使って, 実行速度を向上できる. TCW はメッセージの送信先の設定が収納されるデータブロックである. この場合, エグゼキュータはループの反復毎に TCW を準備する必要がなくなる (図 4.13) .

表 4.1: 二種類の SPMD 変換方法

ソース	単純な命令の結合	命令単位の融合
PID=0:	IF(PID. EQ. 0)THEN	
A(I)=A(I)+1	A(I)=A(I)+1	
	ELSE	A(I)=A(I)+1+PID
PID=1:	A(I)=A(I)+2	
A(I)=A(I)+2	END IF	

4.7.6　SPMD コードの生成 (PC クラスタ版のみ)

1PC の上で動作するコンパイラが生成したコードはほぼそのままで SPMD プログラムになる. なぜなら, 全てのプロセッサがリモートデータに関して同じ依存関係を持つことをプログラマが保証しているので, 全プロセッサがほぼ同じ動作をすることが保証されているからである.

しかし, PC クラスタ上で動作するコンパイラは各クラスタノードで生成したコードをメインマシンで SPMD に変換しなくてはならない. これを実現するためには, シミュレーション実行中にプロセッサの識別子を保持している変数 PID を用いて, 各プロセッサ用のコードを融合しなければならない. この処理を行なうには全てのプロセッサのコードが必要になるので, 全コードをメインマシンに集めて処理する.

実装したコンパイラは, PC クラスタのノードが生成した複数のプログラムを命令単位で SPMD にする. 複数のプログラムを PID で一本の SPMD にまとめるには単純に IF 文で接合する方法 (図 4.17) が考えられる. しかしこの方法では, プロセッサ数とコード長が比例の関係をもってしまうので, スケーラビリティが悪い. そこで, コンパイラはより強力な融合方法で SPMD コードを生成する. もし, 融合したい全てのコードが酷似した計算で構成される場合, IF 文でつなげるよりは PID を計算式に埋め込んだ方が効果的で短いコードを生成することができる. シミュレーションのように計算の多いプログラムではその可能性が高い. 表 4.1 は酷似した計算を融合する例である. この場合, 計算 $A(I) = A(I) + 1$ と $A(I) = A(I) + 2$ は $+1$ と $+2$ しか違わない. これを IF 文でつなげると, PID の条件判定の計算が増え, 二つの式がそのまま記述されてしまう. それに対し, 本手法では式が一つで済み足し算が一つ増えただけである.

本研究の SPMD の生成方法は対応している式を探し, その式が命令単位で融合可能であれば式単位で融合し, それが困難な場合は IF 文で接合する. この時, 本方式で接合する部分と IF 文で接合する部分が交互に現れるとその度に IF 文の分岐

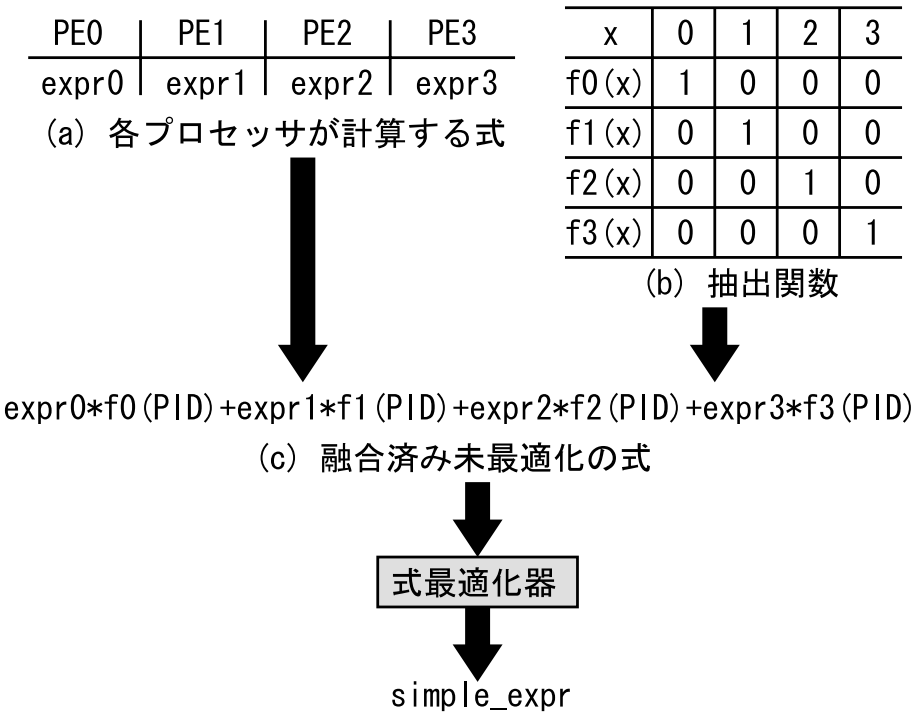


図 4.14: 命令の融合

が現れて, 効果的でないコードを生成してしまう可能性があるが, 本研究ではこの点を考慮していない.

開発したコンパイラは図 4.14 で表される流れで式を融合する. (a) は各プロセッサに実行させたい計算を表している. ここでは PE0 に $expr0$ を実行させなければならない. もし (b) に示されるように, 0 を代入すると 1 を返して他の数 (1〜3 の整数) を代入すると 0 を返す式 $f0(x)$ があれば, $f0(PID) \times expr0$ を計算することで PID が 0 の時に $expr0$ を返し, それ以外で 0 を返す式ができる. 同様に $expr1$〜$expr3$ に対して同様の式を作り, + 算で接合すれば, PID 毎に目的の式を得られるようになる (c). $f0(x)$〜$f3(x)$ を満たす式はガウスの消去法を用いれば整式として得ることができる. 得られた式は複雑な計算を持ち, 無駄が多いので式最適化器に通して最適化を行なう. しかし, この方法では高次のガウスの消去法を行なうことが難しいので, 4 個単位で式を融合する. この場合, 組み合わせる 式の数 -1 次のガウスの消去法を行なわなければならない. 本方式では 9 次の計算で, 計算中に 32bit の整数範囲をオーバーフローしてしまう.

式最適化器は, 括弧を展開して変数を含む消去可能な項を消去しながら, 一意な多項式を生成する. 指数や関数や配列参照はある程度対応するが, その関数が持つ

表 4.2: 式最適化ルール (抜粋)

変換前	変換後	備考
(A)	A	
$-(-A)$	A	
$A + B$	$B + A$	辞書順で B が A より前
$A \times B$	$B \times A$	辞書順で B が A より前
$A \times 1$	A	
$A \times 0$	0	
$A + 0$	A	
$A - 0$	A	
$A - A$	0	
$A \times (B + C)$	$A \times B + A \times C$	
$A \times (B - C)$	$A \times B - A \times C$	
$(A + B) \times C$	$A \times C + B \times C$	
$(A - B) \times C$	$A \times C - B \times C$	
$A \div 1$	A	
$A \div (-1)$	$-A$	
$A \times 1$	A	
$A \times (-1)$	$-A$	
$1 \times A$	A	
$(-1) \times A$	$-A$	

特殊な性質を考慮した計算は行なわない. この最適化器は式を木で表し, パターンマッチングで対応する公式 (表 4.2) にしたがって変換を行なう.

実装したコンパイラは SPMD を生成する際, もともとのソースコードで同じ箇所の式を優先的に組み合わせて融合する. なぜなら, 元の式が同じであれば融合後に極めて簡単な式を得られる可能性が高いからである. コンパイラでは, 各クラスタノードはソースコード上での行番号を埋め込んでコードを生成し, メインマシンはそれをヒントにして融合する.

融合の際には厳格でなく柔軟な行のマッチングが必要になる. ヒントに利用できる行番号は欠けている場合がある. これには大きく二つのケースがあり, コンパイラによって追加された命令は行番号がなく, コンパイラによって削除された命令は行番号が削除されてしまう. 例えば, コンパイラが追加した通信命令は行番号を持たないであろうし, 並列化されたループで反復を受けとらなかったプロセッサの

コードは削除されてしまい, その行番号はそのプロセッサのコードでは抜け落ちてしまう.

実装したコンパイラは各クラスタノードから渡された全てのコードを先頭から比較しながら融合する. このために, コンパイラは対応した行番号の命令を探しつつ, 対応した行が見つからなかった場合は IF 文で処理しなければならない. 図 4.15 は命令単位の融合と IF による結合を分類する過程を示している. まず, 行番号照合器は PE0 のコードを先頭から調べていき, 行番号がある位置まで読み進める (a). この時, 読み捨てた分は IF 文で接合して SPMD にするので, プールしておく (b). 行番号が見つかると, 同様に PE1 の処理を行ない行番号がある位置まで読み進める (a). この時 PE0 と同様に, 読み捨てた分は IF 文で接合して SPMD にするので, プールしておく (b). PE0 と PE1 で見つかった命令の行番号が等しいのであれば, 同様に PE2 と順次進めていく. もし, 行番号が異なるのであれば, PE0 か PE1 でソースコードの一部が削除されていることになる. そこで, PE0 と PE1 の行番号のうち, 小さい番号を持っている方を再び調べていき, 別の行番号つきの命令を探す. この時も読み捨てた命令は IF 文で接合するのでプールしておく (b). このようにして, 全プロセッサ分で同じ行番号の命令が見つかると, 命令単位の融合アルゴリズムで融合する.

命令単位の融合アルゴリズムが利用できるのは, LVAL が同じか同じ配列変数で別の要素を指している計算, IF 文, DO 文である. IF 文, DO 文は制御構造を持っているので, 対応する IF 文または DO 文を見つけた場合, 先にボディの融合を試み, 効果的に融合できれば IF 文や DO 文の制御部を融合する.

4.8　コンポーネント

開発したコンパイラは, 一台の PC 上で動作するものと PC クラスタ上で動作するものとでそれぞれ, 3 つと 4 つのコンポーネントから成り立つ. 一台の PC 上で動作するものはコントローラとインスペクタコード生成器と本コード生成器である. PC クラスタ上で動作するものはこれにノードデーモンが加わる. コントローラは全体の処理の流れを制御し, インスペクタコード生成器と本コード生成器はそれぞれインスペクタと最終的なコードを生成する専用のコンポーネントである. ノードデーモンはメインマシンからの仕事を受けとるためにネットワークを LISTEN しつつ, PC クラスタ上で行なう解析の処理をする.

コントローラは処理の中心になるコンポーネントである. PC クラスタ版ではメインマシン上で動作する. 他のコンポーネントを呼び出したり, プログラマからの

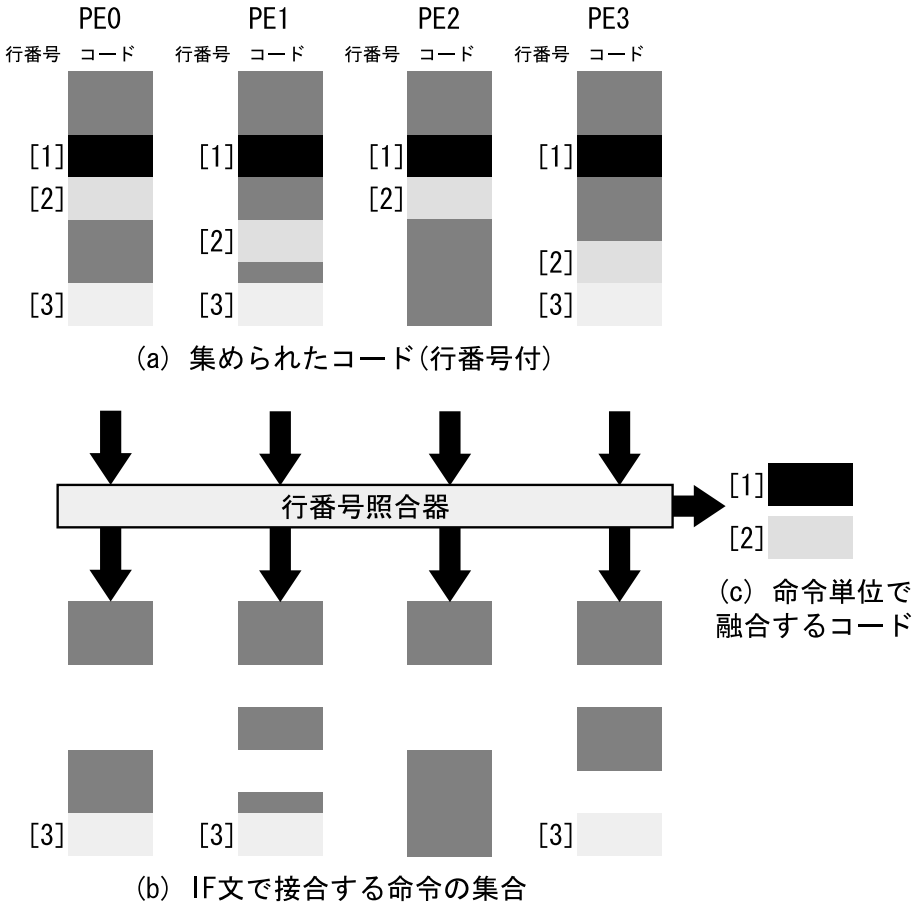


図 4.15: 行番号のマッチング

コンパイルの要求を受け付ける. 1 台の PC 版ではコントローラは system 関数などで他のコンポーネントを呼びだし, PC クラスタ版ではソケット通信でノードデーモンに要求を送る. また, 1 台の PC 版ではインスペクタが得た情報の解析処理を行ない, PC クラスタ版では並列に処理することが困難な解析を行なう.

ノードデーモンは PC クラスタ版にだけあるコンポーネントである. ノードデーモンは PC クラスタの各ノードで, コントローラからの指示を受けて適切な処理を行う. ノードデーモンは PC クラスタの各ノードでコントローラからの要求をソケットを LISTEN して待っている. このコンポーネントはクラスタのノードでコントローラの指示にしたがって他のコンポーネントを起動, 情報の回収, 解析を行う. ノードデーモンがターゲットマシンのプロセッサ数より多く動作している場合は, プロセッサ数と等しい数のノードデーモンだけが利用される. しかし, 足りな

い場合は一つのノードデーモンが複数のプロセッサの処理を行なう. このように複数の処理を受け持った場合, それらは並列でなく逐次に行なわれる.

インスペクタコード生成器は, インスペクタ用のコードを生成する. 1 台の PC 版ではコントローラから直接呼び出され, PC クラスタ版ではクラスタノードでノードデーモンから呼び出される. Fortran から Fortran への変換器を持ち, この結果を f2c と gcc を呼び出しバイナリにする.

本コード生成器は, ターゲットマシン用のコードを生成する. 1 台の PC 版ではコントローラから直接呼び出され, PC クラスタ版ではクラスタノードでノードデーモンから呼び出される. 生成されるコードは Fortran なので, バックエンドにターゲットマシン用のコンパイラが必要になる. PC クラスタ版ではこの結果をノードデーモンを通してコントローラに戻し, コントローラが SPMD を生成する.

4.9　まとめ

本章では実際に実装した二つのコンパイラ, 1 台の PC 上で動作するコンパイラと, PC クラスタ上で動作するコンパイラについて詳しく説明してきた. まず, 開発したコンパイラの仕様を述べ, 次に利用した実行時情報の解析方法, コンパイルの処理の流れの概要, 重要なフェーズであるインスペクタとその生成方法, ループの繰り返しを最適にプロセッサに分配することで通信量を抑える最適化の原理と最適化のアルゴリズム, ブロックストライドを利用することで通信回数を抑える最適化のアルゴリズム, 実行時の情報を用いて定数畳み込みを行なう手法, TCW 再利用型通信を利用するための手法などである. また, これらを行なう本コンパイラの構成に関しても説明した.

1 台の PC 上で動作するコンパイラは簡便であるが, 大きな制約があった. これはリモートにあるデータ間の依存が全てのプロセッサで同じであることをプログラマが保証しなければならない点である. それに対して PC クラスタ版はその制約がなく, より汎用的であるが多くの計算が必要になる.

本研究が行なった最適化の主だったものは通信に関するものと, 実行時の情報をテーブル参照なしにコードに埋め込む点である. これにより本方式は実行時にランタイムで動作を調整することだけで行なう最適化より強力な最適化を行なうことができた. この処理はコンパイル中にインスペクタを実行し, その情報をコンパイラが改めて回収することで行なうことができた.

また, PC クラスタ版では各クラスタノードが生成したコードを一本の SPMD にまとめる必要があったので, メインマシンでまとめるようにした. この時, プロセッ

サ数に対するコード長の増加を抑えるため, 命令単位での融合を試みるコンパイラを実装した.

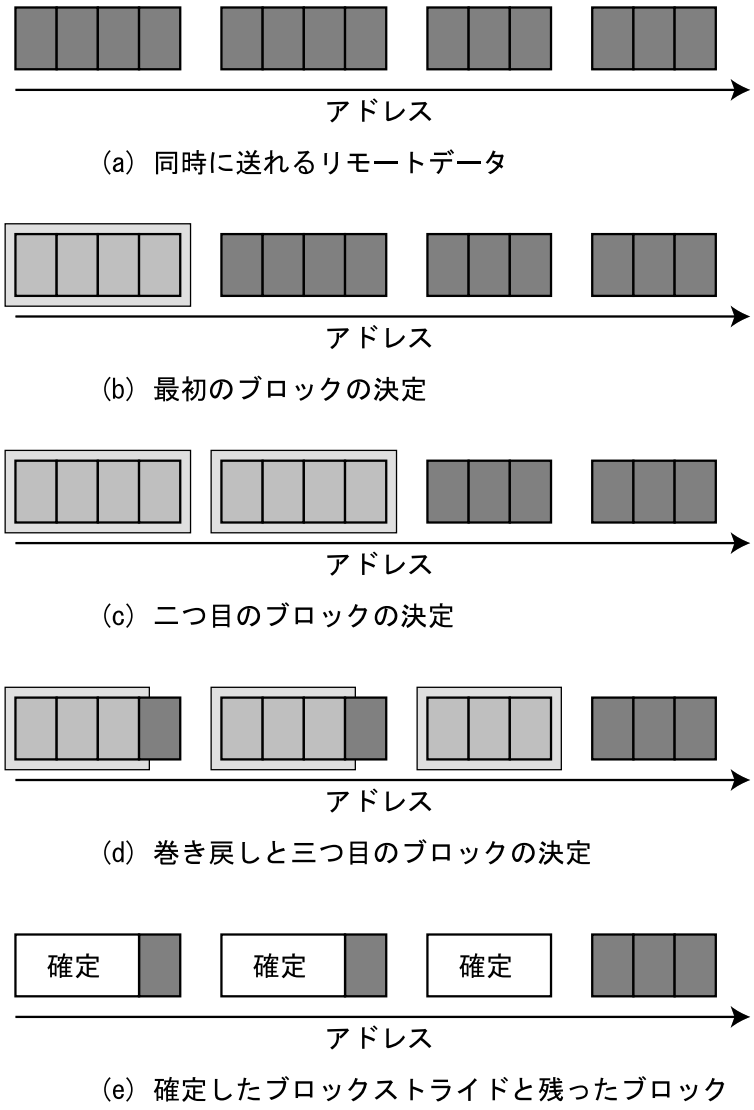


図 4.16: パターンマッチングを用いたブロックストライドの取り出し例

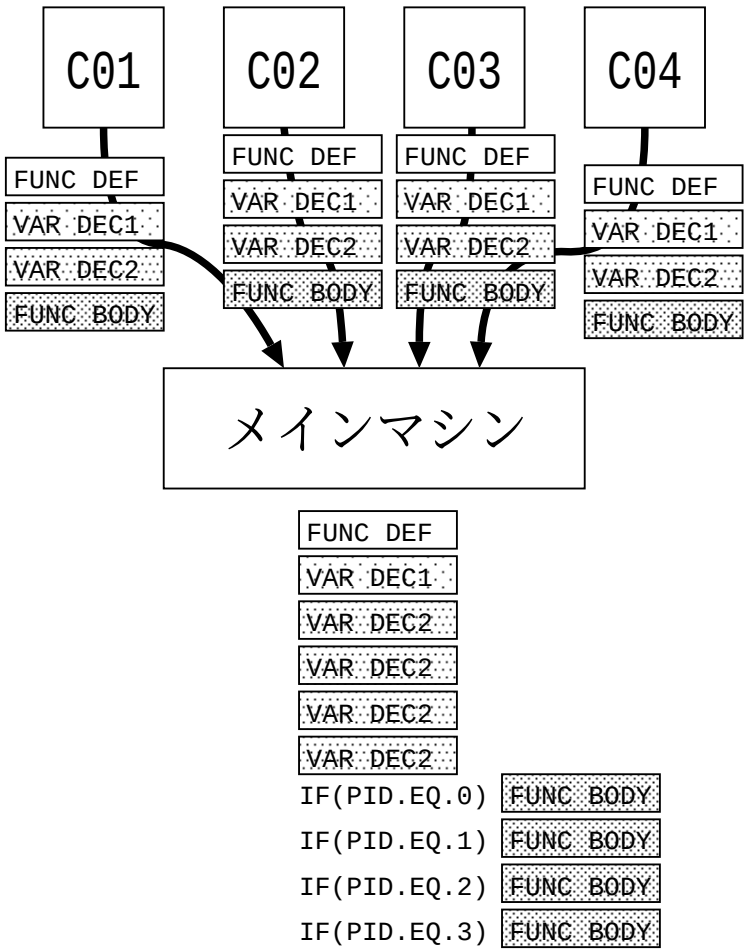


図 4.17: 単純なコード結合による SPMD 生成

第5章　実験と評価

本研究で実装したコンパイラの性能をベンチマークを用いて測定した．本研究の最大の目的は実行時間の短縮である．それに加え, コンパイル時間, 本研究の各最適化手法それぞれの効果, 一般的な通信ライブラリを使って人間のプログラマが最適化したコードとの比較などを行なった．コンパイル時間に興味を持つのは, コンパイル時間が通常のコンパイラよりも増加する可能性があるからである．この理由は, 本研究のコンパイラはコンパイル中にインスペクタを実行するからである．このため, コンパイル時間がソースコードのサイズだけでなく, ベンチマークが持つ計算量や INNER ループのくり返し数に影響される．また, コンパイラは複数の種類の最適化を行なうので, 各最適化プロセスがどの程度の成果があるのかということを, 実際のベンチマークを用いて測定した．また, 本研究が対象としている物理のシミュレーションではMPIなどの一般的な通信ライブラリでプログラムが記述されることが多いので, そのような場合で人間の手で最適化されたものと比較するとどの程度の差が出るのか測定した．

5.1　環境

実験に使ったターゲットマシンはPilot3中の16プロセッサである．また, コンパイラを実装したのは以下のような環境である．Linuxの Redhat 7. 1, PentiumIII 733MHz, メモリ512 Mbyte．コンパイルには17台からなるPCクラスタを用いた．バックエンドには日立製の最適化 Fortran (バージョン 02-06-/C + 02-06-XF + 02-06-XJ) を用いた．このバックエンドコンパイラはPilot3の1ノードの上で動作する．

5.2　ベンチマーク

使用したベンチマークは, genesis distributed benchmarks[37] から pde1, NAS parallel benchmarks[38] から FT と BT である．本研究はこの HPF 版 [39] を使っ

表 5.1: ベンチマークのコード長

	FT	BT	pde1
行数	736	2251	154

表 5.2: 分散処理される配列変数のアクセス箇所

	FT	BT	pde1
総数	33	1035	39
静的に解決できなかった個数	7	189	26

た. pde1 の仕事規模のパラメータ N は 7, FT と BT は class A で行なった. これらのベンチマークについてそれぞれ簡単に説明する.

5.2.1　pde1 (Genesis Distribute Benchmarks)

pde1 はポアソン変微分方程式を赤/黒反復解法で解くベンチマークである. pde1 は全プロセッサのリモート間のデータの依存が同じである. このアクセスは, 中核になる配列変数の境界部分に集中する. また, 分散配置される配列変数のアクセスが行なわれる箇所が, ソースプログラム上のごく一部である. 表 5.2 はマルチプロセッサに分割される配列変数に参照または代入を行なっているソースプログラム上の箇所の数である. pde1 は行数は FT より長い (表 5.1) が BT よりは短く, アクセスのある箇所は最も少ない FT とあまり変わらない. これは, ソースプログラム自体がローカルな一時変数をうまく利用して記述されているからである.

5.2.2　FT (Nas Parallel Benchmarks)

FT は 3 次元偏微分方程式を FFT を用いて解くベンチマークである. FT はプロセッサごとにリモート間のデータの依存が異なる. このため, 本方式の 1 台の PC 上で動作するコンパイラは受理できない. このアクセスは, 3 次元の巨大な配列変数を回転させるために生じる. また, 分散配置される配列変数のアクセスが行なわれる箇所の数, ソースプログラムの長さが, 今回用いたベンチマークの中で最も小さい (表 5.1, 表 5.2) . FT はローカルな一時変数をうまく利用して記述されているからである.

5.2.3　BT (Nas Parallel Benchmarks)

BT は非優位対角な 5 × 5 のブロックサイズを持つブロック 3 重対角方程式を解くベンチマークである. BT はプロセッサごとにリモート間のデータの依存が異なる. このため, 本方式の 1 台の PC 上で動作するコンパイラは受理できない. 分散配置される配列変数のアクセスが行なわれる箇所の数, ソースプログラムの長さが, 今回用いたベンチマークの中で最も大きい (表 5.1, 表 5.2) . BT はローカルな一時変数を利用して通信を避けるような記述がされていない.

5.3　評価

5.3.1　MPI, PVM との比較

本研究のコンパイラと, RDMA ではなく一般的な通信ライブラリを用いて人間のプログラマによって最適化されたコードの実行速度を比較した. これらのプログラムはノンブロックな通信で計算を隠蔽するようにしている. 本研究のコンパイラの利点の一つに RDMA を容易に利用できることがある. RDMA は MPI や PVM のような一般的な通信ライブラリより, ハードウェアを直接制御できる利点がある. それに対して, 多くのシミュレーションプログラムでは, 抽象化されていて扱いやすい MPI や PVM が使われることが多い.

実行結果は図 5.1 に示される. ただし, スピードアップの基準となるプロセッサ数 1 の実行時間は, 通信のない逐次なコードの実行時間を用いた. プロセッサ数 16 において, 実行時間が短いのは MPI, 本方式の 1PC 版, 本方式の PC クラスタ版, PVM であった. 本方式の 1PC 版は人間のプログラマが最適化した MPI のコードに比べて 86%の速度, PC クラスタ版は人間のプログラマが最適化した MPI のコードに比べて 73% の速度を得られた. また, MPI と PVM のプログラムのコンパイル時間は日立製の最適化 Fortran (バージョン 02-06-/C + 02-06-XF + 02-06-XJ) を用いて 14〜16 秒であった. 本研究のコンパイラは人間のプログラマが最適化した MPI のコードに比べて, 遅いという結果が得られてしまった. しかし, そのようなコードを記述するには, 詳しい知識を持ったプログラマが時間をかけてコードを記述しなければならない. それに対して, 本方式ではそのような知識や労力が要求されない.

PVM は最適化が行なわれているにもか変わらず, 他の方式にくらべ実行時間が長くかかった. PVM は他の通信ライブラリと異なり, 異機種間の接続が前提に設

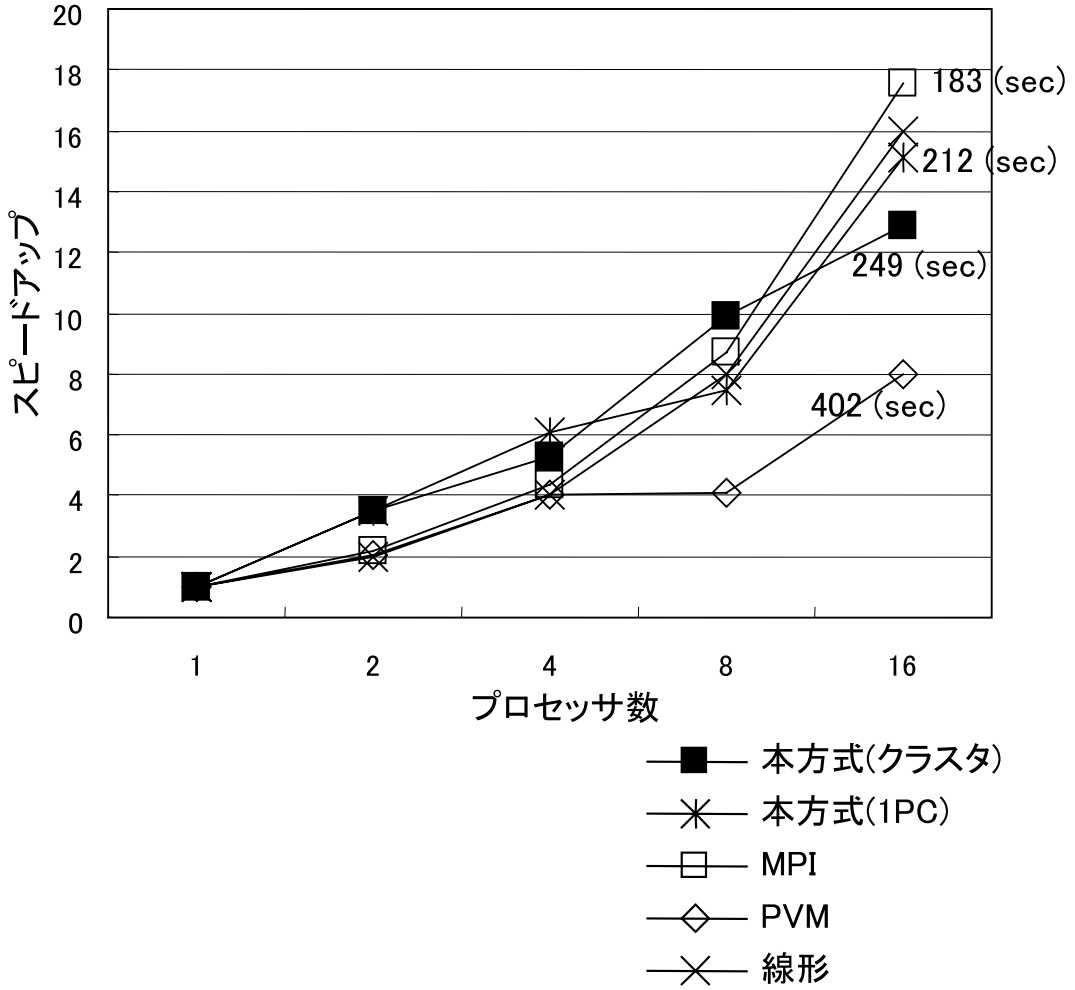


図 5.1: pde1, N=7 速度向上比 (MPI, PVM)

計されている. また, 全く異なる型や変数のデータを一度にまとめて送るために余分なメモリコピーが発生するために効率の点で劣ると考えられる.

開発した二つのコンパイラを比較すると, より強力な最適化 (3.5.4 節) を行なっている PC クラスタ版の方が 1 台の PC 版より実行時間を要している. 1 台の PC 版は, 正しくコンパイルできるプログラムの制限が厳しく, PC クラスタ版の方より汎用的でない. 1 台の PC 版は汎用的ではないが, もっとも速いコード (図 5.1) を生成できた.

5.3.2　コンパイル時間

本研究のコンパイラは典型的なコンパイラに比べて長いコンパイル時間がかかる. 例えば, 本研究のコンパイラは 1 台の PC 版でプロセッサ数 16 の pde1 をコンパイルするのに 140 秒, PC クラスタ版で 207 秒かかった. しかし, 通常のインスペクタ-エグゼキュータ方式のプログラムはは 85 秒でコンパイルできた. その差はそれぞれ 55 ($140 - 85$) 秒と 122 ($= 207 - 85$) 秒であり, 実行時間の短縮である 50 ($262 - 212$) 秒と 13 ($= 262 - 249$) 秒を上回った. 本方式を用いた場合, コンパイル時間の増加を実行時間の短縮で回収するためには, 十分な $OUTER$ ループの反復数が必要である. この実験の場合, $OUTER$ ループの反復数がそれぞれ 1100 回と 9400 回あれば (通常は 1000 回), コンパイル時間の増加を実行時間の短縮で回収することができる. しかし, コンパイル時間の増加は大きな問題ではないと考えている. なぜなら, 対象としているアプリケーションはシミュレーションであり, それらの $OUTER$ ループは莫大な反復数を持っているからである.

図 5.2, 5.3, 5.4 は PC クラスタ版のコンパイルの時間の詳細を示している. 実験に使ったベンチマークプログラムをコンパイルするために, この実験では PC クラスタを用いた. 実験に用いた PC クラスタのノード数はターゲットマシンのプロセッサ数と同じである. 例えば, ターゲットマシンのプロセッサ数 4 用にコンパイルする場合, コンパイラは PC クラスタの 4 ノードを用いる. これらの図では, コンパイル時間を 4 つに分類して示している. backend はバックエンドに用いたコンパイラが消費した時間, sequental は PC クラスタのマスターノードだけが動作した時間, parallel は PC クラスタの全ノードが並列に動作した時間, data exchange は PC クラスタのノード間のデータの交換で費やされた時間である.

この図はバックエンドのコンパイラの処理時間がプロセッサ数に対してほぼ比例していることを示している. なぜなら, SPMD コードがプロセッサの増加にともなって, 長くなる傾向があるからである. 本方式が行なっている SPMD への融合を改良して, より短い SPMD コードを生成できるように改良すれば, バックエンドのコンパイラで消費されている時間を短縮できる.

BT のコンパイル時間は極めて長く, 実用的でなかった. コンパイラの処理時間のほとんどが, PC クラスタのノード間でのデータの交換で使われていた. この理由は, 分散処理される配列変数へのアクセスがプログラム中にたくさんあるため (表 5.2), ノード間で交換されるデータが巨大になったためである.

図 5.5 は 1PC 版のコンパイルの時間の詳細を示している. 1PC 版はインスペクタ時にループをどのようにプロセッサに分割するかということが決まっているので,

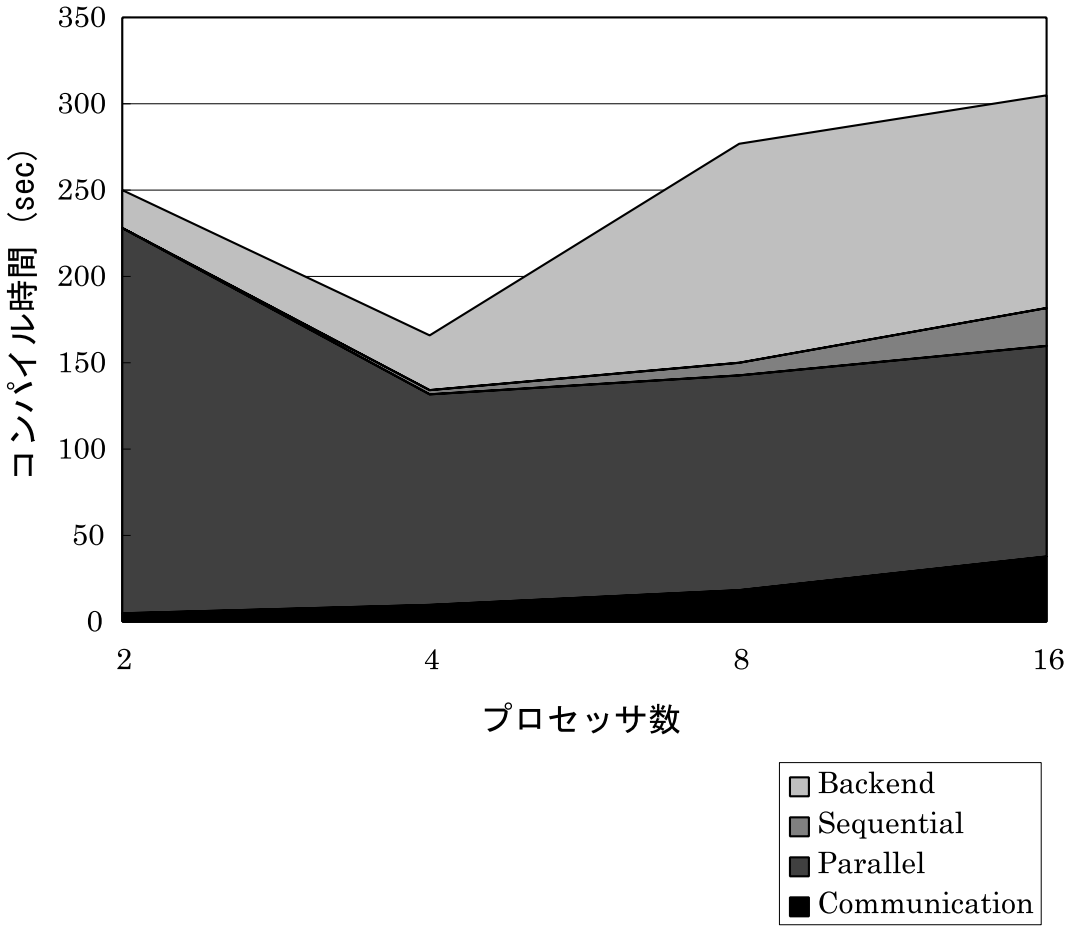


図 5.2: FT-classA PC クラスタ版コンパイル時間

インスペクタはプロセッサ 1 台分の範囲だけ調べる．このためコンパイルもターゲットマシンのプロセッサによる台数効果が得られる．また, 1PC 版は生成されるコードがプロセッサ台数によって増加するということがないので, ほぼ定数の時間でバックエンドの処理ができた．

PC クラスタ版のコンパイラのクラスタノード間で行なわれる通信はメインマシンと各ノード間で行なわれる．このため, 通信量の増大はコンパイラ自体の並列性を低下させる (図 5.6)．BT では 16 プロセッサで 5%の並列性しか得られなかったが, FT, pde1 では 16 プロセッサでも 40〜50%の並列性を得ることができた．

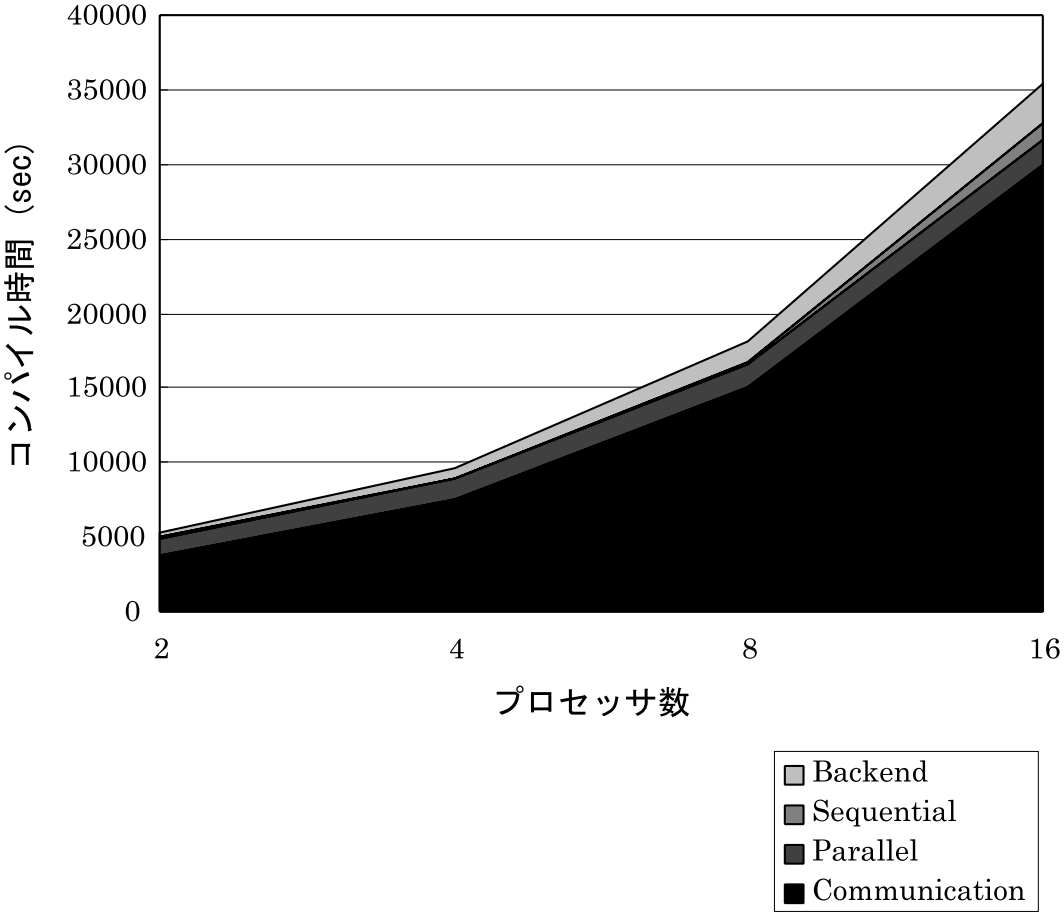


図 5.3: BT-classA PC クラスタ版コンパイル時間

5.3.3　商用コンパイラとの比較

ここでは, HPF ディレクティブが足りない状況で本方式と商用コンパイラとの実行時間の差を示す. (図 5.7, 5.8, 5.9) . ただし, スピードアップの基準となるプロセッサ数 1 の実行時間は, 通信のない逐次なコードの実行時間を用いた. ここで用いた商用の HPF コンパイラは日立製のもので, Pilot-3 用のものである. (バージョン 02-05) によってただし, 日立製の HPF コンパイラによる BT の実行時間は 1/10 規模で実行し, 計算で補間している.

本研究のコンパイラによる結果は, 日立製のコンパイラと比べて明確に良い結果を示した. 実行時間とスケーラビリティの両方に関して, 本方式が優れていた. 理由は以下の通りである. 日立製のコンパイラは典型的な最適化を行なうコンパイラである. またこのため, HPF 命令によるプログラマのヒントが少ない場合は, 効

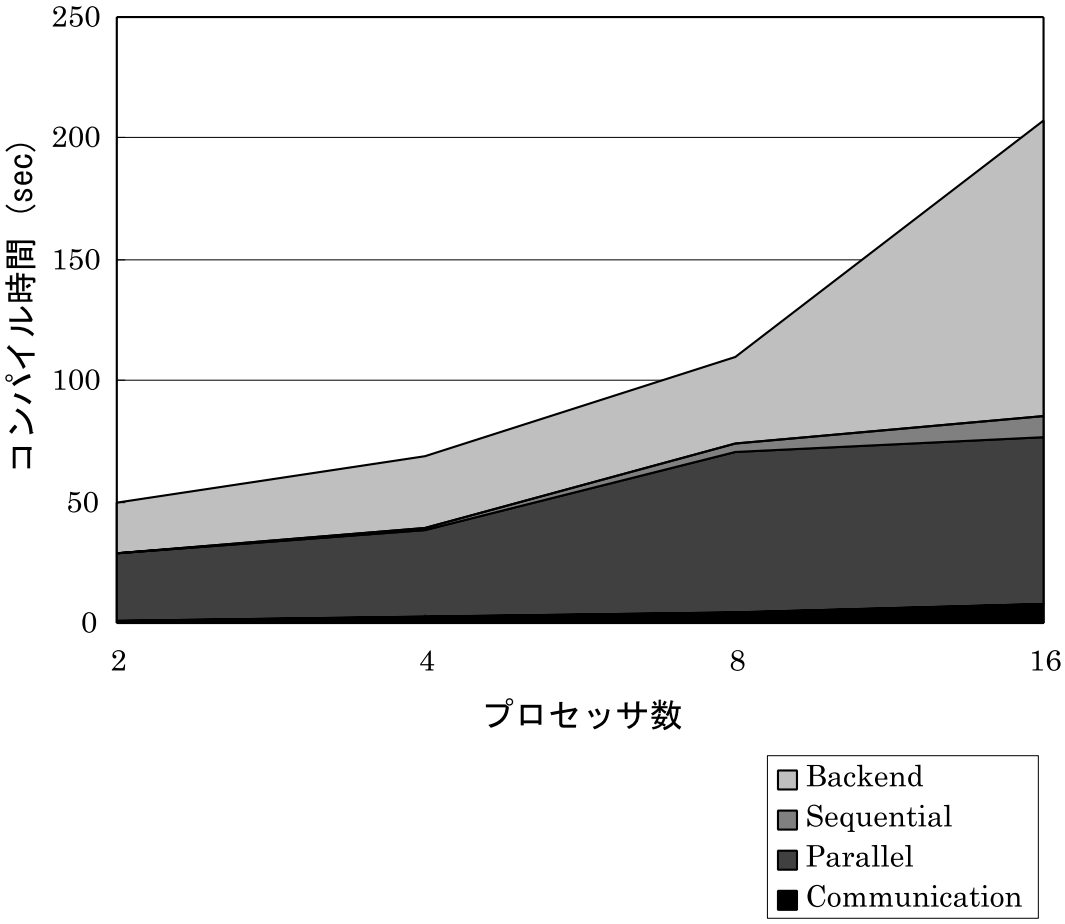


図 5.4: pde1, N=7 PC クラスタ版コンパイル時間

果的なコードを生成できない．今回, 実験に用いたベンチマークでは HPF のヒントが不足であった．しかし, 本研究で対象にするプログラムは物理, 天文などのシミュレーションプログラムのように, プログラマがコンピュータの専門家でないことを前提としている．このため, 今回用いた HPF 命令の少ないベンチマークは妥当であると考えている．

5.3.4　コード最適化の効果

図 5.10 は通信の最適化を実行時に処理したものとの比較である．本方式の特徴は, 実行時の情報をコードの最適化に利用する点である．そこでこの節では, コードの最適化でどの程度の効果があるのか, 実行時に解析を行ない, その結果でパラ

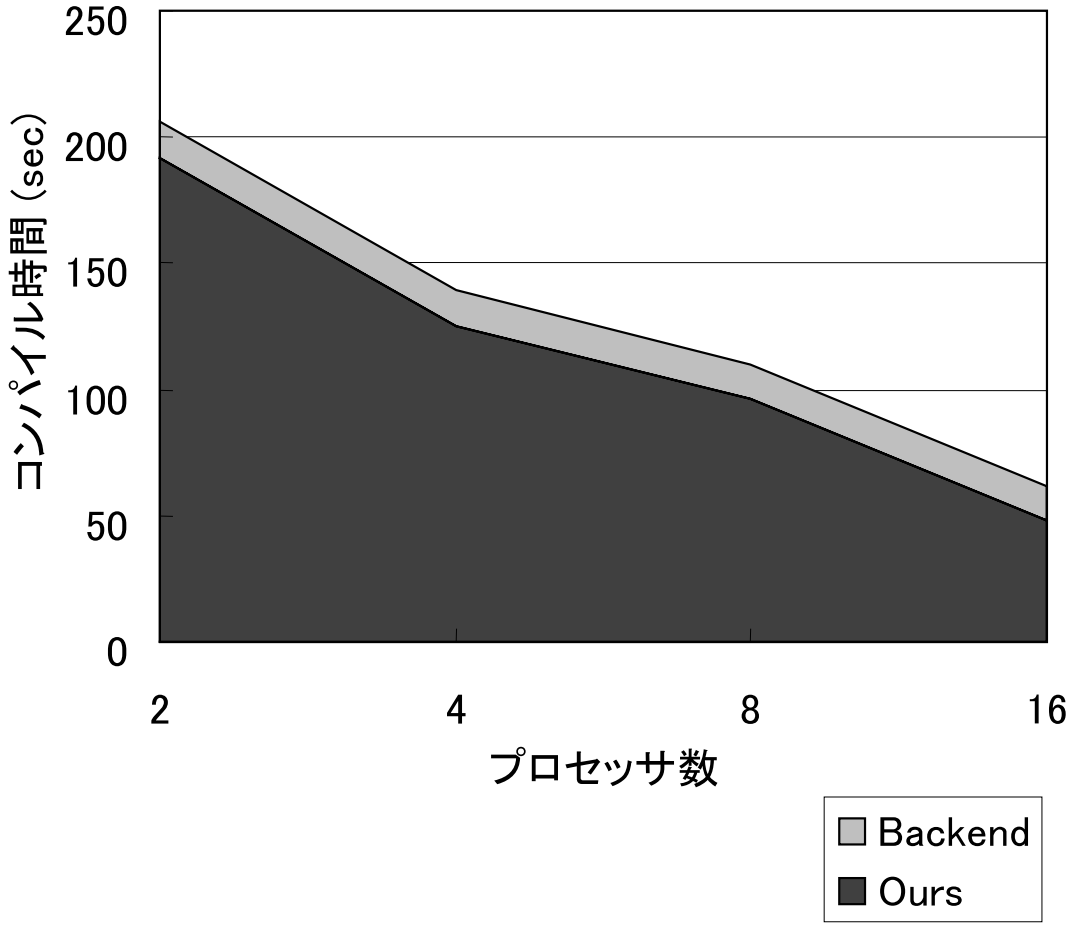


図 5.5: pde1, N=7 1PC 版コンパイル時間

メータを調整して実行時に通信を最適化する手法と比較した. 図 5.10 のコード最適化なしがこの手法である.

最適化を実行時にやることで, 本手法で用いた最適化がいくつか利用できなくなる. 図 5.10 のコード最適化なしはは 4.7.3 節で説明した技術と同様の最適化が行なわれている. しかし, 定数の畳み込み, ループの反復の最適なプロセッサに分配などは行なわれていない. これらの最適化はコード最適化なしに実現することが難しい. コード最適化なしの手法では実行時に複数のメッセージを一つのメッセージにまとめる. 図 5.9 で示されるように, 本研究のコンパイラはこの方式に比べプロセッサ数 16 で 1 台の PC 版で 19%, PC クラスタ版で 4. 9%高速であった.

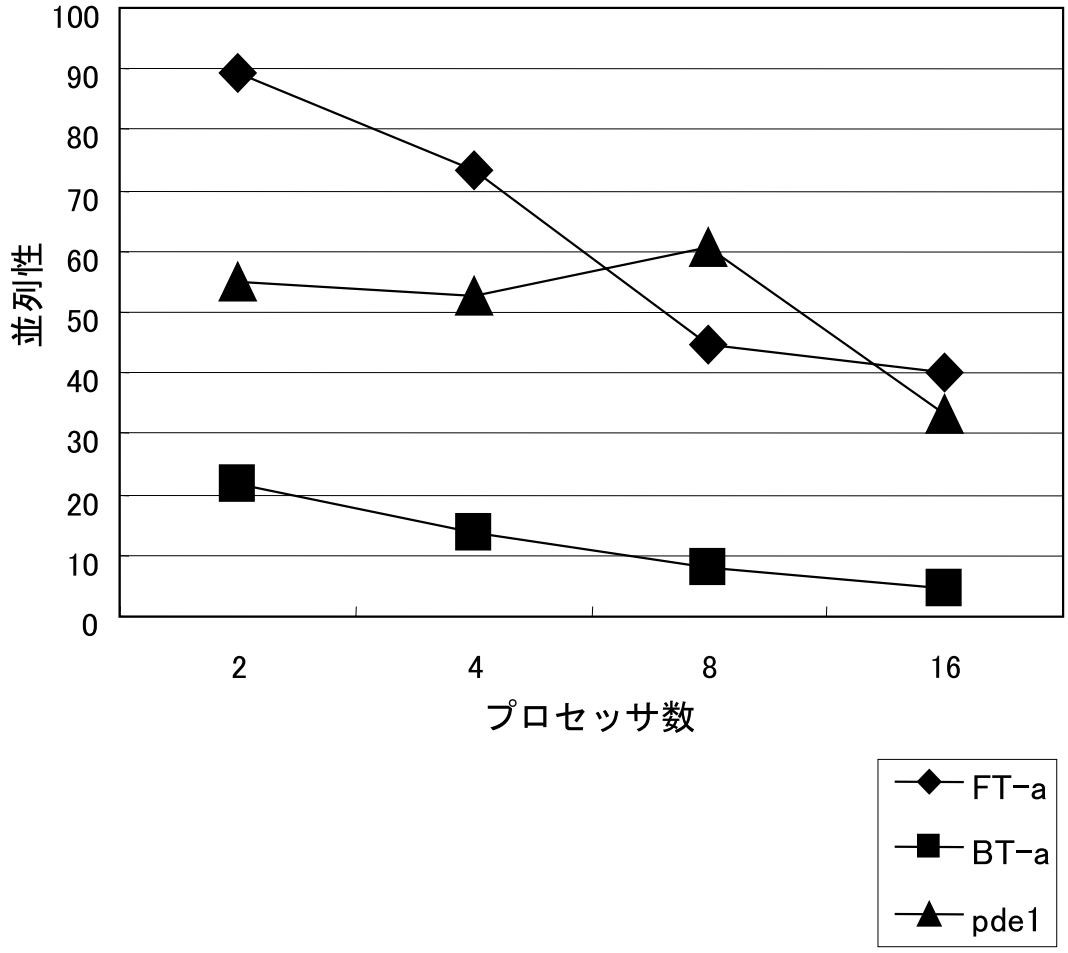


図 5.6: PC クラスタ版コンパイルの並列性

5.3.5　TCW 再利用型通信の効果

実装したコンパイラの最適化の効果を調べるため, TCW 再利用型通信を全く使用しなかった場合と使用した場合の実行速度の差を調べた. この最適化は実行時の情報を用いておこなうことが難しい. なぜなら, TCW 再利用型の通信は, 同じ通信相手 (同じプロセッサ, 同じアドレス, 同じブロックサイズ等) に対して繰り返し通信が行なわれることをコンパイラがコンパイル時に把握しなければ利用が難しいからである. この処理を実行時の情報を用いて行なうことが本方式の特徴の一つである.

図 5.11 はPC クラスタ版のコンパイラでTCW の再利用を用いたコードのスピードアップと用いなかったコードのスピードアップの比較である. どちらもスピード

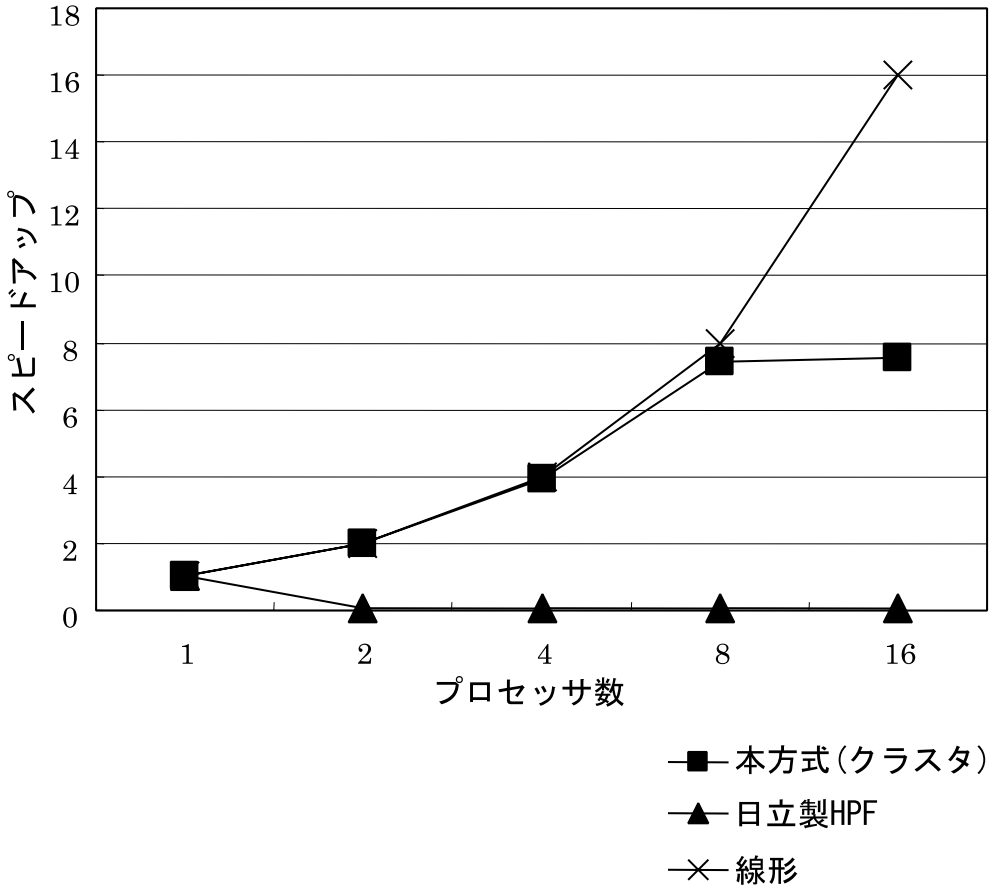


図 5.7: FT-classA 速度向上比

アップの基準となるプロセッサ数 1 の実行時間は, 通信のない逐次なコードの実行時間を用いた. この最適化の成果はプロセッサ数 2, 4 で 3%程度しか得られなかった. また, プロセッサ数 16 では逆に 0. 8%程度遅くなってしまった (表　5.3). これは TCW を再利用する効果が通信時間の全体に比べて大きくないからだと考えられる. 図 5.12 は TCW を再利用した場合と再利用しなかった場合の単純なデータ転送時間の比較である. 1〜5Mbytes のデータを 10000 回転送するのに必要な時間を比較している.

5.3.6　ブロックストライドの効果

実装したコンパイラの最適化の効果を調べるため, ブロックストライド通信をを全く使用せず, ブロックとループの組合せで処理した場合と, ブロックストライド

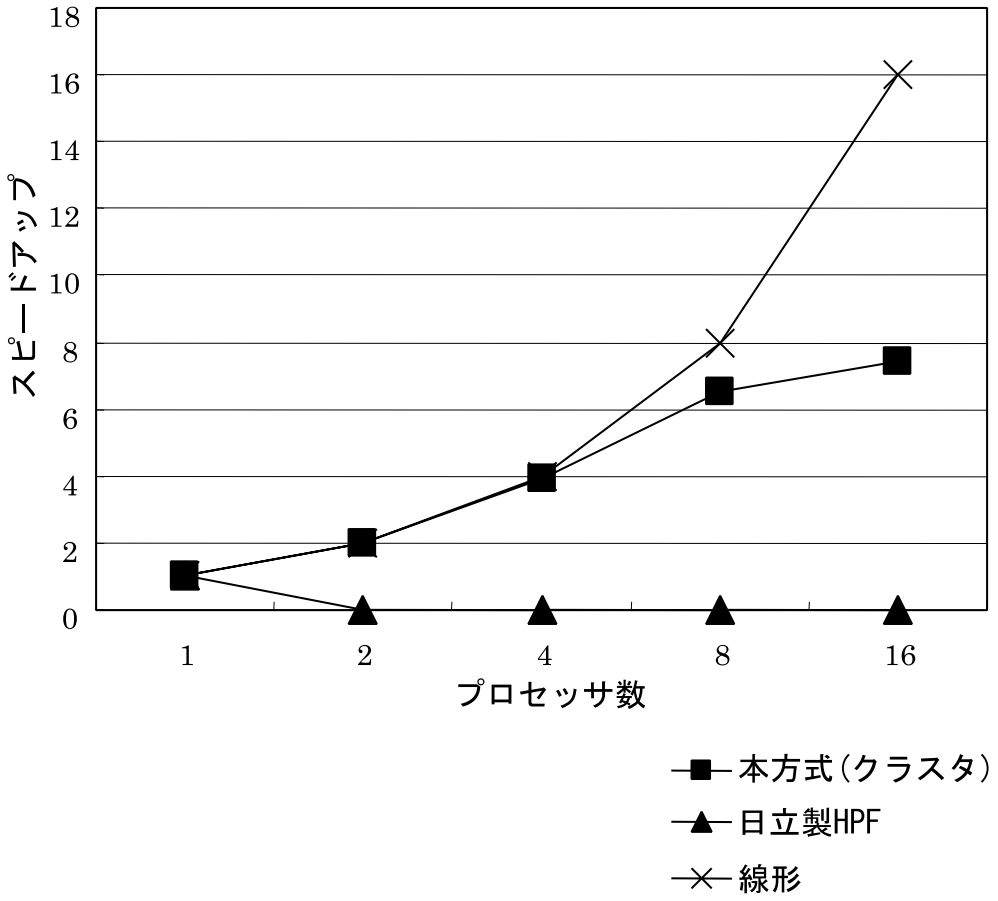


図 5.8: BT-classA 速度向上比

通信を使った場合の実行速度の差を調べた. pde1 では 126 回ブロックを繰り返すブロックストライド通信を利用する. RDMA はブロックストライド通信を使うことで, 核になる計算の通信の回数を 126 分の 1 に減らすことができる.

図 5.13 はPC クラスタ版のコンパイラでブロックストライド通信を用いたコードのスピードアップと用いなかったコードのスピードアップの比較である. どちらもスピードアップの基準となるプロセッサ数 1 の実行時間は, 通信のない逐次なコードの実行時間を用いた. ブロックストライドの利用はは 16 プロセッサで 18%の効果が出た. プロセッサ数が少ない実験では, 計算量に対して通信が少ないので影響が出にくかったと考えられる.

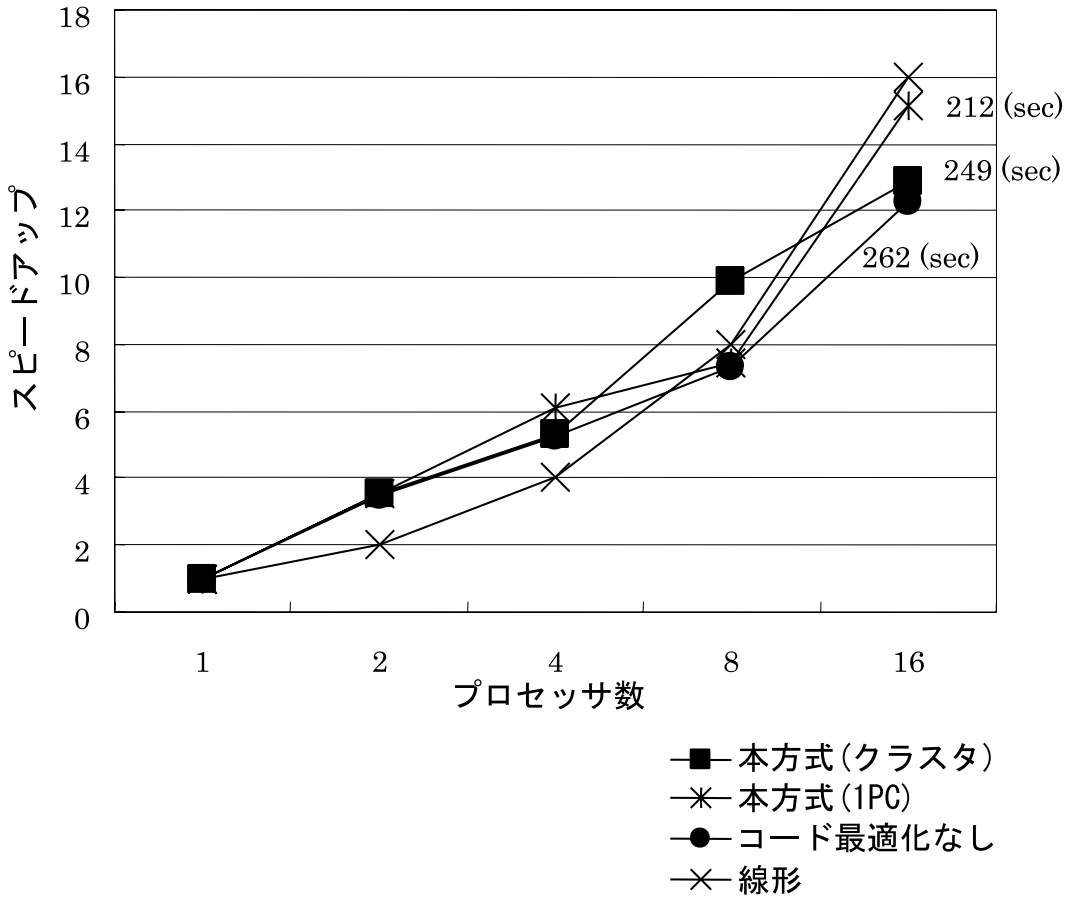


図 5.9: pde1, N=7 速度向上比

5.4　まとめ

本章では, 開発したコンパイラで実際に pde1, FT, BT 等のベンチマークをコンパイルし, 得られたコードの実行時間とコンパイル時間を調べた. また, TCW を再利用する効果やブロックストライド単位で送信することによる通信回数の削減の効果などを確かめた. さらに, 広く使われている使いやすい通信ライブラリを使って, 人間の手で最適化されたベンチマークとも比較した.

MPI を用いて人の手で最適化されたコードは本手法より高速であるという結果が得られた. 本方式と MPI を比較すると, 1PC 上で動作するコンパイラで 86%, PC クラスタ上で動作するコンパイラで 73%の速度となった. しかし, 本方式は PVM を用いて最適化されたコードを記述するより良好な結果を得ることができた.

通常のインスペクタ-エグゼキュータ方式を拡張して, 実行時に通信を最適化す

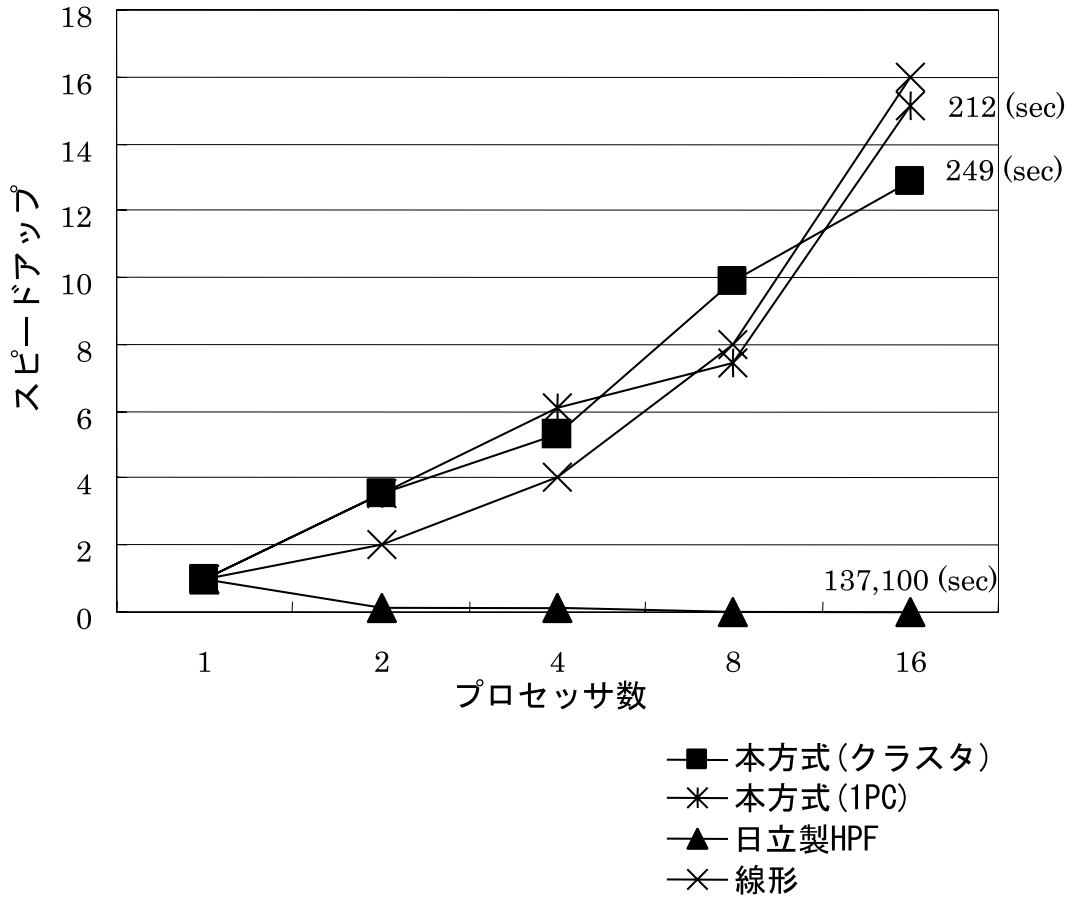


図 5.10: pde1, N=7 速度向上比

るようにしたものに比べ, 本方式の二つのコンパイラの両方が高速であった. しかし, 本方式ではコンパイル中にインスペクタを行なうので, コンパイル時間を含めた上で比較すると, 動的に最適化を行なうインスペクタ-エグゼキュータ方式の方が高速であった. しかし, 計算上 $OUTER$ ループの反復回数が十分であればコンパイル時間を回収でき, 計算上は 1PC 版では 1100 回, PC クラスタ版では 9400 回の反復数があれば回収できる (デフォルトは 1000 回).

本方式によるコンパイラの最適化のうち, TCW の最利用による効果はわずかであることがわかった. これは 0〜3%程度であった.

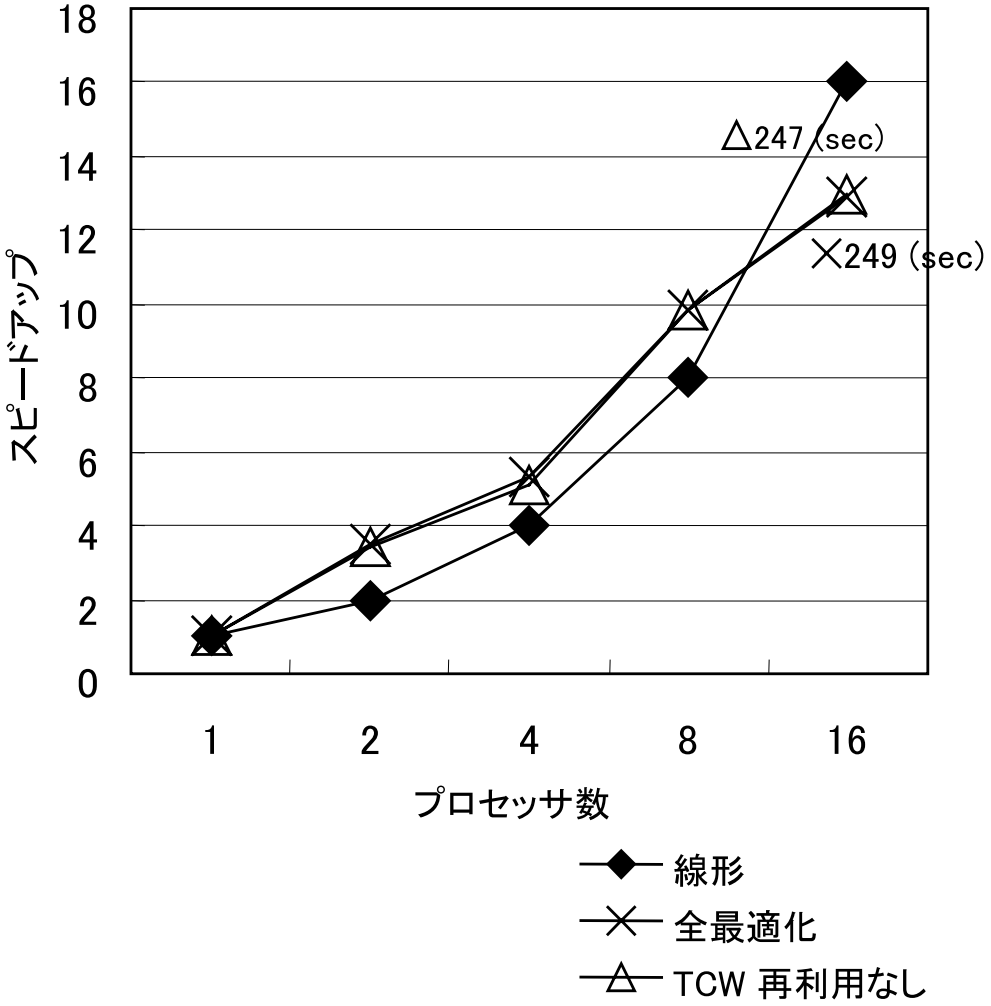


図 5.11: pde1, N=7 速度向上比 (TCW の再利用なし)

表 5.3: TCW を再利用する効果 (pde1, N=7)

プロセッサ数	2	4	8	16
TCW 再利用による実行時間短縮 (%)	2. 5	3. 5	0. 0	-0. 8

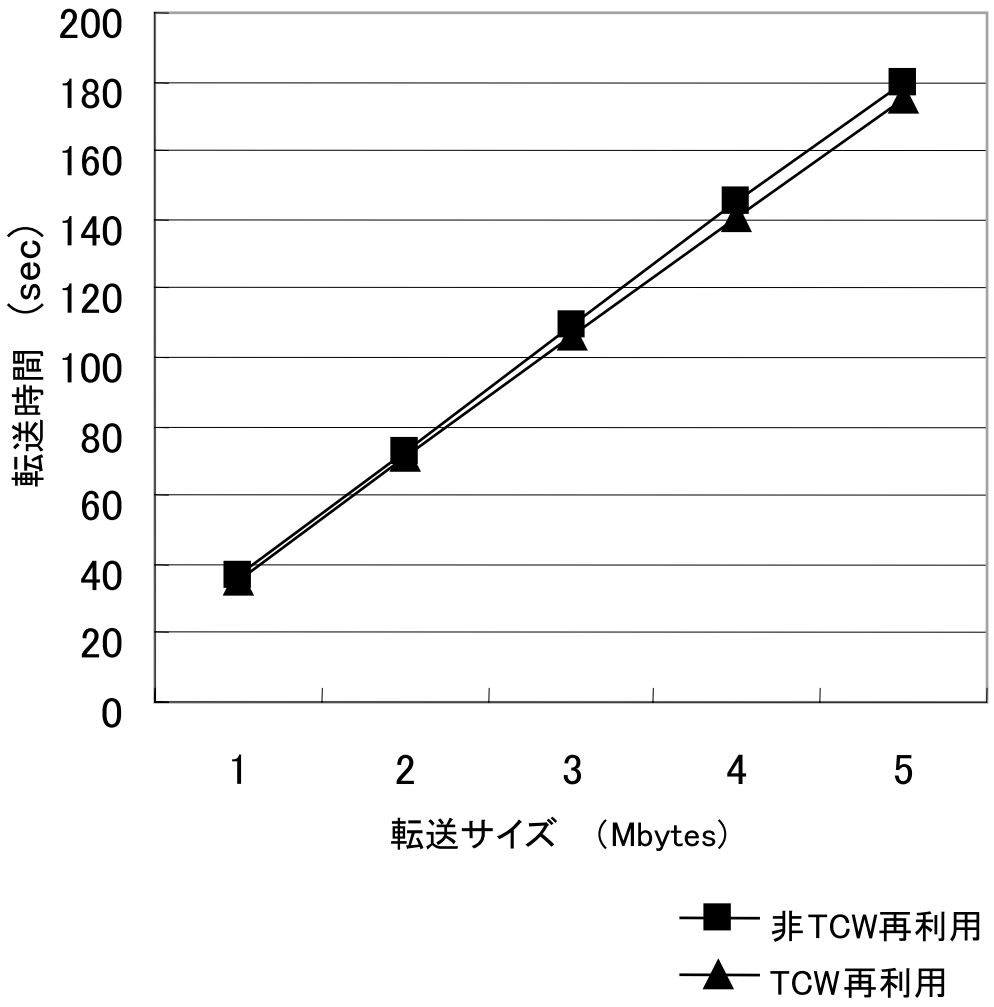


図 5.12: RDMA 転送時間 (10000 回繰り返し)

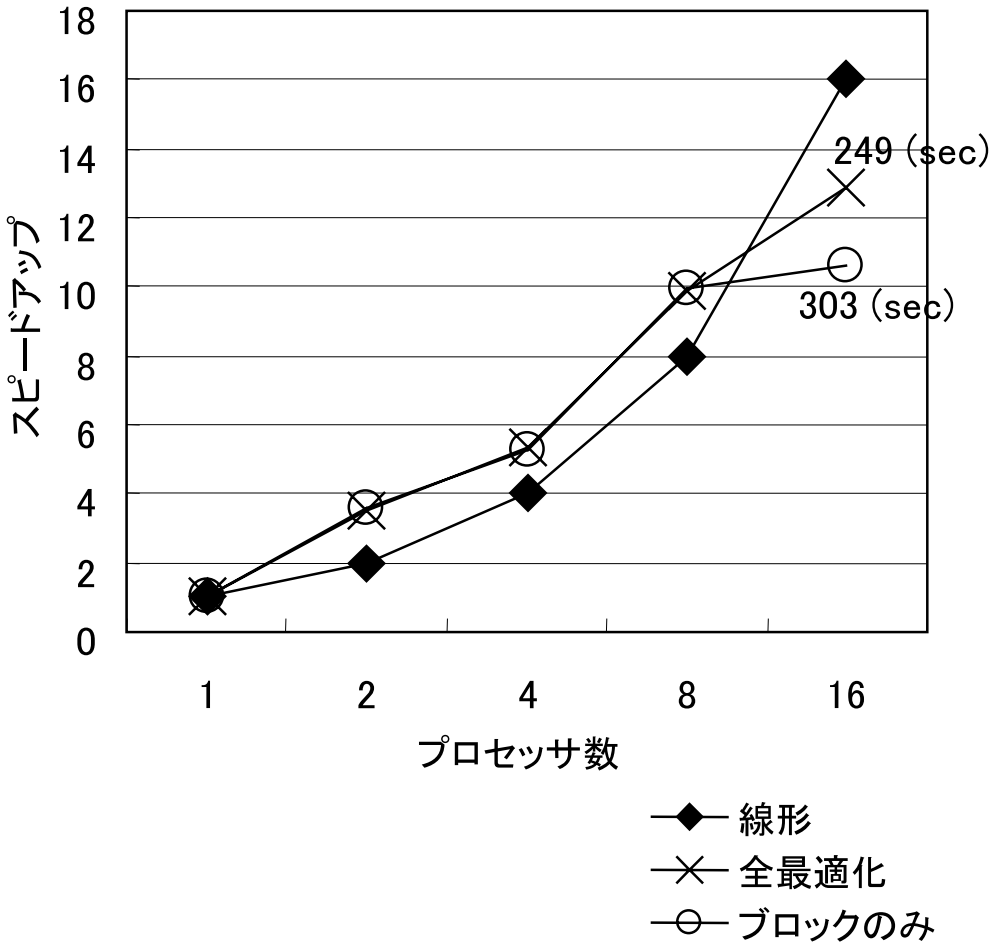


図 5.13: pde1, N=7 速度向上比 (ブロックストライドの利用なし)

第6章　まとめ

本論文では, 開発した一部の HPF 命令をサポートする Fortran コンパイラについて説明した. このコンパイラは, HPF 命令で分散を指定された配列へのアクセスを改良するために, 拡張されたインスペクタ-エグゼキュータ方式を用いる. 本研究のコンパイラの特徴は, コンパイル時にインスペクタを実行することである. コンパイラは, インスペクタが調べたプロセッサ間のデータの依存の情報を利用する. そして, インスペクタを含まずエグゼキュータだけを含んだ実行コードを生成する. 本手法を使うことで, インスペクタで求めたプロセッサ間のデータの依存の情報を静的に利用することができる. コンパイラはこの情報で実行コードを最適化する.

実装したコンパイラは CP-PACS や Pilot-3 のがターゲットマシンでいくつかの HPF ディレクティブと独自のディレクティブを一つ持った Fortran77 で書かれたプログラムを受理し, 通信が最適化されていて, RDMA で記述された Fortran コードを生成する.

最適化の対象は主に通信である. メッセージをブロックストライド通信を使って融合して通信の回数を減らし, 実行時の情報を定数の畳み込みで伝搬して動的な手法にあるテーブルの参照などを除去し, 可能な場合は TCW を再利用するコードを生成する. また, PC クラスタ版はループをプロセッサに分割する際, 通信量が最小になるように繰り返しをプロセッサへ分配する.

実行時の情報を用いて通信を最適化するコンパイラ ORE を実装した. 実装したコンパイラは簡便性を重視し 1 台の PC の上で動作するものと, 汎用性を重視して PC クラスタ上で動作するものである. 1 台の PC の上で動作するコンパイラは得られるコードは高速であるが, 全てのプロセッサについて, 同じデータの依存を持ったプログラムしかコンパイルできなかった. このため不規則な通信パターンを必要とする場合, 厳しい制限となった. それに対し, PC クラスタ上で動作するコンパイラは, より汎用的でこの制限がない.

本論文は, この最適化と実験の結果を説明した. 実験の結果, 実行時間を大きく短縮することができたが, コンパイル時間が増加した. さらに, 本研究のコンパイラは $OUTER$ ループの繰返しで, プロセッサ間のデータの依存が変わらないプログ

ラムしかコンパイルできない. なぜなら, 実行コードにはインスペクタが含まれていないからである. しかしながら, この問題は対象とするアプリケーションでは重要な問題ではない. 自然科学のシミュレーションなどではこの制限を満たしているものが多いからである.

また, 従来の手法に基づいて最適化を行なったものと比べて, 本手法がどの程度利点を得られるか実験で比較した. 比較した対象は, 動的な手法に基づいてランタイムでプログラムの挙動を調整し通信を最適化したコードと, 広く一般的に使われている通信ライブラリを用いて人間の手で最適化されたコードである. 動的な手法だけで行なう最適化は, 実行時の情報を定数の畳み込みで伝搬することが難しいことや, TCW の最利用のような最適化を盛り込みにくい点があるが, 本手法のようにコンパイル時間が増加する危険性がない. ベンチマークを用いて行なった実験では, コンパイル時間を含めると, ランタイムで最適化したコードの方が高速であったが, ベンチマークプログラムの $OUTER$ ループの繰り返しが不十分であったためであると考えている. 実際のシミュレーションプログラムではより多い数の $OUTER$ ループの繰り返し数を期待できる. 広く一般的に使われている通信ライブラリは, 計算物理のシミュレーションでも良く使われている. このようなライブラリを使うと RDMA のようにハードウェアに近く扱いにくいインターフェースを使う必要がないが機能が制限されてしまう. そこで代表的な通信ライブラリである MPI と PVM を用いてどこまで最適化ができ, どの程度の成果があるかベンチマークを用いて比較した. この結果, MPI を用いて最適化されたコードを生成することが最も効果的なコードを生成できることがわかったが, 利用者が計算機の専門家でないという点を考えると, 本コンパイラとの性能差は許容できるものと考えている.

本研究で全ての目標を達成することはできなかった. 今後に残された大きな課題が 3 つある. 実シミュレーションによる実験と, 人間が最適化した HPF によるプログラムとの比較と, 商用のコンパイラのように静的な解析を極めた場合との比較である. まず, 本研究ではターゲットが計算物理のシミュレーションであるにもかかわらず, 実験にはベンチマークを用いた. ベンチマークは物理計算を元に作成されたものであるが, 実際に計算物理で用いられたものではない. したがって, 本方式を検証するために実シミュレーションを行う必要がある. 次に, 本研究では一般的に並列プログラムに使われる言語やライブラリとして HPF, MPI, PVM をあげたが, 人間の手によって最適化されたプログラムとの比較は MPI と PVM だけしか行っていない. そこで同条件で HPF を用いた実験を行う必要がある. 最後に, 本研究では静的な解析と最適化を用いた手法との比較を十分に行っていない. 静的な解析を用いたアプローチとの十分な比較が必要である.

参考文献

[1] R. Das, M. Uysal, J. Saltz, and Y. S. Hwang : Communication optimizations for irregular scientific computations on distributed memory architectures *Technical Report CS-TR-3163, University of Maryland*, Oct. , 1993.

[2] S. D. Sharma, R. Ponnusamy, B. Moon, Y. S. Hwang, R. Das, J. Saltz : Run-time and compile-time support for adaptive irregular problems *In Proc. of Supercomputing'94*, pp 97−106, Nov. , 1994.

[3] C. Ding and K. Kennedy : Improving cache performance in dynamic applications through data and computation reorganization at run time *In Program. Language Design and Imple.* , pp. 229−241, 1999.

[4] C. Koelbel and P. Mehrotra : Compiling Global Name-Space Parallel Loops for Distributed Execution *IEEE Trans. on parallel and distr. systems*, Vol. 2, No. 4, pp. 440−451, 1991.

[5] M. Philippsen and B. Haumacher : Locality optimization in JavaParty by means of static type analysis *In Proc. Workshop on Java for High Performance Network Computing at EuroPar '98, Southhampton*, Sep. , 1998.

[6] R. Ponnusamy, J. Saltz, A. Choudary, Y. S Hwang, and G. Fox : Runtime Support and Compilation Methods for User-Specified Irregular Data Distributions *IEEE Trans. on parallel and distr. Systems*, Vol. 6, No. 8, pp. 815−831, 1995.

[7] R. C. Whaley, J. J. Dongarra: Automated Empirical Optimization of Software and the ATLAS Project *In Parallel Computing*, Vol. 27, No. 1−2, pp. 3−25, 2001.

[8] R. Ponnusamy, J. Saltz, A. Choudhary : Runtime-Compilation Techniques for Data Partitioning and Communication Schedule Reuse
In Proc. of Supercomputing '93, pp. 361−370, Nov. , 1993.

[9] G. Viswanathan and J. R. Larus : Compiler-directed shared-memory communication for iterative parallel computations
In Proc. of Supercomputing '96, Pittsburgh, PA, Nov. , 1996.

[10] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandani : Distributed memory compiler design for sparse problems
IEEE Trans. on computers, Vol. 44, No. 6, pp. 737−753, 1995.

[11] Michael Voss, Rudolf Eigenmann: Dynamically adaptive parallel programs
In proc. of Int'l Symposium on Highperformance Computing, springer, May. LNCS1615, pp. 109 − 120, 1999.

[12] Pedro Diniz, Martin Rinard: Dynamically feedback:An effective technique for adaptive computing
ACM SIGPLAN PLDI, pp. 71 − 84, 1997.

[13] Y. S. Hwang, B. Moon, S. Sharma, R. Ponnuswamy, R. Das, J. Saltz: Runtime and language support for compiling adaptive irregular programs on distributed memory machines
Software - Practice and Experience, 25, June 1995.

[14] S. J. Fink, S. B. Baden, S. R. Kohn: Flexible communication mechanisms for dynamic structured applications
IRREGULAR 1996, 1996

[15] S. T. Leung, J. Zahorjan: Improving the performance of run-time parallelization
ACM SIGPLAN PPoPP, pp. 83 − 91, May, 1993.

[16] A. M. Ghuloum, A. L. Fisher: Flattening and Parallelizing Irregular, Recurrent Loop Nests
ACM SIGPLAN PPoPP, pp. 58 − 67, May, 1995.

[17] K. A. Tomko, S. G. Abraham: Data and Program Restructuring of Irregular Applications for Cache-Coherent Multiprocessors
Proc. ICS94, pp. 214 − 225, 1994.

[18] N. Mitchell, L. Carter, J. Ferrante: Localizing non-affine array references
In Parallel Architectures and Compilation Techniques, Oct. , 1999.

[19] L. Huelsbergen: Dynamic Parallelization of Modifications to Directed Acyclic Graphs
Proc. of the Conference on Parallel Architectures and Compilation Techniques, pp. 186−197, 1996.

[20] L. Huelsbergen, J. Larus: Dynamic Program Parallelization
In Proc. of ACM LISP, pp. 311−323, 1992.

[21] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, K. Crowley: PARTI primitives for unstructured and block structured problems
Interim Report 22, Symposium on High Performance Computing for Flight Vehicles, June 25, 1992, 32 pps. , Dec. , 1992.

[22] A. L. Cox, S. Dwarkadus, H. Lu, W. Zwanapoel: Evaluating the performance of software distributed shared memory as a target for parallelizing compilers
Proc. IPPS'97, pp. 474−482, Apr. , 1997.

[23] S. Benkner, K. Sanjari, V. Sipkova, B. Velkov: Parallelizing Irregular Applications with the Vienna HPF+ Compiler VFC
Proc. HPCN'98, Springer Verlag, pp. 816−827, Apr. , 1998.

[24] D. K. Chen, J. Torrellas, P. C. Yew: An efficient algorithm for the run-time parallelization of DOACROSS Loops
Proc. of Supercomputing, pp. 518−527, 1994.

[25] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, W. Zwaenepoel: Compiler and software distributed shared memory support for irregular applications
ACM SIGPLAN PPoPP, pp. 48−56, Jun. , 1997.

[26] L. Rauchwerger, D. Padua: The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization
ACM SIGPLAN PLDI, pp. 218−232, Jun. , 1995.

[27] Y. Zhang, L. Rauchwerger, J. Torrellas: Hardware for Speculative Run-Time Parallelization in Distributed SharedMemory Multiprocessors
In Proc. of 4th International Symposium on High Performance Computer Architecture, pp. 162−173, Feb. , 1998.

[28] Y. Zhang, L. Rauchwerger, J. Torrellas: Hardware for Speculative Parallelization of Partially-Parallel Loops in DSM Multiprocessors
In Proc. of the 5th International Symposium on High Performance Computer Architecture, pp. 135 − 139, Jan. , 1999.

[29] 佐藤 三久, 建部 修見, 関口 智嗣, 朴 泰祐: 自動適応並列プログラム性能最適化ツールTEA Expert
情報処理学会 *HPC*研究会報告 pp. 13 − 18, 1998.

[30] 山崎 泰伯, 窪田 昌史, 津田 孝夫: Java クラスファイルの実行時ループ最適化手法
情報処理学会 *HPC*研究会報告 pp. 119 − 124, 2000.

[31] *High Performance Fortran Forum*
http://www. crpc. rice. edu/HPFF/home. html

[32] 坂上 仁志, 水野 貴夫: 国産HPF コンパイラの性能評価と互換性検証
情報処理学会 *HPC*研究会報告 pp. 73 − 78, 2001.

[33] *Message Passing Interface*
http://www. erc. msstate. edu/mpi/

[34] *MPI-J* メーリングリスト
http://www. ppc. nec. co. jp/mpi-j/

[35] 佐々木 まこ: *PVM*による並列プログラミング
http://gaia. tokai. jaeri. go. jp/activity/mvp/mvp/parallel/PVM/index. html

[36] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, Vaidy Sunderam: *PVM3 USER'S GUIDE AND REFERENCE MANUAL*, ORNL/TM-12187, September 1994.

[37] *Nas Parallel Benchmarks*
http://www. nas. nasa. gov/Software/NPB/

[38] J. Klose, M. Lemke, *GENESIS Distributed Memory Benchmarks*
http://www. pallas. de/

[39] *Portland Group*
ftp://ftp. pgroup. com/pub/HPF/examples

[40] 筑波大学計算物理学研究センター
http://www. rccp. tsukuba. ac. jp

[41] 日立製作所 汎用コンピュータ事業部 HPC 推進センタ: 並列計算機の最新技術動向-日立 SR2201 を中心に- 学術情報処理研究 No. 1, pp. 90 − 100, 1997.

[42] 筑波大学計算機システム運用委員会: 筑波大学計算物理学センター計算機システム利用の手引

[43] *Homepage of GRAPE group (Hongo), Department of Astronomy, University of Tokyo*
http://grape. astron. s. u-tokyo. ac. jp/

[44] 日立製作所: リモート *DMA* 転送 使用の手引 *-FORTRAN-*

[45] 日立製作所: *MPI・PVM* 使用の手引

[46] 日立製作所: *Parallel FORTRAN* 言語

[47] 日立製作所: 最適化 *FORTRAN90* 言語

[48] D. F. Bacon, S. L. Graham, O. J. Sharp: Compiler Transformations for High-Performance Computing
Computing Surveys, pp. 345 − 420 , Vol. 26, No. 4, 1994.